THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

GAME PROGRAMMING ALL IN ONE

2ND EDITION

JONATHAN S. HARBOUR

Premier
Press

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

GAME PROGRAMMING ALL IN ONE

2ND EDITION

JONATHAN S. HARBOUR

Premier
Press

This page intentionally left blank

Game Programming All in One, 2nd Edition

Jonathan S. Harbour

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

ISBN: 1-59200-383-4

Library of Congress Catalog Card Number: 2004091915
Printed in the United States of America

04 05 06 07 08 BH 10 9 8 7 6 5 4 3 2 1

THOMSON
COURSE TECHNOLOGY
Professional ■ Trade ■ Reference

Course PTR, a division of Course Technology
25 Thomson Place
Boston, MA 02210
http://www.courseptr.com

SVP, Thomson Course Technology PTR:
Andy Shafran

Publisher:
Stacy L. Hiquet

Senior Marketing Manager:
Sarah O'Donnell

Marketing Manager:
Heather Hurley

Manager of Editorial Services:
Heather Talbot

Acquisitions Editor:
Mitzi Koontz

Senior Editor:
Mark Garvey

Associate Marketing Managers:
Kristin Eisenzopf and Sarah Dubois

Project Editor/Copy Editor:
Cathleen D. Snyder

Technical Reviewer:
Joshua Smith

Thomson Course Technology PTR Market Coordinator:
Amanda Weaver

Interior Layout Tech:
Shawn Morningstar

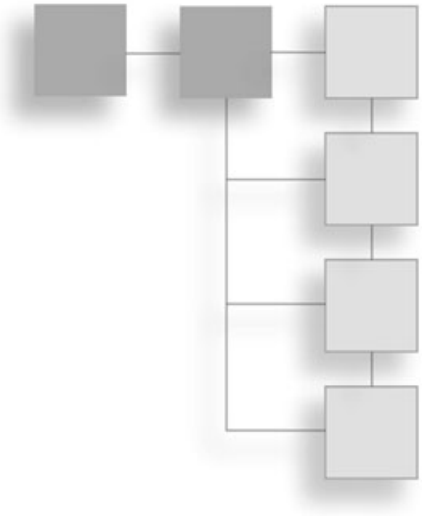
Cover Designer:
Steve Deschene

CD-ROM Producer:
Brandon Penticuff

Indexer:
Kelly Talbot

Proofreader:
Sean Medlock

For Jeremiah

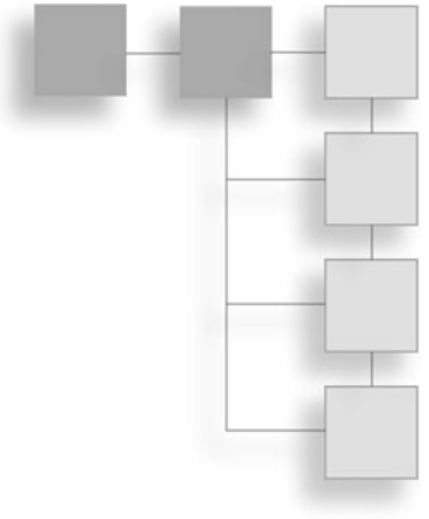


Acknowledgments

A book of this size involves a lot of work even after the writing is done. It takes a while just to read through a programming book once, so you can imagine how difficult it is to read through it several times, making changes and notes along the way, refining, correcting, and preparing the book for print. I am indebted to the hard work of the editors, artists, and layout specialists at Premier Press who do such a fine job. Thank you Mitzi Koontz, Emi Smith, and Stacy Hiquet for your encouragement and support.

I owe many thanks to Cathleen Snyder, one of the most amazing editors in the business, who both managed the project and copy edited the manuscript, and to Joshua R. Smith, who offered his technical expertise and long experience in the game industry to point out my mistakes and to offer advice. I believe you will find this a true gem of a game programming book due to their efforts.

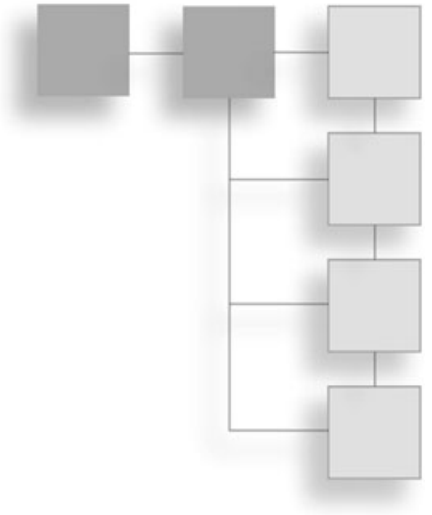
I would also like to thank Bruno Miguel Teixeira de Sousa for writing the first edition of this book. Some of his original work may still be found in this new edition, in Chapters 6, 18, 19, and 20.



About the Author

Jonathan S. Harbour has been an avid gamer and programmer for 17 years, having started with a TI-99, a Commodore PET, and a Tandy 1000. In 1994, he earned a bachelor of science degree in computer information systems. He has since earned the position of senior programmer with seven years of professional programming experience. Jonathan is a member of the *Starflight III* team, working with the original designers and other volunteers on a sequel to the first two *Starflight* games (using Allegro), originally published by Electronic Arts in 1985 and 1989, respectively. Jonathan has released two retail Pocket PC games, *Pocket Trivia* and *Perfect Match*, and has authored or coauthored five other books on the subject of game programming, including *Pocket PC Game Programming*, *Visual Basic Game Programming with DirectX*, *Visual Basic .NET Programming for the Absolute Beginner*, *Beginner's Guide to DarkBASIC Game Programming*, and *Beginning Game Boy Advance Programming*. He maintains a Web site dedicated to game programming and other topics at http://www.jharbour.com. Jonathan lives in Arizona with his wife, Jennifer, and children, Jeremiah and Kayleigh.

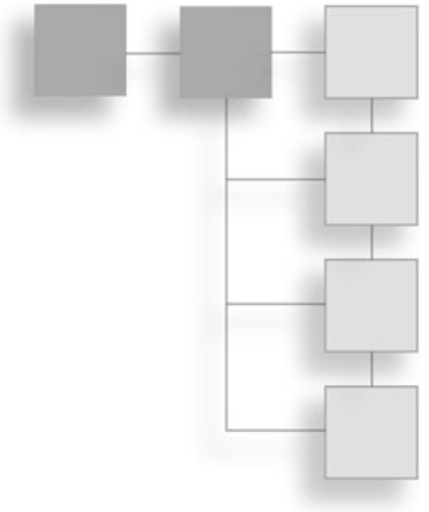
Contents at a Glance



Contents

Part II: 2D Game Theory, Design, and Programming 185

This page intentionally left blank



Introduction

Greetings! This book is the second edition to the best-selling *Game Programming All in One* by Bruno Miguel Teixeira de Sousa, to whom I am indebted for the original work. This new second edition is a complete rewrite of *Game Programming All in One*, with a completely new direction, new goals, new assumptions, and new development tools. *All in One 2E*, as I have come to call it, has done away with the C++ tutorials, Windows programming tutorials, and DirectX tutorials. In fact, this book does not cover Windows or DirectX at all. Instead, this book focuses on the subject of game programming using a cross-platform game library called Allegro. This library is extremely powerful and versatile. Allegro opens up a world of possibilities that are ignored when you focus specifically on Windows and DirectX. A full quarter of the first edition was devoted to a C++ language primer, while another fourth of the book focused on Windows and DirectX basics. I decided that for this second edition, we did not need to cover those subjects again; thus, this book uses the standard C language, and the sample programs will compile on multiple platforms.

The Windows version of Allegro uses DirectX, as a matter of fact, but it is completely abstracted and transparent, hidden inside the internals of the Allegro game library. Instead, you are provided with a basic C program that includes the Allegro library and is capable of running in full-screen DirectDraw mode using any supported resolution and color depth. Additionally, Allegro provides a uniform interface for sound effects, music, and device input, which are implemented on the Windows platform with DirectSound, DirectMusic, and DirectInput. Specifically, Allegro supports DirectX 8. Imagine writing a high-speed arcade game using DirectX, and then being able to recompile that program (without changing a single line of code) under Linux, Mac OS X, Solaris, FreeBSD, IRIX, and other operating systems! Allegro is a cross-platform game library that will double or triple the user base for the games you develop with the help of this book, and at no loss in performance.

Cross-Platform Game Programming

This book will teach you to write complete games that will run on almost any operating system. Specifically, I focus on three compilers—Visual C++, Dev-C++, and KDevelop—and the sample programs will be written using both Windows and Linux, with screenshots taken from both operating systems. In all likelihood, you will have the opportunity to use your favorite development tool because Allegro supports several C compilers, including Borland C++, Borland C++Builder, Apple Development Tools 2002, and several other compilers on various platforms, including the ubiquitous GNU C++ (GCC).

The target audience for this book is beginning to intermediate programmers who already have some experience with C or C++. Also, those who want to learn to write games using C or C++ can use this book as an entry-level guide. The material is not for someone new to programming—just someone new to *game programming*. I must assume you have already learned C or C++ because there is too much to cover in the game libraries, interfaces, and so on to focus on the basic syntax of the actual language. It was difficult enough to support three different compilers and integrated development environments without also explaining every line of code. Intermediate-level programming experience is assumed, while extreme beginners (newbies) will definitely struggle.

In Appendix D, "Recommended Books and Web Sites," I recommend introductory books for those readers. I encourage you to keep a C primer handy while reading through *All in One 2E* because this book moves along at a rapid pace. My goal was not to cover a *lot* of information, but to quickly get into the *important* information you'll need to write good games. This book is not extremely advanced—the source code is straightforward, with no difficult libraries to learn per se, but I do not explain every detail. I do cover most of the function library in Allegro, since that is the focus of this book, but I do not explain any standard C functions. The goal is to get up and running as quickly as possible with some game code! In fact, you will be writing your first graphics programs in Chapter 3 and your first game in Chapter 4. You will, however, quickly ramp up to advanced topics, such as creating game levels and scrolling the game world on the screen, with sample code, such as the *PlatformScroller* program (see Chapter 14).

Yes, it is true, this book focuses entirely and exclusively on 2D games. This is a huge genre that includes many real-time and turn-based strategy games, such as *Civilization III,* the *Age of Empires* series, *Diablo*, *Starcraft*, and so on. If you scoff at 2D games, then I encourage you to pick up *3D Game Programming All in One* (Premier Press, 2004) instead of (or in addition to) this volume. I make no apologies for ignoring 3D because these two books were designed to complement each other in the *Game Development* series.

Someone who has done some programming in Visual C++, CodeWarrior, Watcom C, Borland C++, GNU C++, or even Java or C# will understand the programs in this book.

Those with little or no coding experience will benefit from a C primer before delving into these chapters. I recommend many good C primers and C programming books in Appendix D. The emphasis of this book is on a cross-platform, open-source compiler, integrated development environment, and game library. You will not need to learn Windows or DirectX programming, and these subjects are not covered. The primary IDE is an open-source (freeware) program called Dev-C++, released by Bloodshed Software (http://www.bloodshed.net), and it is included on the CD-ROM. The game library is called Allegro; it is also freeware, open-source, and included on CD-ROM that accompanies this book. You have all the free tools you need to run the programs in the book, and then some! Using these tools, you can write standard Windows and DirectX programs with or without Allegro, and without the cost of an expensive compiler, such as Visual C++. This book is highly accessible to all C programmers, regardless of their platform of choice.

Use Your Favorite Compiler

Dev-C++ is a capable compiler package that includes an editor with source code highlighting. It uses the infamous GNU C++ compiler (GCC) to convert your chicken-scratch code into real programs with targets for Win32 or console programs and full support for DirectX 9. In other words, you might find Dev-C++ a useful companion for writing games with or without Allegro, and many of the sample programs in other Premier Press game development books will compile with Dev-C++ as well. It is a worthy, free, and easy-to-use alternative to a commercial compiler.

This book's source code and sample programs will run without modification on all of the following systems: Windows 9x/2000/Me/XP/2003, Mac OS X, Linux (any version), BeOS, QNX, and many other UNIX systems (IRIX, Solaris, Darwin, FreeBSD, to name a few) with X Windows. Believe it or not, these programs will also run under MS-DOS (DJGPP, Watcom C). That is almost every computer system out there. It's a sure bet if someone wants to use an old but mainstream C compiler, it will probably run the code in this book (with perhaps some limitations on compiling the Allegro library itself, which uses a modern makefile). I tell you this, not believing that you will need to write a game for MS-DOS, but just to demonstrate the versatility of Allegro.

Yet, at the same time, the Windows version of Allegro supports DirectX. The programs in this book will run in full-screen or windowed mode with support for just about any video card out there. Allegro is not an advanced, next-generation 3D engine; it is a cross-platform game library with a long history that dates back to the original Atari ST version. You might not care about cross-platform programming at this point, but imagine the possibilities if you were able to double the number of people who would play your game, just by compiling your game for other operating systems—and all without modifying any of your

source code. When is the last time you saw an online multiplayer game with Mac, Linux, and Windows players? Although I do not cover online multiplayer games in this book, they are a very real possibility using Allegro and standard TCP/IP socket libraries. As an example, I cover multi-threading in this book using a Windows port of the Posix Thread library, and the sample program I wrote to demonstrate multi-threading compiles under Windows and Linux without modification! The same is true for other libraries that conform to a standard, such as Berkeley Sockets for TCP/IP network programming.

This book is not *entirely* about cross-platform programming, though. I do discuss the subject in the first two chapters, but from that point forward, I simply focus on Allegro and specific game concepts, such as scrolling and animation. The overall theme and focus of this book are on writing games. To that end, you will develop a complete game and add to it in each chapter of the book, starting in Chapter 4.

Is This Book for You?

If you have any experience with the C language, then you will be able to make your way through this book. If you are new to the C language, I recommend against reading this book as your first experience with C because it will be confusing due to the extensive use of Allegro. (Very few standard C functions are used.) The example programs use a simple C syntax with no complicated interfaces or lists of include files. In fact, most of the programs will have a simple format like this:

```
#include "allegro.h"
int main(void)
{
    allegro_init();
    allegro_message("Welcome To Allegro!");
    return 0;
}
END_OF_MAIN();
```

This is a very simple program that is used as a test program for Appendix E, "Configuring Allegro for Microsoft Visual C++ and Other Compilers," just as an example. This program simply verifies that the Allegro library has been linked with the main program and is working as expected. This particular program outputs to the console and does not run in graphical mode. Allegro provides comprehensive support for all of the video modes supported on your PC, including full-screen and windowed DirectX modes used by most commercial games. On the UNIX side, Allegro supports the X Window system, SVGAlib, and other libraries (as appropriate to the platform), providing a similar output no matter which system it is running on. For instance, the `allegro_message` function is displayed in a pop-up message box in Windows, but prints a message to a terminal window in Linux.

If you are a Windows user and you don't care about Linux, that won't be a problem. The screenshots presented in this book look exactly the same no matter what operating system you are using, and my choice of Windows or Linux in each particular case is simply for variety. Likewise, if you are a Linux user and you care not for Windows, you will not be limited in any way because every program in this book is tested on both Windows and Linux. The CD-ROM that accompanies this book includes the complete source code for the sample programs in this book, with project files for Visual C++ (Windows), Dev-C++ (Windows), and KDevelop (Linux). The tools on the CD-ROM include both Windows and Linux versions in most cases. If you are using an operating system other than these two, you should have no problem adapting the source code to your compiler of choice.

Do you like games, and would you like to learn how to create your own professional-quality games using some of the same tools used by professional game developers? This book will help you get started in the right direction toward that goal, and you'll have a lot of fun learning along the way! This is a very practical programming book, not rife with theory, so you will find many, many sample programs herein to reinforce each new subject.

System Requirements

The programs in this book will run on many different operating systems, including Windows, Linux, Mac OS X, and almost any UNIX variant that supports the X Window system. All that is really required is a decent PC with a video card and sound card.

Here are the recommended minimum hardware requirements:

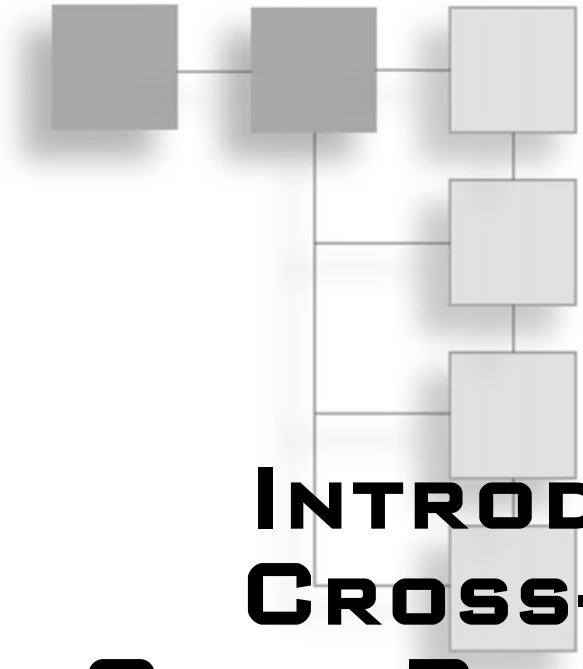
- Pentium II 300 MHz
- 128 MB memory
- 200 MB free hard disk space
- 8 MB video card
- Sound card

Book Summary

This book is divided into four parts:

- **Part I: Introduction to Cross-Platform Game Programming.** This first section provides all the introduction you will need to get started writing cross-platform games with Allegro and Dev-C++, with screenshots from both Windows and Linux. By the time you have completed this first set of chapters, you will have a solid grasp of compiling Allegro programs. This section concludes with a sample game called *Tank War* that you will enhance throughout the book.

- **Part II: 2D Game Theory, Design, and Programming.** This section is the meat and potatoes of the book, providing solid tutorials on the most important functions in the Allegro game library, including functions for loading images, manipulating sprites, scrolling the background, double-buffering, and other core features of any game. This section also provides the groundwork for the primary game developed in this book.
- **Part III: Taking It to the Next Level.** This section is comprised of more theoretical chapters covering basic artificial intelligence, a chapter on basic game physics, and a chapter about publishing your game.
- **Part IV: Appendixes.** This final section of the book provides answers to the chapter quizzes, a tutorial on numbering systems, a set of useful mathematical tables, tutorials on installing and using Allegro, a list of recommended resources, and an overview of the CD-ROM.

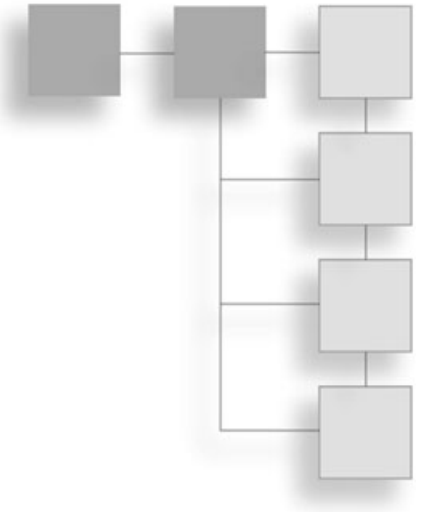


PART I

Introduction to Cross-Platform Game Programming

Welcome to Part I of *Game Programming All in One, 2nd Edition*. Part I includes five chapters that introduce you to the basic concepts of game development with Allegro. Starting with an overview of game development roots and covering the subject of motivation, this part goes into detail about how to use the free Dev-C++ compiler/IDE and the Allegro game programming library. Also, this part shows how to write, compile, and run several Allegro programs.

CHAPTER 1



Demystifying Game Development

This chapter provides an overview of the game industry, the complexities of game development, and the personal motivations that drive members of this field to produce the games we love to play. Herein you will find discussions of game design and how your world view and upbringing, as well as individual quirks and talents, have a huge impact on not only whether you have what it takes to make it big, but also whether it is a good idea to work on games at all. There is more to writing games than motivation. While some programmers see game development purely as a monthly salary, some perceive games at a higher level and are able to tap into that mysterious realm of the unknown to create a stunning masterpiece. In this chapter, I discuss that vague and intangible (but all too important) difference.

I also give you a general overview of what it is like to work as a programmer. If you are interested in game programming purely for fun or as a hobby, I encourage you to absorb this chapter because it will help you relate to those on the inside and judge your own creations. When you consider that it takes a team to develop a retail game—and you are an individual—it is not unreasonable to believe that your own games are high in quality and worthy of note. What you must consider are total invested project hours and the size of the team. How does your solo project compare to a team game development project? You see, your solo (or rather, "indie") game may be comparable to a retail game, all things being equal. One goal of this chapter is to help you realize this fact, to encourage you to continue learning, and to create games from your imagination. Whether you are planning a career in the game industry or simply partaking in the joy of writing games to entertain others, this chapter has something beneficial for you. After all, there are employed game programmers who only make their mark after going solo, and some solo game programmers who only make their mark after joining a team. Taking games seriously from the start is one way to attract attention and encourage others to take your work seriously.

Here is a breakdown of the major topics in this chapter:

- Gaining a high-level view of game development
- Recognizing your personal motivations
- Getting into the spirit of gaming
- Getting an introduction to Dev-C++ and Allegro

Introduction

Before I delve into the complexities of learning to write a game, I want to take a few moments to discuss the big picture that surrounds this subject. I'd like to think that some of you reading this book very likely will enter the game industry as junior or entry-level programmers and make a career of it. I am thrilled by that possibility—that I may have contributed in some small way to fulfilling a dream. I will speak frequently to both the aspiring career game programmer and the casual hobbyist because both have the same goals—first, to learn the tricks and techniques used by professional game programmers, and then to learn enough so it is no longer difficult and it becomes fun. Programming is difficult already; *game* programming is exponentially more difficult. But by breaking down the daunting task of writing a modern game, you can learn to divide and conquer, and finish a great game! Thus, my goal in this chapter is to provide some commentary along those lines while introducing you to the technologies used in this book—namely, the C language and the Allegro game library.

First, a disclaimer—something that I will repeat several times to nail the point home: DirectX is *not* game programming. DirectX is one library that is indisputably the most popular for Windows PCs. However, consoles such as the Sony PlayStation 2, Nintendo GameCube, Nintendo Game Boy Advance, and the many other handheld devices do not use DirectX or anything like it (although Xbox does use DirectX). There are dozens of DirectX reference books disguised as game programming books, but they often do little other than expose the interfaces—DirectDraw, Direct3D, DirectInput, DirectSound, DirectMusic, and DirectPlay. Talk about getting bogged down in the details! In my opinion, DirectX is the means to an end, not the goal itself. Learning DirectX is optional if your dream is to write console games (although I recommend learning as much as possible about every subject).

For the newcomer to game development, this misconception about the nature of some so-called game programming books can be a source of consternation. Beginners can be impatient (as I have been myself, and will discuss later in this chapter). Let me summarize the situation: You want to get something going quickly and easily, and *then* you want to go back and learn all the deep and complicated details, right? I mean, who wants to read an 800-page programming book before they actually get to write a game?

Practical Game Programming

This book focuses on the oft-misused phrase "game programming" and has no prerequisites. I don't discuss Windows or DirectX programming at all in this book. For some excellent reference books on those subjects (which I like to call *logistical* subjects), please refer to Appendix D, "Recommended Books and Web Sites." If I may nail the point home, allow me to present a simple analogy—one that I will use as a common theme in this and other chapters. Writing a game is very similar to writing a book. There are basic tools required to write a game (such as a compiler, a text editor, and a graphic editor), just as there are tools required to write a book (such as a word processor, a dictionary, and a thesaurus). When you are planning a new project, such as a game, do you worry about electricity? As such, when you are planning a new book, would you worry about the alphabet? These things are base assumptions that we take for granted.

I take the operating system completely for granted now, and I try to abstract my computing experience as much as possible. It is a liberating experience when I am able to get the same work done regardless of the electronics or operating system on my computer. Therefore, I take those things for granted, whether I am using Windows Explorer or GNOME, Internet Explorer or Mozilla, Visual C++ or Dev-C++. This is an important concept that I encourage you to consider because the game industry is in a constant state of flux that conducts the vibrations of the entire computer industry.

The concept of a "new computer" is important to the general public, but to a computer industry professional, "new" is a very relative term that only lasts a few weeks or months at most. Everyone has his or her own way of dealing with constant change, and it is part of the experience of working with computers. (Those who can't handle it never last long in this industry.) Rather than seeing change as a tidal wave and trying to keep ahead of it, I often let the wave crash over my head, so to speak, and wait for the next wave. It's an intriguing experience, allowing high technology to pass you up and zoom ahead. But do you know why there is some wisdom in skipping a trend now and then? Because technology is not only in a constant state of change, but it is also in a constant state of experimentation. Not every new "improvement" is good or accepted. Remember videodiscs? Probably not! The movie industry had to rethink videodisc technology in part because the discs resembled vinyl records, which the public perceived as old technology.

For example, the computer hardware industry markets heavily for the need to constantly upgrade computers. It is logical that these companies would do so because the general public really believes that everything is obsolete year by year. In fact, it is the gross inefficiency of the software that makes this so. Rather than grasping at the latest everything with a must-have belief system, why not continue to use known, stable systems and stand up to the frequent tidal waves of technology? What one calls progress, another calls marketing. Games have single-handedly pushed the personal computer industry to extraordinary new

heights in the past decade due for the most part to advances in graphics technology. But that cutting edge leaves a lot of well-meaning and talented folks out in the cold when they might otherwise be developing well-loved games.

So we come back to the point again: What is the cutting edge of game development, and what must I do to write great games? For the first part, the cutting edge is gameplay, not the latest 3D buzzword. Second, to write a great game, you must be passionate and talented. Studying the subject at hand (game programming) is another factor—although it is the focus of this book! For my own inspiration, I look at games such as *Sid Meier's Civilization III* and *Age of Wonders: Shadow Magic*, among other recent 2D titles. You can find your inspiration in whatever subject interests you, and it need not always be a video game.

Goals Revisited

One of the aspects of this book that I want to emphasize early on is that my goal is to reach a majority of hobbyists and programmers who are either aspiring to enter the game industry as career programmers or who are simply writing games for the fun of it. As I explained in the Introduction, this book won't hold your hand because there is so much information to cover. At the same time, it's my job to make a difficult subject easy to comprehend; if you have some fun along the way, that's even better. I don't want to simply present and discuss how to write 2D graphics code; my goal is for you to master it.

By the time you're finished with this book, you'll have the skills to duplicate any game released up to the late 1990s (before 3D hardware acceleration came along for PCs). That includes a huge number of games most often not regarded by the "twitch generation"—that is, those gamers who would describe "strategy" as which direction to circle strafe an enemy in a first-person shooter, the best kind of car to "jack" to make the most money, or how to escape via a side alley where the cops never follow you. We can poke fun at the twitch generation because they wouldn't know what to do with a keyboard, let alone how to write game code; therefore, they are not likely to read this book. But if there are any twitch gamers now reading, I congratulate you!

The High-Level View of Game Development

Game development is far more important to society than most people realize. Strictly from an economic point of view, the design, funding, development, packaging, delivery, and sale of video games (both hardware and software) employs millions of workers around the world. There are electronics engineers building the circuit boards and microprocessors. Programmers write the operating systems, software development kits, and games. Factory workers mass-produce the packaging, instructions, discs, controllers, and other peripherals. Technical support workers help customers over the phone. There are a large number of investors, business owners, managers, lawyers, accountants, human resource workers, network

and PC technical support personnel, and other ancillary job positions that support the game industry in one way or another. What it all amounts to is an extraordinarily complex system of interrelated industries and jobs, and millions of people who are employed solely to fill the shelves of your local video game store. The whole point of this is simply to entertain you. Because we're talking about high-quality interactive entertainment, we have a tendency to spend a lot of money for it, which increases demand, which drives everyone involved to work very hard to produce the next bestseller.

Although this narrative might remind you of the book publishing industry, where there are many people working very hard to get high-quality books onto store shelves, I submit that games might be more similar to motion pictures than to books. All three of these subjects are closely related forms of entertainment, with music included. Books are turned into movies, movies into video games, and both movies and video games into books. All the while, music soundtracks are available for movies and video games alike. Much of this has to do with marketing—getting the most income from a particular brand name. One excellent side effect of this is that many young people grow up surrounded by themes of popular culture that spawn their imaginations, thus producing a new generation of creative people every few years to work in these industries.

Consider the effect that science fiction novels and movies have had on visionaries of popular culture, such as Gene Roddenberry and George Lucas, who each pushed the envelope of entertainment after being inspired by fantastic stories of their time, such as *The Day the Earth Stood Still* and *The Twilight Zone*, to name just two. Before these types of programs were produced, Hollywood was enamored with westerns—stories about the old West. What was the next great frontier, at least for an American audience? Having spread across the continent of North America, and after fighting in two great and terrible world wars, popular culture turned outward—not to Earth's oceans, but to the great interstellar seas of space. What these early stories did was spurn the imaginations of the young up-and-coming visionaries who created *Star Trek*, *Lost in Space*, *Star Wars*, and action/adventure themes such as *Indiana Jones*, set in a past era (where *time* is often associated with *space*). These are identifiable cultural icons.

The game industry is really the next generation of entertainment, following in the footsteps of the great creative powerhouses of the past few decades. Games have been growing in depth and complexity for many years, and they have come to be so entertaining that they have eclipsed the motion picture industry as the leading form of entertainment. But just as movies did not replace books, neither will games replace movies as a dominant player. Although one might eclipse the others in revenue and profit, all of these industries are interrelated and interdependent.

Thinking hypothetically, what do you suppose will be the next stage of cutting-edge entertainment, the likes of which will supercede games as the dominant player? In my opinion, we have not seen it yet and we might never see it. I believe that books, music, movies, and

video games will continue unheeded to inspire, challenge, and entertain for decades to come. But I do hold an opinion that is contrary to my last statement. I believe that western society will embrace entertainment less and adopt more productive uses for games in the decades to come. Why do I feel this way, you might ask? Momentum and progress. Games are already being used for more than just entertainment. They are being used by governments to train soldiers in the strategy and tactics of a modern battlefield, one in which military commanders no longer have the luxury of experiencing for real. Without real long-term engagements like those during World War II (battles since that time have been skirmishes in comparison), modern militaries must rely on alternative means of training to give troops a feel for real battle. What better solution than to play games that are visceral, utterly realistic, shocking in unpredictability, and awe-inspiring to behold? Who needs a real battlefield when a game looks and feels almost like the real thing?

I have now explored several areas of our society that benefit from the game industry. What about gamers themselves—you, me, and other video game fanatics? We love to play games because it is exhilarating to conquer, pillage, destroy, and defeat an opponent (especially if he or she is a close friend or relative). But there is the converse to this point of view, regarding those games that allow you to create, imagine, build, enchant, and express yourself. Some games are so artistic that it feels as if you are interacting with an oil painting or a symphonic orchestra. To conclude this game brings forth the same set of emotions you feel upon finishing a good book, an exceptional movie, or an orchestral performance—exhilaration, joy, pride, fascination, appreciation, and yet a tinge of disappointment. However, it is that last emotion that draws you back to that book, movie, game, painting, or symphony again, where it brings you some happiness in life. This experience transcends mere entertainment; it is a joy felt by your soul, not simply a sensual experience in your mind and through your eyes and ears.

Interactivity has much to do with some of the new lingo used to describe the game industry. Although insiders won't mince words, those who are concerned with public consumption and opinion prefer to call the game industry a form of interactive or electronic entertainment. Game programming has become game development. Outlining the plot of a game has become game design. Very lengthy scripts are now written for games, and some designers will even storyboard a game. Do you begin to see similarities to the movie industry?

Storyboarding is a process in which concept artists are hired to illustrate the entire game scene by scene. This is a very expensive and time-intensive process, but it is necessary for complicated productions. Some films (or games, for that matter) are rather simple in plot: Aliens have invaded Earth, so someone must stop them! Although a storyboard might help a hack-and-slash type of game, it is often not necessary, particularly when the designer and developers are intimately familiar with the subject matter. For instance, think about

a game adaptation of a novel, such as Michael Crichton's *Jurassic Park*. The developers of a game based on a novel do not always have the benefit of a feature film, as was the case with *Jurassic Park* and other movies based on Michael Crichton novels. Simply reading the book and watching the movie is probably enough to come up with a basic idea for what should happen in the game; you probably don't need to storyboard.

Why do I feel that this discussion is important? It is absolutely relevant to game development! In fact, "game programming" has become such a common phrase in video game magazines, on Web sites, and in books that it is often taken for granted. What I'm focusing on is the importance of perspective. There is a lot more to consider than just what to name a program variable or what video resolution to use for your next game. You need to understand the big picture, to step away from the tree to see the entire forest.

Recognizing Your Personal Motivations

Why do you want to learn game programming? I want you to think hard about that question for a moment, because the time investment is great and the rewards are not always up to par in terms of compensation. *You must love it.* If you don't love absolutely everything about video games—if you don't love to argue about them, review them online, and play them obsessively—then I have some good but somewhat hard advice. Just treat video games as an enjoyable hobby, and don't worry too much about "breaking in" to the game industry or getting your game published. Really. Because that is a serious source of stress, and your goal is supposed to be to have fun with games, not get frustrated with them.

note

For a fascinating insider narration of the video game industry's early years, I highly recommend the book *Hackers* by Steven Levy, which puts the early years of the game industry into perspective. For a historic ride down memory lane, be sure to read *High Score! The Illustrated History of Electronic Games* by Rusel DeMaria and Johnny Wilson (former editor of *Computer Gaming World*), a full-color book with hundreds of fascinating photos. Browsing the local bulletin board systems in the late 1980s and early 1990s to download shareware games was also a fun pastime. For an intriguing look into this era, I recommend *Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture* by David Kushner.

I was inspired by games such as *King's Quest IV: The Perils of Rosella*, *Space Quest III: The Pirates of Pestulon*, *Police Quest*, *Hero's Quest: So You Want To Be A Hero?*, and other extraordinarily cool adventure games produced by Sierra. There were other companies, too, such as Atari, Electronic Arts, Activision, and Origin Systems. I spent many hours playing *Starflight*, one of the first games that Electronic Arts published in 1985 (and one of the greatest games made at the time) and the sequel, *Starflight II: Trade Routes of the Cloud Nebula*, which came out in 1989.

Decision Point: College versus Job

In the modern era of gaming, a college education is invaluable. What if you grow tired of the game industry after a few years? Don't cringe; this is a very real possibility. A lot of hardcore gamers have moved on to casual gaming or given it up entirely while pursuing other careers.

Focus every effort on writing complete and polished games, however big or small, and consider every game as a potential entry on your résumé. If you want to work on games for a living, go for it full tilt and don't halfheartedly fool around about it. Be serious! Go get a job with *any* game studio and work your way up. On the other hand, if you want to get involved in high-caliber games, then go to college and focus heavily on your studies. Let the game industry pass you by for a short time, and when you graduate, you will be ready and equipped to get a great job. There are some really great high-tech colleges that are offering game programming degree programs. University of Advancing Technology in Tempe, Arizona, for instance, has associate's, bachelor's, and master's degree programs in game development! Take a look at http://www.uat.edu.

Once you have made the decision to go for it, it's time to build your level of experience with real games that you will create on your own. Don't assume that one of your hobby games isn't good enough for an employer to see. Most game development managers will appreciate brimming enthusiasm if you have the technical skills to do the job. Showing off your previous work and recalling the joy of working on those early games is always enjoyable for you and the interviewer. They want to see your *personality*, your *love* of games, and how you spent *hundreds* of hours working on a particular game, fueled by an uncontrollable drive to see it completed. Your emphasis should be on completed games. Most important, always be genuine.

I would go so far as to say that having a dozen shareware games (of good quality) on your résumé is better than having worked on a small part of a commercial game. Yes, suppose you did work on a retail game. That doesn't guarantee choice employment with another company. What sort of work did you do on that game—level editors, unit editors, level design, play testing? These are common tasks for entry-level programmers on a professional team where the "cool" positions (such as 3D engine and network programming) are occupied by the highly skilled programmers with proven track records who always get the job done quickly.

The best hobbies will often pay for themselves and might even earn a profit. If you have a full-time job that is otherwise fine, then you might turn the hobby of game programming into a money-making adventure. Who knows—you might release the next great indie game.

Every Situation Is Unique

There are many factors to consider in your own determination, and there is no best direction to take in life. We all just try to do the best that we can do, day by day and year by

year. I recommend that you pursue a career that will bring you the most enjoyment while still earning the highest possible salary. You might not care about salary at this point in your life; indeed, you might feel as if you would pay someone to hire you as a game programmer. I know that feeling all too well! I thought it was a strange feeling, getting paid to work on a retail game. When that game came out in stores and I saw it on the shelf, then it was an exhilarating feeling.

However, most of the world does not feel the same way that you do about video games. Very few people bother to read the credits. The feeling of exhilaration is really an internal one, not widely shared. You might already feel that this is true, given your own experience with relatives and friends who don't understand why you love games so much and why you wig out over the strangest things.

I remember the first time I discovered Will Wright's *Sim City*; it was in the late 1980s. It was quite an educational game, but extremely fun, too. Traveling with my parents, I would point out along the road, "Residential zone. Commercial zone. Ah ha! There's an industrial zone. Sure to be a source of pollution." I would note traffic jams and point out where a light rail alongside the road would ease the traffic problems. The fact is, the way you feel about video games has a strong bearing on whether you will succeed when the going gets rough, when the hours are piled on and you find yourself with no free time to actually play games anymore. All you have time to do is write code, and not even the most interesting code at that. But that spark in your eye remains, knowing that you are helping to complete this game, and it will go on your résumé as an accomplishment in life, maybe as a stepping stone in your career as a programmer.

Another argument that you might consider is the very real possibility that you could always go to college later and focus on your career now, especially if you have a lead for a job at a game company. That trend seems to be dwindling because games are now exceedingly complex projects that require highly trained and educated teams to complete them. Any self-taught programmer might have found corporate employment in the 1970s and 1980s, but the same is no longer the case with game companies. Now it has become an exceedingly competitive market. As you already know, competition causes quality to rise and costs to go down. A programmer with no college degree and little or no experience will have a very difficult time finding employment with a recognizable game company. Perhaps he can find work with one of the few hundred independent studios, but even private developers are in need of highly skilled programmers.

You might find more success by taking the indirect route to a career in game development. Many developers have gone professional after working on games in their spare time, by selling games as shareware or publishing them online. And there are as many success stories for high school graduates as there are for college graduates. As I said, every situation is unique. During this period of time, you can hone your skills, build your résumé of games (which is absolutely critical when you are applying for a job in the game industry),

develop your own game engines, and so on. Even if you are interested in game programming (which is a safe bet if you are reading this book!) just as a hobby, there is always the possibility that you will come up with something innovative, and you might be surprised to receive an unexpected job offer.

A Note about Specialization

As far as specialization goes, there is very little difference between programming a game for console or PC—all are based on the C or C++ language. These are two distinct languages, by the way. It is out of ignorance that many refer to C and C++ interchangeably, when in fact they are very different. C is a structured language invented in the 1970s, while C++ is an object-oriented language invented in the 1980s. It is a given that you must know both of these languages (not just one or the other) because that is the assumption in this industry —you simply must know them both, without exception, and you should not need a programmer's reference for most of the standard C or C++ libraries (although there are some weird functions that are seldom used). If you are a capable programmer (from a Windows, Linux, Mac, or other background), you know C and C++, and you have some experience with a game engine or library (such as Allegro), then you should be able to make your way when working on a console, such as the PlayStation 2, Xbox, or GameCube.

The software development kits for consoles typically include libraries that you must link into your program when the program is compiled and linked to an executable file. Many game companies now produce games for all of the console systems and the PC, as well as some handheld systems (such as the Game Boy Advance). Once all of the artwork, sound effects, textures, levels, and so on, have been created for a game, it is economically prudent to reuse all of those game resources for as many platforms as possible. That is why many games are released simultaneously for multiple consoles. The cost of porting a game is just a fraction of the original development cost because all of the hard coding work has been done. The game's design is already completed. Everything has been done for one platform already, so the porting team must simply adapt the existing game for a different computer system (which is essentially what a video game system is). Since all of the code is already in C or C++ (or both), the porting team must simply replace platform-specific function calls with those for the new platform.

For instance, suppose a game for the PC is being ported to Xbox—something that is done all the time. The Xbox is very similar to a Windows PC, with a Windows 2000 core and a custom version of DirectX. There is no keyboard or mouse, just a controller. Porting a PC game requires some forethought because there is a lot of input code that must be converted so the game is operated from a controller. As an example, one of the most popular online PC games of all time, *Counter-Strike*, was ported to Xbox and features online play via the Xbox Live! network.

The usual setup for a PC game includes the use of keyboard in tandem with mouse—usually the ASDW configuration (A = left, D = right, W = forward, D = backward) while using the mouse to aim and shoot a weapon. Also, you use the CTRL key to crouch and the space-bar to jump. If your mouse has a mouse-wheel, you can use that to scroll through your available weapons (although the usual way is with the < and > keys).

I have always found this to be a terribly geeky way to play a game. Yes, it is faster than a controller. But it's like we have been forced to use a data entry device for so long just to play games that we not only accept it, but we defend it. I've heard many elite *Counter-Strike* players proclaim, "I'll never switch to a controller!" The fact of the matter is, when you get used to controlling your character using dual analog sticks and dual triggers on a modern console controller (such as the Xbox Controller S, shown in Figure 1.1), it is easy to give up the old keyboard/mouse combo.

Figure 1.1 Xbox Controller S

Counter-Strike was originally a Half-Life *mod* (or rather, expansion). To play the original *Counter-Strike*, you had to already own *Half-Life*, after which *Counter-Strike* was a free (but very large) download. Porting the game to Xbox must have been a major undertaking if it was truly rewritten just for the Xbox. Based on the similarity to the now-aged PC game, I would suspect that it is the same source code, but very highly modified. There are no Xbox enhancements that I can see after having played the game for several years on the PC. It is interesting to see how the developers dealt with the loss of the keyboard/mouse input system and adapted the game to work with a controller. The in-game menus use a convenient, intelligent menu system in which you use the eight-way directional pad to purchase gear at the start of each game round.

Regardless of the differences in input control and hardware, the source code for a console or a PC game is very similar, and all of it is written in C or C++. (The biggest difference are the development environment and game libraries, or SDKs.) One common practice at a game studio is to fabricate a development system in which the SDK of each console is abstracted behind *wrapper code*, which is a term used to describe the process of wrapping an existing library of functions with your own function calls. This not only saves time, but it also makes it easier to add features and fix bugs.

Game Industry Speculation

According to Jupiter Research (http://www.jupiterresearch.com), the game industry will continue to grow, having reached an estimated $12 billion revenue during 2003. Although console sales amount to more than PC game sales, there are many more PC gamers than console gamers, and the gap will continue to widen.

I have a theory about this apparent trend. I have seen the growth of consoles over the last five years, and I am convinced that console games will be more popular than PC games in a few years. It is just a simple matter of economics. A $200 console is as capable and as powerful as a $1,500 PC. Not too long ago I was a frenetic upgrader; I always found an excuse to spend another $500 on my PC every few months.

When I stopped to look at this situation objectively, I was shocked to learn that I had been spending thousands annually—on games, essentially. Not just retail games, but the hardware needed to run those games. It seemed to be a conspiracy! The hardware manufacturers and software game companies were in league to make money. Every six months or so, new games would be released that required PC upgrades just to run. One benefit that the consoles have brought to this industry is some platform stability, which makes it far easier to develop games. Not only can you (as a game programmer) count on a stable platform, but you can push the boundaries of that platform without worrying about leaving anyone with an aging computer behind. No newly released PC game will run on a computer that is five years old (in general), but that is a common practice for the average five-year lifespan of a game console.

Given this speculation and the trends and sales figures that seem to back it up, it is very likely that the PC and console game industries—which were once mostly independent of each other—will continue to grow closer every year. That is why it is important to develop a cross-platform mindset and not limit yourself to a single platform, such as Windows. Mastery of C and C++ are the most important things, while your specific platform of choice comes second. Regardless of your proficiency with Windows and DirectX, I encourage you to learn another system. The easiest way to gain experience with console development is to learn how to program the Nintendo Game Boy Advance (GBA) because open-source tools are available for it.

Emphasizing 2D

There is a misunderstanding among many game players as well as programmers (all of whom I will simply refer to as "gamers" from this point forward) that 2D games are dead, gone, obsolete, forever replaced by 3D. I disagree with that opinion. There is still a good case for working entirely in 2D, and many popular *just-released* games run entirely under a 2D game engine that does not require a 3D accelerator at all. Also, numerous games that can only be described as cult classics have been released in recent years and will continue to be played for years to come. Want some examples?

- Sid Meier's *Civilization III* with *Play the World* and *Conquests* expansions
- *StarCraft* and the *Brood War* expansion
- *Diablo II* and the *Lord of Destruction* expansion
- *Command & Conquer: Tiberian Sun* and the *Firestorm* expansion

- *Command & Conquer: Red Alert 2* and *Yuri's Revenge*
- *Age of Empires* and the *Rise of Rome* expansion
- *Age of Empires II* and *The Conquerors*
- *Age of Mythology* and *The Titans* expansion
- *The Sims* and a dozen or so expansions and sequels
- *Real War* and the *Rogue States* expansion

What do all of these games have in common? First of all, they are all bestsellers. As you might have noticed, they all have one or more expansion packs available (which is a good sign that the game is doing well). Second, these are all 2D games. This implies that these games feature a scrolling game world with a fixed point of view and various fixed and moving objects on the screen. Fixed objects might be rocks, trees, and mountains (in an outdoor setting) or doors, walls, and furniture (in an indoor setting). With a few exceptions, these are all PC games. There are several-hundred console and handheld games that all feature 2D graphics to great effect that I could have listed. For instance, here are just a handful of exceptional games available for the Game Boy Advance:

- *Advance Wars*
- *Advance Wars 2: Black Hole Rising*
- *Super Mario World: Super Mario Advance 2*
- *Yoshi's Island: Super Mario Advance 3*
- *The Legend of Zelda: A Link to the Past*
- *Sword of Mana*
- *Final Fantasy Tactics Advance*
- *Golden Sun: The Lost Age*

What makes these games so compelling, so hot on the sales charts, and so popular among the fans? It is certainly not due to fancy 3D graphics with multi-layer textures and dynamic lighting, representative of the latest first-person shooters. What sets these 2D games apart are the fantastic gameplay and realistic graphics for the characters and objects in the game.

Finding Your Niche

What are your hobbies, interests, and sources of entertainment (aside from your PC)? Have you considered that what interests you is also of interest to thousands or millions of other people? Why not capitalize on the fan base for a particular subject and turn that into a game? Nothing beats experience. When it comes to designing a game, there is no better source on a particular subject than a diehard fan! If you are a fan of a particular sci-fi show or movie, perhaps, then turn it into your vision of a game. Not only will you have a lot of

fun, but you will create something that others will enjoy as well. I have found that when I work on a game that *I* enjoy playing and I create this game for my own enjoyment, there are people who are willing to pay for it.

Pocket Trivia Takes a Bow

Entirely for my own enjoyment and for nostalgia, I wrote a trivia game about one of my favorite sci-fi shows (*Star Trek*). The game featured 1,600 questions, 400 photos, theme-based sound effects, and a very simple multiple-choice interface (see Figure 1.2).

Figure 1.2 *Pocket Trivia* features multiple-choice trivia questions.

I decided to put the game up on my Web site and on a few game sites as a free download. Then I started to think about that decision for a moment. I had spent about two years working on that trivia game off and on during my free time, without setting any deadlines for myself. (Don't let the simplistic graphics and user interface fool you; it is very difficult to so fully cover a subject like this.) So I set a very low price on the game, just $12.00. The game sold 10 copies in the first week. That's $120.00 that I didn't have a week before, and for doing…well, nothing really, because I hadn't written that game for sale, just for fun. One month and 30 sales later, I decided to port the game to the Pocket PC platform, running Windows CE. This was about the time when my book, *Pocket PC Game Programming: Using the Windows CE Game API*, came out. (For more information about this book, see Appendix D.) I was fully immersed in Pocket PC programming, so it was not a difficult job. Oddly, I wrote the original PC game using Visual Basic 6.0, so the Pocket PC version had to be written from scratch using eMbedded Visual C++ 3.0.

Long story short, over the next year I made enough money from this little trivia game to buy myself a new laptop. That is not enough to live on, but it occurred to me that having 10 to 15 similar games in the "trialware" (try before you buy, synonymous with shareware) market, one could make a good living from game sales. The key is to continue cranking out new games every month while existing games provide your income. To do this, you need to hire out the artwork. (A professional artist will not only do far better work than the typical programmer, but he or she will do so very quickly.) I consider artwork to be at least as important as programming. Do you see how you could make a living as a game programmer by filling in niche products? You work for yourself and report to no one. If you can produce enough games to make a living, then you will be on the heels of many giants in the business.

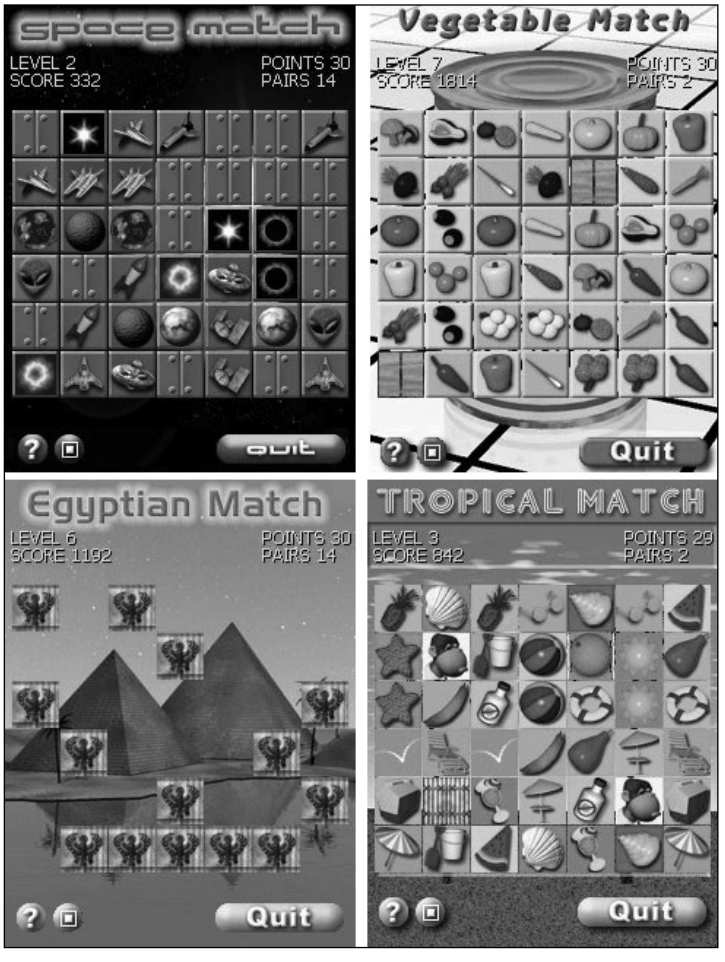


Figure 1.3 *Perfect Match* is a tile matching game with high-quality rendered graphics (four screens shown).

Perfect Match for the Fun of It

Another interesting game that I wrote is called *Perfect Match* (see Figure 1.3). It is a good example of the significant improvements you will see in the quality of your games when you collaborate with a professional artist. This game was written in about a month (again, during my spare time), and it features seven levels of play. The artwork in this game was completely modeled and rendered in 3D, and each level is a specific theme. This is another game that I personally enjoy playing, especially with such high-quality graphics (courtesy of Edgar Ibarra).

In addition to selling trialware, you can also approach a "budget" game publisher such as Xtreme Games LLC (http://www.xgames3d.com), operated by André LaMothe. Some publishers of this kind produce game compilations on CD-ROM, which have a good market at superstores, such as Wal-Mart. But the trend is heading more toward online sale and download. This is a very good way to make money by selling games that aren't "big enough" for the large retail game publishers, such as Electronic Arts. Companies such as Xtreme Games LLC make it possible for individual ("indie") developers to publish their games with little or no startup or publishing costs. Simply work on the games in your spare time and send them in when they're complete. Thereafter, you can expect to receive royalties on your games every month. Again, the amount of income depends on the quality and demand for each game.

Getting into the Spirit of Gaming

In this section I want to show you a hobby project I worked on when I was just getting started. This game is meager and the graphics are terrible, but it was a labor of love that became a learning tool when I was first learning to write code. This is an unusual approach, I realize, so I hope you will bear with me. My goal is to show you that you can turn any subject or hobby into a computer game of your own design, and no matter how good or bad it turns out to be, you will have grown significantly as a programmer from the experience.

I remember my first two-player game, which took a year to complete because there were no decent game programming books available in the late 1980s (only a handful that focused on the BASIC language), and I was literally teaching myself while working on this game. I called the game *Starship Battles*, and it was an accurate simulation of FASA's now-defunct *Tactical Starship Combat* game, right down to the individual starship specifications. This was a very popular pen-and-paper role-playing game in the 1980s, and at the time I had a collection of pewter miniature starships that I hand-painted for the game. Apparently, Paramount Pictures Corporation reined in many popular licensed products in the late 1980s, which is why this game is no longer available.

Starship Battles: An Inspired Fan Game

I wrote *Starship Battles* with Turbo Pascal 6.0 using 16-color EGA graphics mode 640×350. It featured double-buffered graphics, support for dual joysticks, and Sound Blaster effects. Figure 1.4 shows the game in action.

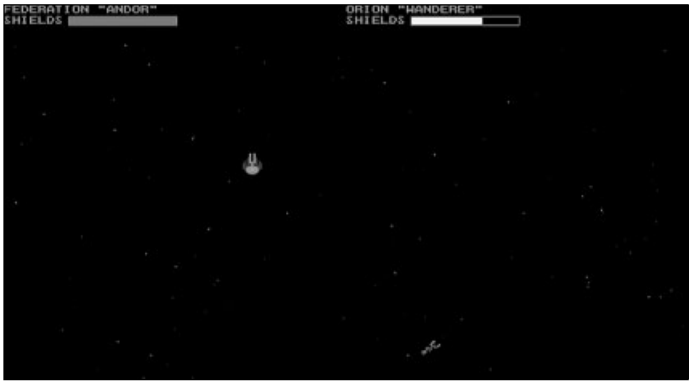


Figure 1.4 *Starship Battles* was a game of one-on-one starship combat set in the *Star Trek* universe.

Figure 1.5 shows the player selection screen in *Starship Battles*. This was a simplistic front-end for the game.

Overview of the Game

This simple-looking game took me a year to develop because I had to teach myself everything, from loading and drawing sprites to moving the computer-controlled ship to providing dual joystick support. This game also made use of the Sound Blaster Developer Kit (shown in Figure 1.6), which was very exciting at the time. I was able to produce my own sound effects (in VOC format) using the included tools and play real digital sound effects in the game. For the joystick support, I had a joystick "Y" adapter and two gamepads, requiring some assembly language programming on my part.

Figure 1.5 The player selection screen in *Starship Battles*

The game included a starship editor (shown in Figure 1.7) and my own artwork (as you probably already guessed). The original hex-based pen-and-paper game with cardboard pieces was the space battle module of FASA's larger *Star Trek: The Role Playing Game*. There were episodic add-on booklets available for this role-playing game, as well as ship recognition manuals and die-cast starship miniatures. The editor included fields for beam weapon and missile weapon types, which the game used to determine how fast a ship was able to shoot in the game, as well as how many shots could be fired at a time.

Figure 1.6 The Sound Blaster Developer Kit by Creative Labs included the libraries and drivers for multiple programming languages.

The Andor-class starship was one of my favorites because it was classified as a missile ship, able to fire eight missiles (or rather, photon torpedoes) before reloading. Some ships featured more powerful beam weapons (such as phasers or disruptor beams), which dealt great damage to the enemy ship. Figure 1.8 shows the specification sheet for the Andor, from FASA's *Federation Ship Recognition Manual* (shown in Figure 1.9). It is always interesting to see the inspiration for a particular game, even if that game is not worthy of note.

My goal is to help you to find inspiration in your own hobbies and interests.

Figure 1.7 The starship editor program made it possible to change the capabilities of each ship.

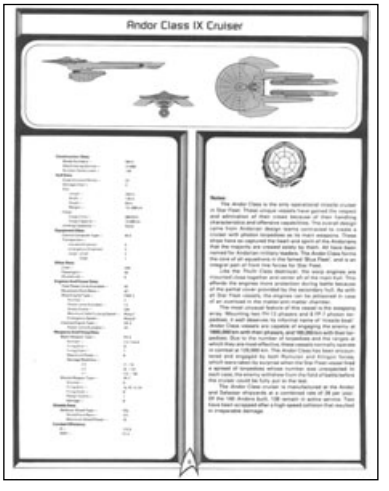


Figure 1.8 The specification sheet for the Andor-class starship

Figure 1.9 FASA's *Federation Ship Recognition Manual* provided the data entered into the starship editor, and thus affected how the game played.

Creating Game Graphics: The Hard Way

I spent a lot of time on this game and learned a lot from the experience, all of which had to be learned the hard way—through trial and error. First of all, I had no idea how to load a graphic file, such as the then-popular PCX format, so I started by writing my own graphic editor. I called this program *Sprites*; over time I wrote a 16-color version and a 256-color version. The 16-color version (shown in Figure 1.10) included limited animation support for four frames and rudimentary pixel-editing features, and was able to store multiple sprites in a data file (shown in Figure 1.11). Most importantly, I learned to program the mouse with assembly language. This sprite editor was very popular on bulletin board systems in the late 1980s.

tip

For the curious fan, the best modern implementation of FASA's tactical starship combat game is Activision's *Starfleet Command* series, excellent Windows PC games that have kept this sub-genre alive.

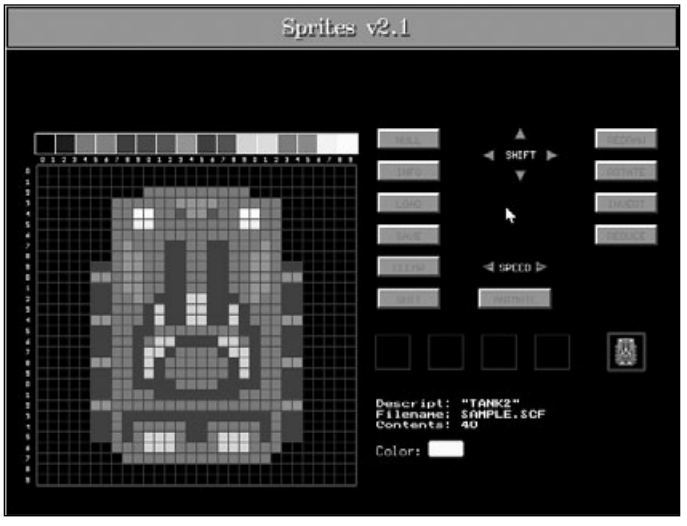


Figure 1.10 *Sprites v2.1* was a pixel-based graphic editor that I wrote in the late 1980s while working on *Starship Battles*.

Figure 1.11 The *Sprites* graphic editor could load and save multiple sprites in a single SPR file.

After completing *Starship Battles*, my plan was to convert the game to 256-color VGA mode 13h, which featured a resolution of 320×200 and support for double-buffering the screen inside the video buffer (which was very fast) for ultra-smooth, flicker-free animation. I came up with *Sprites 3.1* (shown in Figure 1.12) with an entirely new menu-driven user interface.

Rather than finish the sprite editor (which was not compatible with the previous version and lacked support for multiple sprites) and rather than focus on a new version of the game, I stopped using the program. About that time I became frustrated with limitations in Turbo Pascal and I decided to switch to Turbo C. At the same time, I switched to using Deluxe Paint instead of my sprite editor, storing game artwork in a PCX file rather than inside a custom sprite file. This being such a huge step, I never did get around to improving *Starship Battles*, which suffered from its EGA (*Enhanced Graphics Adapter*) roots. Making the transition from Pascal to C was not the most difficult part; the hardest part was rethinking my entire self-taught concept of building a game. During the development of *Starship Battles*, I had no access to a good graphic editor, such as Deluxe Paint, so I had no idea how to rotate the sprites. Instead, I wrote small utility programs to convert a single sprite into a rotated one with 16 frames.

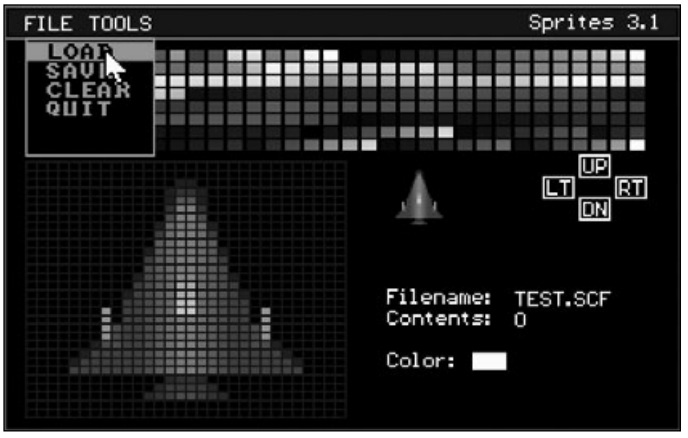


Figure 1.12 *Sprites 3.1* was a 256-color VGA mode 13h graphic editor.

Talk about doing things the hard way! I actually found some matrix math functions in a calculus book and used that knowledge to write a sprite rotation program that generated all of the rotated frames for each starship in the game.

Had I known about Deluxe Paint and Deluxe Animation, with features for drawing and animating sprites onscreen, I might have cried. At any rate, with new technology comes new power, so I gave up this game and moved on to another one of my hobbies—war games.

Axis & Allies: Hobby Wargaming

I have been a fan of Milton Bradley's *Axis & Allies* board game for 20 years, recently getting into the expanded editions, *Axis & Allies: Europe* and *Axis & Allies: Pacific*. This game is still huge, as evidenced by Web sites such as http://www.axisandallies.org. After completing *Starship Battles*, I decided to tackle the subject of *Axis & Allies* using the "proper" tools that I had discovered (namely, the C language and PCX files). The result of the effort is shown in Figure 1.13. What was truly awesome about this game was not the gameplay, per se, but the time spent with friends. (This is another factor in my belief that console games will continue to gain popularity—for all the effort, the greatest appeal of console games is taking on a friend.) What some would consider fond memories, I look back on as additional inspiration.

Figure 1.13 A solid attempt at an *Axis & Allies* computer game

What made *Axis & Allies* so much fun? Winning the game? Hardly! I rarely beat my archrival, Randy Smith (as a matter of fact, I beat him two times out of perhaps sixty games!). When you design your next game, come to terms with the fact that winning is not always the most

important thing. Having fun should be the primary focus of your games. And when your game is irresistibly fun, people will continue to play it. This is so contrary to modern game designs that focus on discrete goals; I feel that this trend coincides with the mechanical feel of the modern 3D game. Only after several years of refinement have gameplay and enjoyment started to enter the equation again. Gamers don't want a whiz-bang 3D technical demo, suitable for the crowds at GDC or E3; they want to have fun.

Overview of the Game

This single screen is packed with information that I believed would be helpful to a fan of the game. For instance, simply moving the mouse over a territory on the world map displayed the territory name, country flag, production value, attack strength, defense strength, and anti-aircraft capability. In addition, the bottom-right displayed global information about the current player, including total industrial capacity, number of territories owned, and global attack and defensive capabilities. Clicking on a territory (such as Eastern USA) would bring up a unit selection dialog, in which the player could select units to move or attack (see Figure 1.14).

Figure 1.14 The unit selection dialog was used to move units from one territory to another.

After moving units onto an enemy territory, the player would then engage in battle for that territory against the defending units. Figure 1.15 shows the battle screen.

Figure 1.15 The battle screen automatically calculated all attack and defense rolls.

Each round of a battle allowed attacker and defender a chance to fire with simulated rolls of the dice (one die for each unit, according to the board game's rules). Figure 1.16 shows the defender's counterattack.

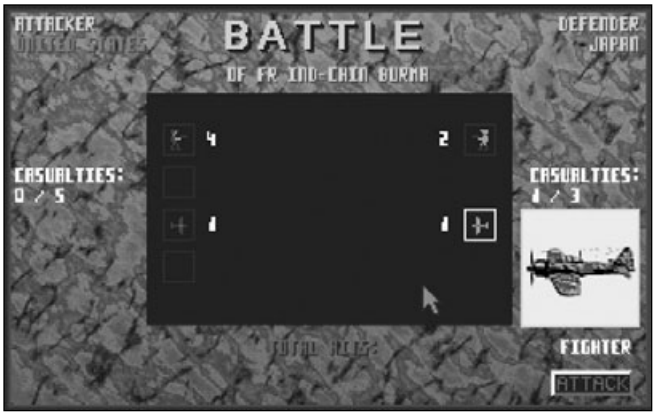


Figure 1.16 The defender makes a counterattack using remaining units.

Concluding the Game

This was the largest game I had attempted at that point, and it was difficult with the constant desire to return to Turbo Pascal, the language most familiar to me. Making strides in a new direction is difficult when it is easier to stay where you are, even if the technology is inferior. I constantly struggled with thoughts like, "It would be so much easier to use my sprite editor." But persistence paid off and I had a working game inside of a year, along with the experience of learning C and VGA mode 13h (the then-current game industry standard). This game really pushed me to learn new things and forced me to think in new ways. After much grumbling, I accepted the new technology and never looked back, although that was a difficult step. When I look back at the enormous amount of time I spent writing the most ridiculously simple (and cheesy) games, it really helps me put things into perspective today—there are wonderful software tools (many of them free) available today for writing games.

Setting Realistic Expectations for Yourself

My goal over the last few pages was not just to traverse memory lane, but to provide some personal experiences that might help explain how important motivation can be. Had I not been such a big fan of these subjects, I might have never completed the games that I have shown you here. Who cares? Touché. Whatever your opinion of reason and motivation, game development is a personal journey, not simply a skill learned solely to earn money. I will admit that these games are poor examples. They were labors of love, as I mentioned, and they suffered from my lack of programming experience. I worked with themes that I enjoyed and subjects that were my hobbies, and I can't stress enough how important that is! However, I will also point out that these game examples got me a job as a game programmer back in the day.

tip

Don't be ashamed of your work, whatever your opinion of it, because you are your own worst critic, and your work is probably better than you think. Be humble and ask the opinion of others before either praising or derailing your work.

My own personal motivations are to have fun, to delve deeper into a subject that I enjoy, to recreate an event or activity, and to learn as I go. With this motivation, I will share with you my own opinion of what makes a great game and, in later chapters, explain exactly how a game is made.

An Introduction to Dev-C++ and Allegro

I want to try to find the best balance of pushing as far into advanced topics as possible in this book while still covering the basics. It is a difficult balance that doesn't always please everyone because while some programmers need help at every step along the way, others become impatient with handholding and prefer to jump right into it and start. One of the problems with game development for the hobbyist today is the sheer volume of information on this subject, in both printed and online formats. It is very difficult to get started learning how to write games, even if your goal is just to have some fun or maybe write a game for your friends (or your own kids, if you have any). I find myself lost in the sheer magnitude of information on the overall subject of game development. It truly is staggering just looking into personal compilers, libraries, and tools, let alone the commercial stuff. If you have ever been to the Game Developers Conference in San Jose, California, then you'll know what I mean. This is a huge industry, and it is very intimidating! Getting started can be difficult. But not only that, even if you have been a programmer for many years (whether you have worked on games or not), just the level and amount of information can be overwhelming.

DirectX Is Just Another Game Library

One subject that is rather universal is DirectX. I have found that the more I talk about DirectX, the less I enjoy the subject because it is basically a building block and a tool, not an end in and of itself. Unfortunately, DirectX has been misunderstood, and many talk about DirectX as if *it* is game programming. If you learn the DirectX API, then you are a game programmer. Why doesn't that make sense to me? If I can drive a car, then am I suddenly qualified to be a NASCAR driver? DirectX is just a tool; it is not the end-all and be-all of game development.

In fact, there are a lot of folks who don't even like DirectX and prefer to stick with cross-platform or open-source tools, in which development is not dictated by a company with a stake in the game industry (as is the case with Microsoft and the Xbox console, in addition to Microsoft Game Studios). The professionals use a lot of their own custom libraries, game engines, and tools, but an equal number use off-the-shelf game development tools such as RenderWare Studio (http://www.renderware.com). This is a very powerful system for game development teams working on multi-platform games. What this means is that a single set of source code is written and then compiled for PC, Xbox, PS2, and GameCube (with support for any new consoles that come out in the future through add-on libraries).

Have you seen any games come out recently for multiple platforms at the same time? (One example is LucasArts' *Secret Weapons Over Normandy.*) It is a sure bet that such games were developed with RenderWare or a similar cross-platform tool. RenderWare includes source code management and logistical control in addition to powerful game libraries that handle advanced 3D graphics, artificial intelligence, a powerful physics system, and other features. And this is but one of the professional tools available!

I have found that there are so many books on DirectX now that the subject really doesn't need to be tackled in every new game development book. My reasoning is logical, I think. I figure that no single volume should try to be the sole source of information on any subject, no matter how specific it is. Should every game development book also teach the underlying programming language to the reader? We must make some assumptions at some point, or else we'll end up back at square one, talking about ones and zeroes!

You should consider another very important factor while we're on the subject of content. Windows is not the only operating system in the world. It is the most common and the most dominant in the industry, but it is not the only choice or even necessarily the best choice for every person (or every computer). Why am I making a big deal about this? I use Windows most of the time, but I realize that millions of people use other operating systems, such as Linux, UNIX, BeOS, FreeBSD, Mac OS, and so on—whatever suits their needs. Why limit my discussion of game development only to Windows users and leave out all of those eager programmers who have chosen another system?

The computer industry as we know it today was founded on powerful operating systems such as UNIX, which is still a thriving and viable operating system. UNIX, Linux, and the others are not more difficult to use, necessarily; they are just different, so they require a learning curve. The vast majority of consumers use Windows, and thus most programmers got started on Windows.

Introducing the Allegro Game Library

I want to support systems other than Windows. Therefore, this book focuses on the C language and the Allegro multi-platform game development library (which does use DirectX on the Windows platform, while supporting many others). Allegro was originally developed by Shawn Hargreaves for the Atari ST; as a result of open-source contributions, it has evolved over time to its present state as a powerful game library with many advanced 2D and 3D features also included. The primary support Web site for Allegro is at http://www.talula.demon.co.uk/allegro. I highly recommend that you visit the site to get involved in the online Allegro community because Allegro is the focus of this book.

Rather than targeting Xbox, PS2, and GameCube (which would be folly anyway because the console manufacturers will not grant licenses to unofficial developers), Allegro targets multiple operating systems for just about any computer system, including those in Table 1.1.

Table 1.1 Allegro and Operating Systems

Operating System	Compiler/Tools
Mac OS X	Apple Developer Tools 2002
Windows	Microsoft Visual C++ 4.0 (or later)
Windows	Borland C++ 5.5, C++Builder 1.0 (or later)
Windows	MinGW32/Cygwin
MS-DOS	DJGPP 2.01 with GCC 2.91 (or later)
MS-DOS	Watcom C++ 10.6 (or later)
IRIX	GCC 2.91 (or later)
Linux	GCC 2.91 (or later)
Darwin	GCC 2.91 (or later)
FreeBSD	GCC 2.91 (or later)
BeOS	Be Development Tools
QNX	QNX Development Tools

Table 1.1 presents an impressive and diverse list of operating systems, wouldn't you agree? Allegro abstracts the operating system from the source code to your game so the source code will compile on any of the supported platforms. This is very similar to the way in which OpenGL works. (OpenGL is another open-source game development library that focuses primarily on 3D.)

Allegro itself is not a compiler or language; rather, it is a game library that must be linked to your main C or C++ program. Not only is this practice common, it is smart. Any time you can reuse some existing source code, do so! It is foolish to reinvent the wheel when it comes to software, and yet that is exactly what many programmers do. I suspect many programmers prefer to rewrite everything out of a sense of pride or arrogance—as in, "I can do better." Let me tell you, game development is so extraordinarily complicated that if you try to write all the code yourself without the benefit of a game library or some help from the outside world, you will quite literally never get anywhere and your hard work will never be appreciated!

Allegro's 2D and 3D Graphics Features

Allegro features a comprehensive set of 2D and 3D graphics features.

Raster operations	Pixels, lines, rectangles, circles, Bezier splines
Filling	Pattern and flood fill
2D sprites	Masks, run-length encoding, compiled sprites, translucency, lighting

Bitmaps	Blitting, rotation, scaling, clipping
3D polygons	Wireframe, flat-shaded, gouraud-shaded, texture-mapped, z-buffered
Scrolling	Double- or triple-buffers, hardware scrolling (if available)
Animation	FLI/FLC playback
Windows drivers	DirectX windowed and full-screen, GDI device contexts
DOS drivers	VGA, Mode-X, SVGA, VBE/AF, FreeBE/AF
UNIX drivers	X, DGA, fbcon, SVGAlib, VBE/AF, Mode-X, VGA
BeOS drivers	BWindowScreen (full-screen), BDirectWindow (windowed)
Mac OS X	CGDirectDisplay (full-screen), QuickDraw/Cocoa (windowed)

Allegro's Sound Support Features

Allegro features some excellent support for music playback and sound effects.

Wavetable MIDI	Note on, note off, volume, pan, pitch, bend, drum mappings
Digital sound	64 channels, forward, reverse, volume, pan, pitch
Windows drivers	WaveOut, DirectSound, Windows Sound System
DOS drivers	Adlib, SB, SB Pro, SB16, AWE32, MPU-401, ESS AudioDrive, Ensoniq
UNIX drivers	OSS, ESD, ALSA
BeOS drivers	BSoundPlayer, BMidiSynth
Mac OS X drivers	CoreAudio, Carbon Sound Manager, QuickTime Note Allocator

Additional Allegro Features

Allegro also supports the following hardware and miscellaneous features.

Device input	Mouse, keyboard, joystick
Timers	High-resolution timers, interrupts, vertical retrace
Compression	Read/write LZSS compressed files
Data files	Multi-object data files for storing all game resources
Math functions	Fixed-point arithmetic, trigonometric lookup tables
3D functions	Vector, matrix, quaternion manipulation
Text output	Proportional fonts, UTF-8, UTF-16, Unicode

Supporting Multiple C/C++ Compilers

Not only is this book focusing on a free open-source game library in the form of Allegro, I will also use an open-source C/C++ compiler and IDE (*Integrated Development Environment*) called Dev-C++, which is shown in Figure 1.17.

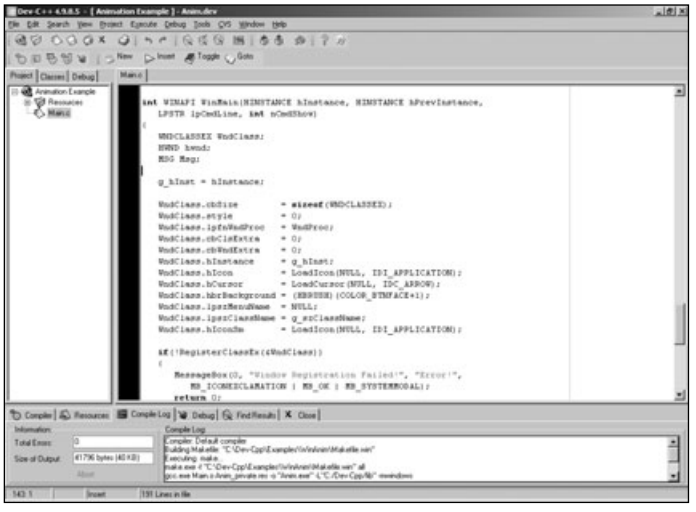


Figure 1.17 Dev-C++ is the open-source C/C++ compiler and IDE used in this book.

Dev-C++ includes an open-source C++ compiler called GCC (*GNU Compiler Collection*) that is the most widely used C++ compiler in the world. I used this compiler to develop the sample programs for my Game Boy Advance book, too! GCC is an excellent and efficient compiler for multiple platforms. In fact, many of the world's operating systems are compiled with GCC, including Linux. It is a sure bet that satellites in orbit around Earth have programs running on their small computers that were compiled with GCC. This is not some small niche compiler—it is a global phenomenon, so you are not limiting yourself in any way by using GCC. Most of the console games that you enjoy are compiled with GCC. In contrast, the most common Windows compilers, such as Microsoft Visual C++ and Borland C++Builder, aren't used as widely but are more popular with consumers and businesses.

This brings up yet another important point. The source code in this book will compile on almost any C/C++ compiler, including Visual C++, C++Builder, Borland C++, Watcom C++, GCC, CodeWarrior, and so on. Regardless of your compiler and IDE of choice, the code in this book should work fine, although you might have to create your own project files for your favorite compiler. I am formally supporting Dev-C++, Visual C++, and KDevelop (under Linux), so you will find the source code for these compilers on the CD-ROM. All that means is that I have created the project files for you. The source code is all the same! Incidentally, Dev-C++ is also included on the CD-ROM. Due to its very small size (around 12 MB for the installer), you might find it easier to use than Visual C++ or C++Builder, which have very large installations. Dev-C++ is capable of compiling native Windows programs and supports a diverse collection of DevPaks—open-source libraries packaged in an easy-to-use file that Dev-C++ knows how to install.

Allegro is one such example of an existing code library, and it's just plain smart to use it rather than starting from scratch (as in learning to program Windows and DirectX). But what if you are really looking for a DirectX reference? Well, I can suggest several dozen good books on the subject that provide excellent DirectX references (see Appendix D, "Recommended Books and Web Sites"). The focus of this book is on practical game programming, not on providing a primer for Windows or DirectX programming (which is quite platform-specific in any event). As I have mentioned and will continue to do, I am a big fan of Windows and DirectX. However, I am also a big fan of console video game systems, and programming a console will open your eyes to what's possible. This is especially true if you have limited yourself to writing Windows programs and you have not experienced the development possibilities on any other system.

Dev-C++ is just one of the IDE/compiler tools you can use to compile the code in this book. Feel free to use any of the compilers listed back in Table 1.1. It might be possible to use older compilers (such as Turbo C++ or an early version of Microsoft C++) for MS-DOS, but I wouldn't recommend it. Who is still using MS-DOS today? I only mention it because Allegro does support MS-DOS and the DJGPP compiler. While GCC is guaranteed to work with Allegro, the same cannot be said for obsolete compilers, which very likely do not support modern library file structures. If you insist on using MS-DOS, then by all means make use of DJGPP because it is based on GCC.

Summary

This chapter presented an overview of game development and explained the reasoning behind the use of open-source tools such as Dev-C++ and Allegro (the primary benefit being that these tools are free, although that does not imply that they are inferior in any way). I explained how Windows and DirectX are the focus of so much that has already been written, and that this book will delve right into game programming rather than spending time on logistical things (such as tools). I hope you will embrace the way of thinking highlighted in this chapter and broaden your horizons by recognizing the potential for programming systems other than Windows. By reading this book and learning to write platform-independent code, you will be a far more flexible and versatile programmer. If you don't fully understand these concepts quite yet, the next chapter should help because you will have an opportunity to see the capabilities of Dev-C++ and Allegro by writing several complete programs.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What programming language is used in this book?
   A. C
   B. Pascal
   C. C++
   D. Assembly

2. What is the name of the free multi-platform game library used in this book?
   A. Treble
   B. Staccato
   C. Allegro
   D. FreeBSD

3. What compiler can you use to compile the programs in this book?
   A. Dev-C++
   B. Borland C++Builder
   C. Microsoft Visual C++
   D. All of the above

4. Which operating system does Allegro support?
   A. Windows
   B. Linux
   C. Mac OS X
   D. All of the above

5. Which of the following is a popular strategy game for the PC?
   A. *Counter-Strike*
   B. *Splinter Cell*
   C. *Real War*
   D. *Advance Wars*

6. What is the most important factor to consider when working on a game?
   A. Graphics
   B. Sound effects
   C. Gameplay
   D. Level design

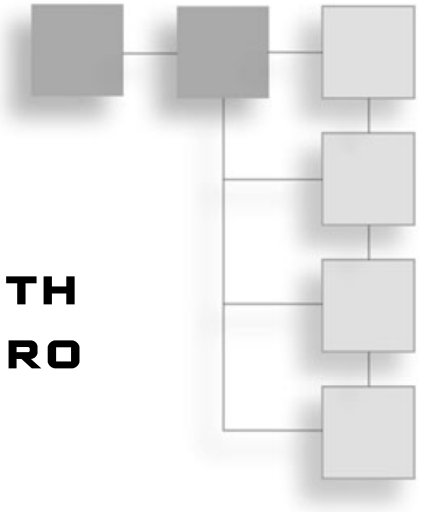
7. What is the name of the free open-source IDE/compiler included on the CD-ROM?
   A. Visual C++
   B. Dev-C++
   C. Watcom C++
   D. C++Builder

8. What is the name of the most popular game development library in the world?
   A. OpenGL
   B. DJGPP
   C. DirectX
   D. Allegro

9. Which of the following books discusses the gaming culture of the late 1980s and early 1990s with strong emphasis on the exploits of id Software?
   A. *Masters of Doom*
   B. *The Age of Spiritual Machines*
   C. *The Inmates Are Running the Asylum*
   D. *Silicon Snake Oil*

10. According to the author, which of the following is one of the best games made in the 1980s?
    A. *Civilization III*
    B. *Counter-Strike*
    C. *King's Quest IV: The Perils of Rosella*
    D. *Starflight*

CHAPTER 2



Getting Started with Dev-C++ and Allegro

This chapter introduces the Dev-C++ integrated development environment, the GNU C++ compiler, and associated tools. You will learn how to install and configure Dev-C++ for game development with the Allegro game library. Because these programs are all available on the book's CD-ROM, everything you need to start writing cutting-edge games was included in the price of this book! However, if you prefer to use a different compiler, such as Microsoft Visual C++ (any version from 4.0 on will work), please refer to Appendix E, "Configuring Allegro for Microsoft Visual C++ and Other Compilers."

There was a time when installing Allegro involved more than just running a setup utility; you had to compile Allegro before using it. The extremely talented contributors to Dev-C++ and Allegro have made things so much easier with the latest versions of these tools. Dev-C++ now includes an update tool that will install the latest version of Allegro automatically, right off the Web! Although this chapter focuses on setting up Dev-C++ and Allegro for Windows, I have added an appendix that will explain how to compile Allegro for systems that might not support the update tool. Please refer to Appendix F, "Compiling the Allegro Source Code" for details.

Everything you need to know to get up and running with Dev-C++ and Allegro is fully explained in the following pages. Unlike some programming books that try to offer standalone chapters as a series of independent tutorials, the chapters in this book should be read sequentially because each chapter builds on the one before it. This chapter in particular is critical in that respect because it explains how to set up the development tools used in the rest of the book.

Here is a breakdown of the major topics in this chapter:

- Installing and configuring Dev-C++ and Allegro
- Taking Dev-C++ and Allegro for a spin
- Gaining more experience with Allegro

Introduction

Allow me to go off topic for a moment. I love role-playing games. I am especially fond of the old-school 2D RPGs that focus on strong character development, exploration, and questing as a solo adventurer. I am still amazed at the attention to detail in games such as *Ultima VII: The Black Gate*, which is now more than 10 years old. This game was absolutely amazing, and its legacy lives on today in the form of *Ultima Online*. The music in this game was so ominous that it actually affected most players on an emotional level, drawing them into the game with a desire to help the Avatar save Britannia. The open storyline and freedom to explore the world made it so engaging and engrossing that it completely suspended my sense of disbelief—that is, while playing, I tended to forget it was merely a game.

Contrast that experience with modern games that are more focused on eye candy than exploring the imagination! It reminds me of the difference between a movie and a book; each has a certain appeal, but a book delights at a more personal level, opening the mind to new possibilities. I am drawn into a good game, such as *Ultima VII*, just as I am with a good book; on the other hand, even an all-time favorite movie usually fails to draw me into the story at a personal level. I am experiencing the imagination and vision of another person, and those impressions are completely different than my own. Teasing the imagination is what separates brilliance from idle entertainment, and it is the difference between a long remembered and beloved memory (found in a good book or a deeply engaging game) and a quickly forgotten one (such as in a typical movie).

It is a rare game that is able to enchant one's imagination while also providing eye candy. One such game is *Baldur's Gate: Dark Alliance* (a console implementation of the best-selling PC game). This game is intelligent, challenging, imaginative, enjoyable, engaging, and still manages to impress visually as well as audibly. The layout of this game is an overhead view, although it is rendered in 3D, giving it a 2D feel that resembled the orientation of *Ultima VII* and *Diablo II*. That someone is still building fantastic RPGs like this is a testament to the power of a good story and the joy of character development and leveling up. The pizzazz of highly detailed 3D graphics simply satisfies the picky gamers.

As you delve further into this chapter, try to keep in mind what your ideal game would be. What is your all-time favorite game? What genre does it represent? How would you improve upon the game, given the opportunity? I will continually encourage you to keep

your ideal game design in mind while working through this book. I hope you will start to develop that game as you progress through each chapter. To that end and to form a basis for building your own game, I will walk you through the creation of a complete game—not just a sample or demonstration program, but a complete, full-featured game with all the bells and whistles! Although I would really enjoy building an RPG, that is far too ambitious for the goals of this book. RPGs are so enormous that even the simplest of RPGs is a huge undertaking, and there are so many prerequisites just to get started. For instance, will the hero be able to wield different weapons? Animating a single character can require more than 100 animation frames for a single sprite—and that is just with one weapon, one set of armor. What if you want your character to be equipped with different kinds of weapons and armor in the game (in my opinion, one of the best aspects of an RPG)? You could design the game with a fixed character image, but you are still looking at a huge investment in artwork.

My second choice is a strategy game, so that is the approach I have taken in this book. Strategy games are enormously entertaining while requiring a meager initial investment in artwork. In fact, in the spirit of the open-source tools used in this book, I will also be using a public domain sprite library called SpriteLib. This library was produced by Ari Feldman, a talented artist who was kind enough to allow me to use his fantastic high-quality artwork in this book. As you will see in the next two chapters, each great game idea starts with a basic prototype, so you will develop the first prototype version of this strategy game in Chapter 4, "Writing Your First Allegro Game." Following that, each major chapter will include a short section on enhancing the game with the new information presented in each chapter. For instance, the first version of the strategy game will have a fixed background, but when I cover scrolling backgrounds I'll show you how to enhance the game to use that new feature. The same goes for animated sprites, sound effects, music, special effects, and so on.

Installing and Configuring Dev-C++ and Allegro

I know you are looking forward to jumping into some great source code and working on some real games. I feel the same way! But before you can do that, I have to explain how to configure the development tools used in this book. Regardless of whether you are a newcomer to programming or a seasoned expert looking for an entertaining diversion, you will find the information in this chapter valuable because it is important to get set up properly before you delve into the advanced programming chapters to come! I think you will come to enjoy using Dev-C++ regardless of your experience level. However, if you don't like the editor and IDE for any reason, you can configure your favorite IDE to use Allegro; see Appendix E. This appendix covers several compilers, such as Visual C++, Borland C++, and KDevelop.

Dev-C++ is an open-source integrated development environment (IDE) for the infamous GCC (*GNU Compiler Collection*), a multi-platform C/C++ compiler. Dev-C++ and GCC are both distributed under the GNU General Public License, which means they are freely redistributable as long as the source code is provided for the tools themselves and any derivative works. In case you were wondering, GNU stands for "GNU is Not Unix." This is something of an inside joke in the open-source community, in that the name is recursive.

note

The GNU General Public License is printed in the back of the book.

Dev-C++ was developed by Bloodshed Software (http://www.bloodshed.net), and the primary Web site for Dev-C++ is located at http://www.bloodshed.net/devcpp.html. The version of Dev-C++ included on the book's CD-ROM includes an updating tool that will download updates to the compiler or tools, although I still recommend visiting Bloodshed Software's site to get up-to-date news and information.

Although I am not going to cover it in this book, Bloodshed Software also has a very interesting product called Dev-Pascal that uses the same IDE as Dev-C++ but features syntax highlighting for the Pascal language (including support for Delphi) and makes use of the GNU Pascal compiler. I sure would have enjoyed this product back in the day, when I was a Turbo Pascal fan!

Installing Dev-C++

The installation process for Dev-C++ is so simple that I'm not even going to go over it here. If you have any problems installing it, refer to the Bloodshed Software Web site. Simply run the executable file containing the Dev-C++ files; the version included on the CD-ROM is called devcpp4980.exe. I do want to make a recommendation on the install location: I recommend installing Dev-C++ on the same hard drive as Allegro (and your game projects). It just makes things easier when everything is readily available, especially when you consider that browsing for files on multiple drives can be a nuisance. I have several drives in my PC and I choose to install game development software on one of the partitions that I have set aside exclusively for that purpose. I also recommend installing things on the root (such as C:\Dev-Cpp). The installer for Dev-C++ is provided on the CD-ROM in the \dev-cpp folder; it is called devcpp4980.exe. Feel free to use your favorite IDE and compiler as long as it's capable of compiling standard C/C++ code for Windows, Linux, Mac OS X, or one of the other supported systems. If you are living in Antarctica and are stuck with an old PC running MS-DOS, then you can use DJGPP or Watcom.

When is the last time you came across a retail game box in a store that listed MS-DOS, Windows, Linux, and Mac OS X support? Yep, Allegro (the library that makes this possible) is awesome.

note

Dev-C++ was created by Bloodshed Software using Borland's Delphi compiler.

Updating Dev-C++

The easiest way to update Dev-C++ is to use the built-in update tool. I have also provided the latest update (at the time of this writing) of 4.9.8.5; it is located in \dev-cpp and it is called devcpp4985.zip. You can simply unzip this file inside your C:\Dev-Cpp folder to perform a manual update. I highly recommend this simple manual update because it supercedes the process involving the old update tool, providing you with WebUpdate right from the start. If you prefer to use the update tool, I'll explain how it works. Once you have installed Dev-C++, open the Tools menu and select Check for Updates/Packages (see Figure 2.1). This will open the update program for Dev-C++.

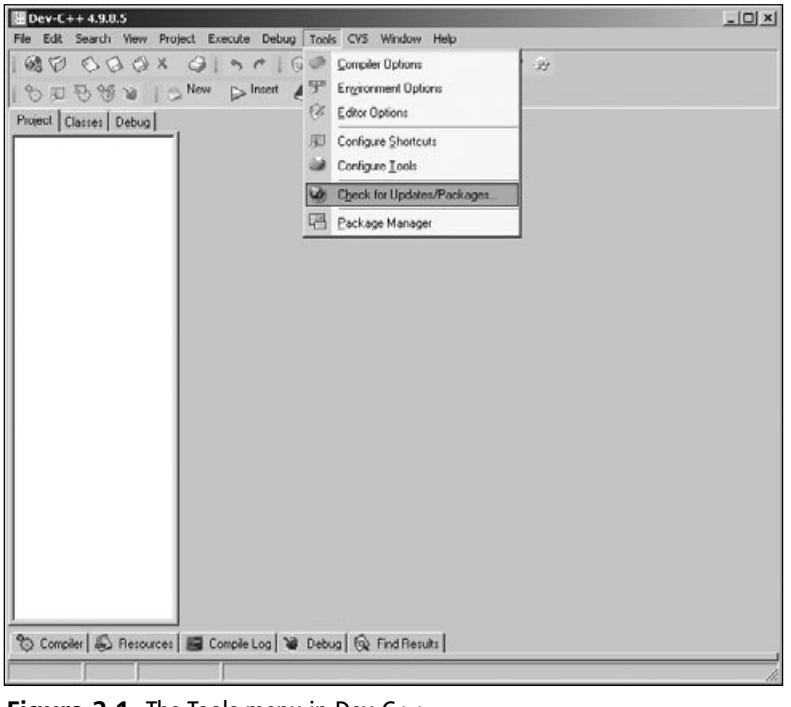


Figure 2.1 The Tools menu in Dev-C++

Dev-C++ includes an update tool to automatically download and install updates. The update program connects to an online server to download packages for Dev-C++, and it is wonderfully easy to use. (In contrast, how often have you ever updated Visual C++ or

Borland C++Builder over the Net? The typical Microsoft service pack is hundreds of megabytes in size.) The update program is shown in Figure 2.2. This default updater has been replaced with a more useful WebUpdate program, but first you must get an update to Dev-C++ to take advantage of this great new updater. Referring to Figure 2.2 again, you will see four buttons on the right. Click on the top button (the one with the purple check-mark on it). This will bring up a dialog box asking you to shut down Dev-C++, which you should do; then click on the Retry button to continue.

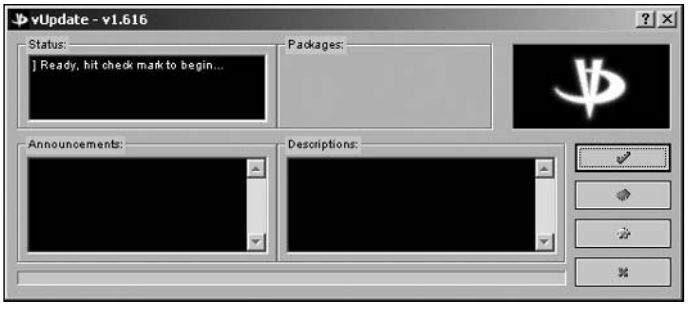


Figure 2.2 Dev-C++ comes with an update tool that will download updates and packages.

The list of updates might change by the time you read this, but at this time there are two updates available (see Figure 2.3). Check both files and click on the Start button to proceed with the download.

Figure 2.3 The update program displays the available downloads for updating Dev-C++.

tip

It is very likely that Dev-C++ will be updated at regular intervals, at which point the screenshots and tutorials in this section might not apply. If you are using at least version 4.9.8.5 of Dev-C++, that is all you really need while working through this book. The advantage of updating is that you receive bug fixes and new features. For instance, the 8.5 revision includes the newer WebUpdate tool.

After you have completed this initial update process and installed the new version of Dev-C++ (which is done automatically by the update program), your copy of Dev-C++ will be ready to use the more advanced WebUpdate feature. If the Package Manager opens at this point, you can just close it. Once again, start Dev-C++, and you will notice that the version displayed in the caption bar is Dev-C++ 4.9.8.5. Open the Tools menu and select Check for Updates/Packages again. Now you should see the WebUpdate tool, as shown in Figure 2.4.

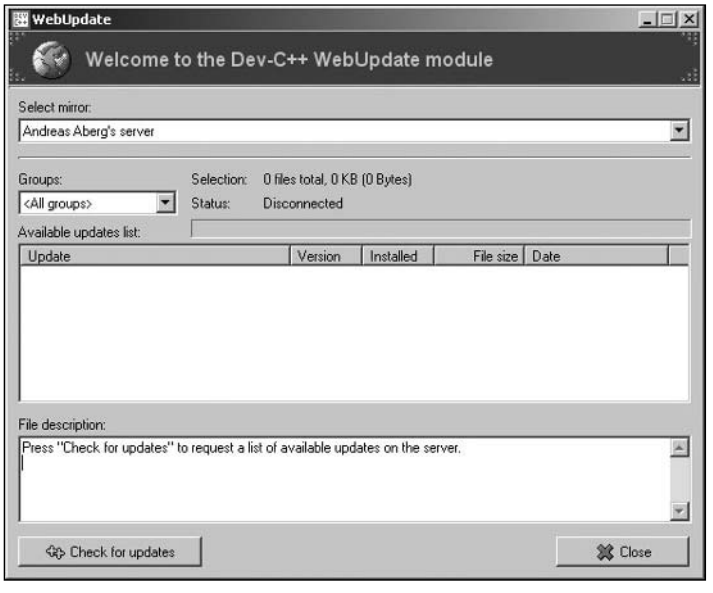


Figure 2.4 Dev-C++ is now equipped with the WebUpdate tool, making it very easy to install new DevPaks.

At the bottom-left corner is a button called Check for Updates. Click on this button to retrieve a list of updates for Dev-C++ (see Figures 2.4 and 2.5).

note

Several Dev-C++ DevPaks have been included on the CD-ROM that accompanies this book so you can install the critical packages you need to compile the source code in this book. The most important DevPak is Allegro! To install a DevPak from Dev-C++, go to Tools, Package Manager and click on the Install icon to browse for a DevPak file (such as Allegro.Devpak). I recommend compiling and installing Allegro yourself; see Appendixes E and F.

I recommend that you not download all of the DevPaks right away, although the temptation is great. Although you can browse the installed DevPaks using the Package Manager, it makes more sense to download only what you need. This not only saves bandwidth for others trying to download files from the update site, but it also gives you time to learn

about the packages one at a time. And there are many—just take a look at the list! You will find everything from a MySQL database library to a CD audio extraction library to a DirectX 9 package for Dev-C++. In addition, you will see a Windows API reference, a GNU C library reference, and many more.

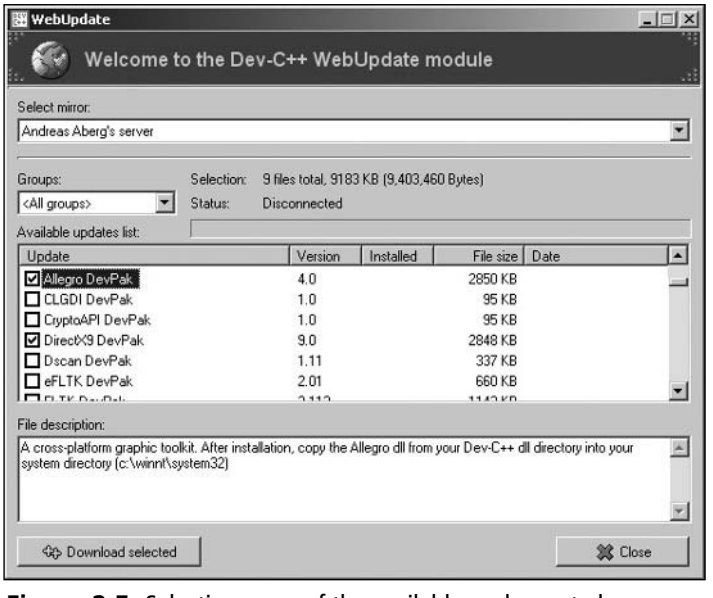


Figure 2.5 Selecting some of the available packages to be installed by WebUpdate

Definitely grab all of the Dev-C++ update packages (the first five or six files), such as Dev-C++ Update, PackMan, and so on. You also must get the Allegro package to run the programs in this book. While you are at it, also select whatever DirectX library is available. (The current package as of this writing is DirectX 9, but you really only need DirectX 8.) Most importantly, I recommend installing the packages one at a time! The Package Manager opens every time an update is installed, so it is much easier to get the updates one at a time.

tip

The DevPaks available on the update site for Dev-C++ are not always up to date because they are maintained by volunteer contributions. If you want to install only the minimum tools needed for this book, they are all provided on the CD-ROM. In particular, the Dev-C++ packages are located in the \DevPaks folder. It might be a good idea to ensure consistency by simply installing everything off the book's CD-ROM rather than relying on WebUpdate.

Installing Allegro

If you followed the steps in the previous section, you should now have Allegro installed from the WebUpdate tool in Dev-C++. If you do not have online access, you can install the Allegro.Devpak off the CD-ROM that accompanies this book (look in \DevPaks). You do not need any of the other DevPaks to compile the code in this book, but you absolutely must have Allegro installed. As was the case with Dev-C++, the version of Allegro provided by default is out of date and must be updated. It is entirely possible to install Allegro manually (see Appendix F). But one benefit that comes with the Allegro package is the convenience of the project templates added to Dev-C++. So what you should do is install the Allegro.DevPak and then copy the Allegro update files.

I have already compiled Allegro 4.0.3 (using the processes covered in Appendix F) and placed the updated library files on the CD-ROM in \allegro. If you are lost at this point, don't worry—I'm just providing an overview of what will be explained in more detail over the next few pages. Installing Allegro is just as easy as installing Dev-C++, but there is an added level of complexity because this software is never set in stone. As is the case with almost every open-source program, Dev-C++ and Allegro both undergo changes frequently to improve functionality and correct bugs. Understanding this and the fact that no single corporation is responsible for the software is a big step toward understanding how open-source works. Indeed, the major difference between commercial and open-source software is the matter of support. The high cost of commercial software pays for not just the development costs, but also the support costs (for those users who need to call the technical support line for assistance). Open-source software basically has no formal support at all, although there are hundreds of other users on the Web who are willing to help.

Figure 2.6 Allegro is an open-source, multiplatform game programming library.

Just as an aside, there are several logos available for Allegro, including the one shown in Figure 2.6. Is it ironic that even the logos for this software were donated?

The Package Manager comes up to install each DevPak after download. Make sure that you have correctly downloaded and installed Allegro by looking at the packages listed in the Package Manager (see Figure 2.7).

tip

If you want to double-check the installation of a particular library (such as Allegro), you can browse to the Dev-Cpp folder and look inside lib. The Allegro library's main file is called liballeg.a; this is specified in the linker options with -lalleg (note the "lib" and ".a" parts of the filename are assumed).

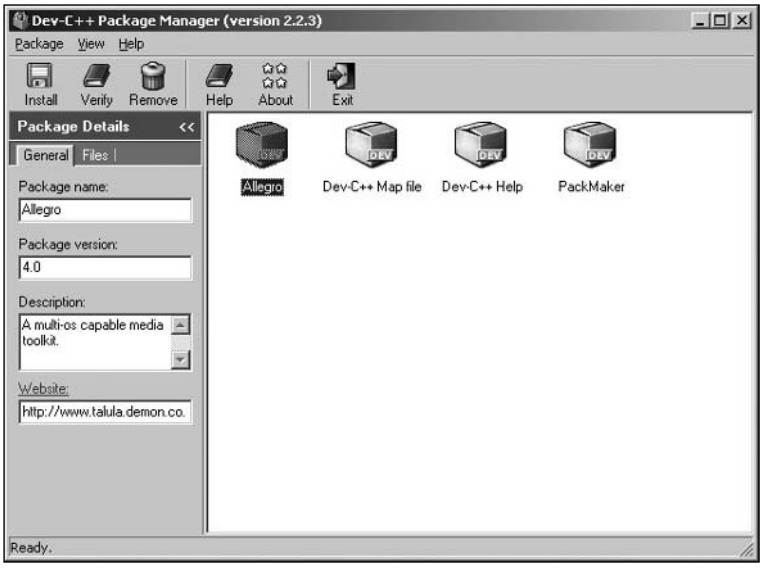


Figure 2.7 The Package Manager displays the packages that have been installed for Dev-C++.

I encourage you to read Appendix F to learn how to compile Allegro for yourself.

The Allegro DevPak and Source Code

There are two versions of Allegro that you can use—the prepackaged version or the source code version. The prepackaged version (Allegro.DevPak) for Windows includes a DLL that you must distribute with any program you compile. This isn't a big deal because you can simply install this DLL with any game or other program that you produce and distribute. The DLL is also useful if you have a Windows compiler that has a hard time compiling the Allegro library and is otherwise not compatible with the Allegro LIB files produced by GCC. Most Windows compilers produce code that is compatible with a standard DLL (not the ActiveX/COM variety, just a standard library). On most systems other than Windows, you will want to use the static library. However, the DevPak also includes a static library that you can have linked right into your programs, nullifying the need for the DLL. I will primarily use the dynamic version of Allegro for the sample projects in the book, but I will lean more toward the static library in later chapters. It's a little more difficult to configure a static project; it is extremely simple to create a new Allegro project using the dynamic library.

The second option is to compile the Allegro source code yourself, creating both the dynamic and static libraries. Rather than get sidetracked setting up GCC to compile the Allegro source code at this point, I refer you to Appendix F for detailed instructions on how to compile the Allegro source code with GCC. This would also be advisable if you are running an operating system other than Windows because the appendix explains how to

compile Allegro under both Windows and Linux (which should be enough to get you going with any other OS). To make things simple while you're just getting started, I will use the Allegro (DLL) version and the projects provided by the DevPak.

note

Are you confused yet? I realize this is a lot of information to absorb all at once, but it is basically how I present information. I prefer to provide an overview of any process (or program, for that matter) to give a bird's-eye view, and then go over that subject in detail. I believe it helps to understand the big picture when you are learning something new.

Allegro's Versatility

Allegro is useful for more than just games. It is a full-featured multimedia library as well, and it can be used to create any type of graphical program. I can imagine dozens of uses for Allegro outside the realm of games (such as graphing mathematical functions). You could also use Dev-C++ and Allegro to port classic games (for which the source code is available) to other computer systems. I have had a lot of fun porting old graphics programs and games to Allegro because it is so easy to use and yet so powerful at run time.

For instance, Relic Entertainment released the source code to *Homeworld* in September, 2003, to great acclaim in the game development community. You can download the *Homeworld* source code by going to http://www.relic.com/rdn. You will need to sign up for an account with the Relic Developer's Network (which is free) to download the source code, an 18-MB zip file. Although *Homeworld* was written for DirectX and OpenGL, it could be adapted to Allegro with a little effort—if you are interested in a challenge, that is! The source code for many other commercial games has been released in the last few years, such as the code for *Quake III*. John Carmack from id Software seems to have started this trend by originally releasing the *Doom* source code a few years after the game's release, and following that with the code for most of id's games through the years. Why? Because he shares the opinion of many in the game industry that software should not be patented, that education and lifelong learning should be encouraged. Carmack is also a cross-platform developer.

Taking Dev-C++ and Allegro for a Spin

It's time to start writing some actual programs with Dev-C++ and Allegro. In this section I will walk you through several short programs. In the process, you will learn how to create a new C project and write the initialization code for Allegro before calling on the Allegro-specific functions. First you need to make sure that Allegro was installed properly, so you'll start by writing a short program to verify that Allegro is available for use. Then I'll go over some more interesting programs with you.

Testing Dev-C++: The Greetings Program

The first step in testing the installation is to write a short program in Dev-C++ to verify that GCC is working as expected because Dev-C++ is just the IDE/editor, and it calls gcc.exe to compile programs. Start Dev-C++. In Windows, it is located in the Start menu under Programs, Bloodshed Dev-C++. Because this is a small, tight IDE, it comes up immediately and presents you with a blank project workspace, as shown in Figure 2.8.

note

For the sake of brevity, I will often refer to both the compiler and IDE collectively as "the compiler." This applies to Dev-C++, Visual C++, or any other compiler system where the IDE actually runs the command-line compiler and presents the programmer with the results returned by the compiler (such as error messages).

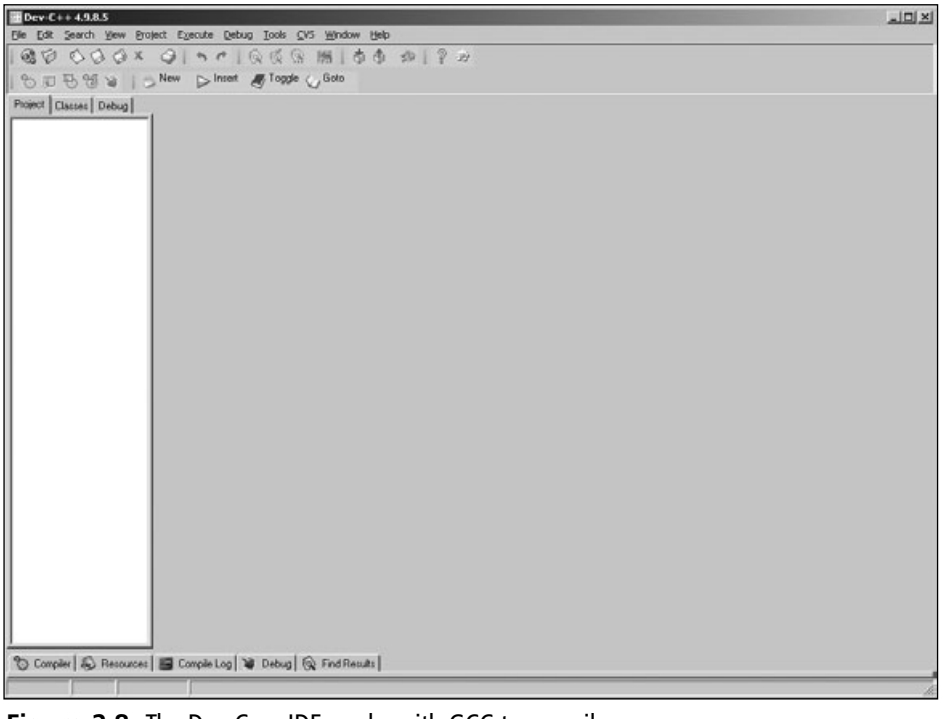


Figure 2.8 The Dev-C++ IDE works with GCC to compile programs.

Becoming Familiar with the Compiler

I understand that working with an open-source compiler can be a little unsettling. Not only is it very different than the compiler you might be used to, but it can be a little surprising to learn that the ultra-expensive commercial compiler that you (or your employer)

purchased works exactly the same way that the free compiler does. Even more surprising is the fact that GCC is an optimizing compiler capable of compiling code with every bit as much efficiency and speed as Visual C++ or Borland C++Builder. I think there is a false impression (furthered by marketing forces) that open-source software is inferior to commercial software and that proponents have simply gotten used to it. Although there is a small margin of truth in that, the fact remains that Dev-C++ works just as well as Visual C++ for constructing Windows programs. What you will not find is a dialog editor, a resource editor, a toll-free customer support number, or case-sensitive help (depending on the IDE).

Case-sensitive help is a very convenient feature if you are used to a commercial compiler package, such as Visual C++. Being able to hit F1 with the cursor over a key word to bring up syntax help is a difficult feature to do without. As an alternative, I like to keep a C reference book handy (such as *C Programming Language* (Prentice Hall PTR, 1988) by Brian Kernighan and Dennis Ritchie or *C: A Reference Manual* (Prentice Hall, 2002) by Samuel Harbison and Guy Steele) as well as an online Web site, such as http://www-ccs.ucsd.edu/c. I also keep the Allegro reference Web site open; the site is located at http://www.talula.demon.co.uk/allegro/onlinedocs/en. After you have programmed for a while without an online help feature, your coding skill will improve dramatically. It is amazing how very little some programmers really know about their choice programming language because they rely so heavily upon case-sensitive help! I don't suggest that you memorize the standard C and C++ libraries (although that wouldn't hurt). This might sound ridiculous at first, but it makes sense: When you have to make a little extra effort to look up some information, you are more likely to remember it and not need to look it up again.

In addition, open-source tools, such as Dev-C++, are not suited for .NET development—which, I might add, is not relevant because .NET is a framework for building business applications, not games, and it is not well suited for games. (In all fairness, Visual Basic .NET and Visual C# .NET are very good languages that do work well with DirectX, but they are not the ideal choice for game development.) You can treat my opinion on this matter as unbiased and objective because I use these tools on a daily basis, both commercial and open-source, and I appreciate the benefits that each tool brings with it. In general, commercial software is just more convenient. To an expert programmer, items of convenience usually only get in the way.

note

You might be using Visual C++ 7.0 in conjunction with this book. That is perfectly fine! Visual C++ is capable of compiling standard C/C++ code (this is called *unmanaged code* by Microsoft) as well as code that is reliant upon the .NET Framework (this is called *managed code*). Many commercial PC games are developed with Visual C++ 7.0 and DirectX, and this version will work with Allegro.

Creating the Greetings Project

Now then, back to Dev-C++. Open the File menu and select New, Project, as shown in Figure 2.9. This will bring up the New Project dialog box showing the types of projects that are available (see Figure 2.10). If you look at the tabs at the top of the dialog box, you will see Basic, Introduction, MultiMedia. These are the three different categories of project templates built into Dev-C++. Click on the Introduction tab to see a Hello World project (see Figure 2.11). The MultiMedia tab (shown in Figure 2.12) includes a sample project template for an OpenGL program. Note that if you have already installed Allegro.DevPak, you should see two Allegro project templates in the MultiMedia section.

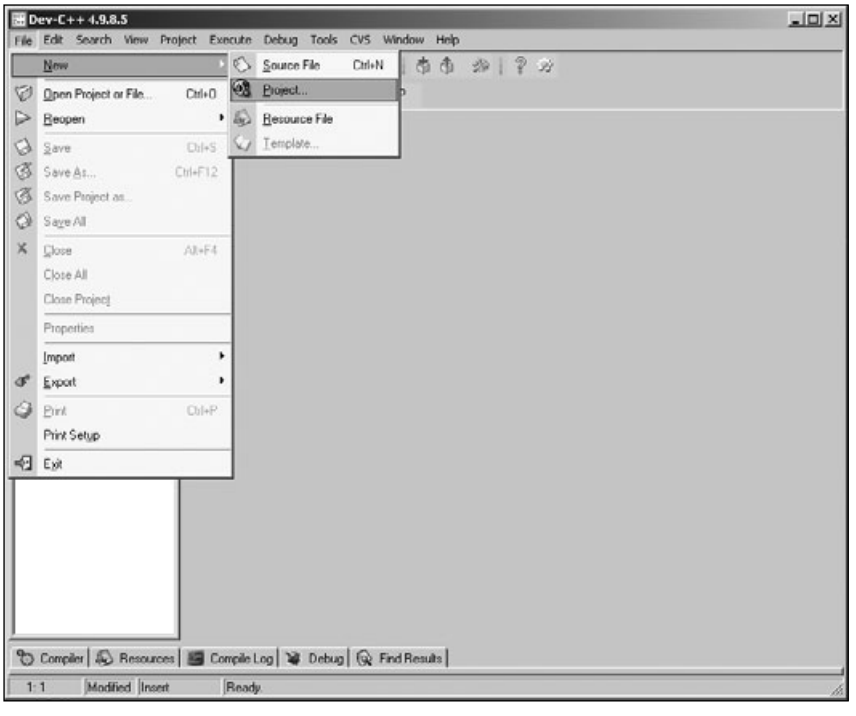


Figure 2.9 Creating a new project in Dev-C++

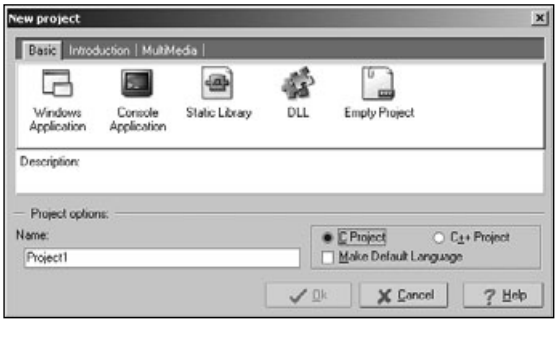


Figure 2.10 The New Project dialog box in Dev-C++ includes numerous project templates.

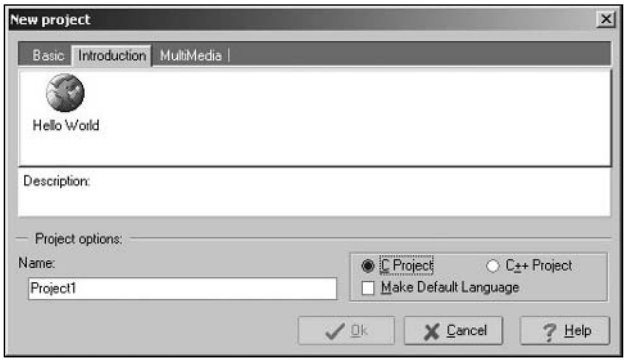


Figure 2.11 The Introduction tab includes a Hello World project template.

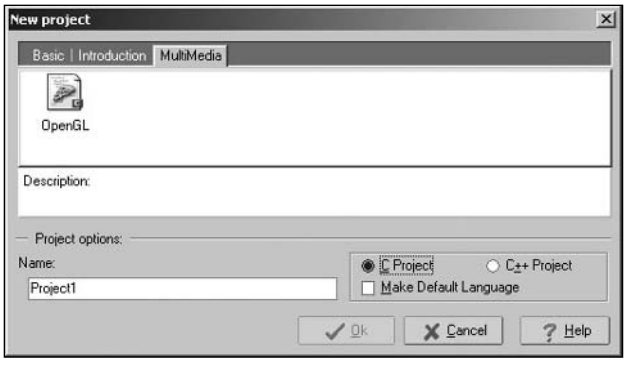


Figure 2.12 The MultiMedia tab includes an OpenGL project template.

Feel free to create a new project using any of these project templates and run it to see what the program looks like. After you are finished experimenting (which I highly recommend you do to become more familiar with Dev-C++), bring up the New Project dialog box again and select the Basic tab. At this point, allow me to provide you with a disclaimer, or rather, a look ahead. Allegro abstracts the operating system from your source code. Therefore, you need not create a Windows Application project (one of the options in the New Project dialog box). Allegro includes the code needed to handle Windows messages through WndProc, WinMain, and so on, just as the versions of Allegro for Linux, Mac OS X, and so on include the specific functions needed for those operating systems.

tip

For more information about the specifics of Windows programming, please refer to Charles Petzold's book *Programming Windows* (listed in Appendix D). Any edition will do, including the fifth edition or some of his newer books. I like the fifth edition because it covers Visual C++ 6.0, which is very similar to Dev-C++ and is easily configurable.

Referring to Figure 2.13, you want to select the Empty Project icon, and for the language choose C Project. For the project name, type Greetings, and then click on OK.

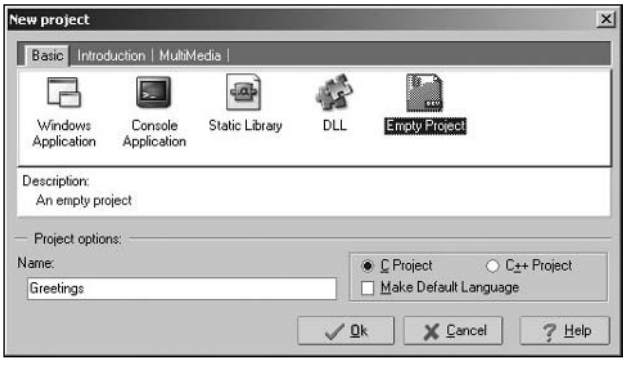


Figure 2.13 Choosing Empty Project from the New Project dialog box

The Project Save dialog box will then appear, allowing you to select a folder for the project. The default folder is inside the main Dev-Cpp folder. I recommend creating a folder off the root of your drive for storing projects.

tip

For future reference, the sample programs in this book are being developed simultaneously under Windows 2000 and Mandrake Linux, and the screenshots reflect this. If you are using another OS, such as Mac OS X or FreeBSD, your user interface will obviously look different.

After you save the new project, Dev-C++ will show the new empty project (see Figure 2.14). Note that Dev-C++ didn't even bother to create a default source file for you to use. That is because you selected Empty Project. Had you chosen Windows Application or another type of project, then a populated source code file would have been added for you. To keep things simple and to fully explain what's going on, I want to go over each step.

Now you need to add a new source code file to the project. Open the File menu and select New, Source File, as shown in Figure 2.15. Alternatively (and this is my preference) you can right-click on the project name to bring up a pop-up menu from which you can select New File (see Figure 2.16). Either method will add a new empty file called Untitled1 to your project.

Now right-click on the new file and select Rename File, and then type in **main.c** for the filename. After you do that, your project should look like the one shown in Figure 2.17.

note

If you are an experienced developer with Visual C++, Borland C++, Dev-C++, or another tool, these steps will be all too familiar to you. I am covering as much introductory information as possible now so it is not necessary to do so in later chapters.

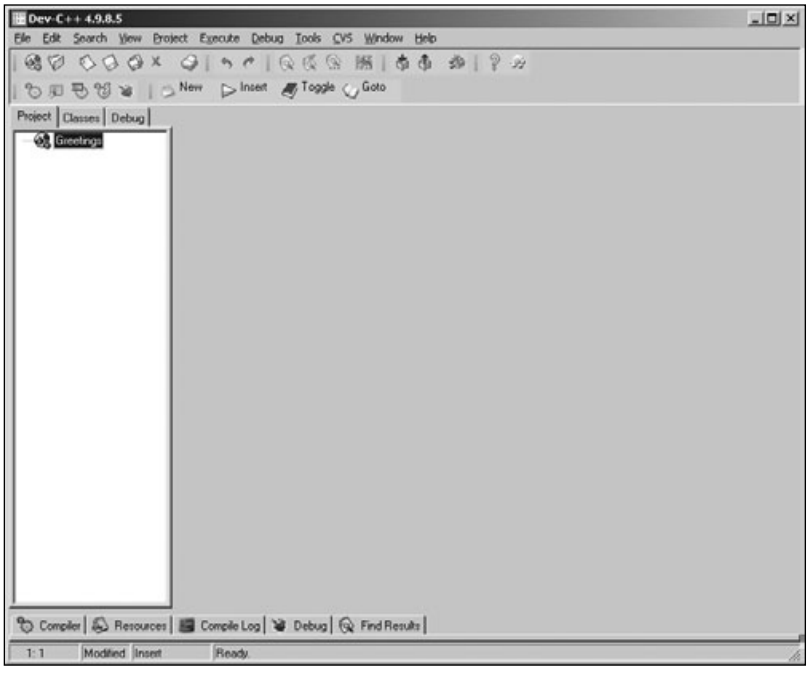


Figure 2.14 The new Greetings project has been created and is now ready to go.

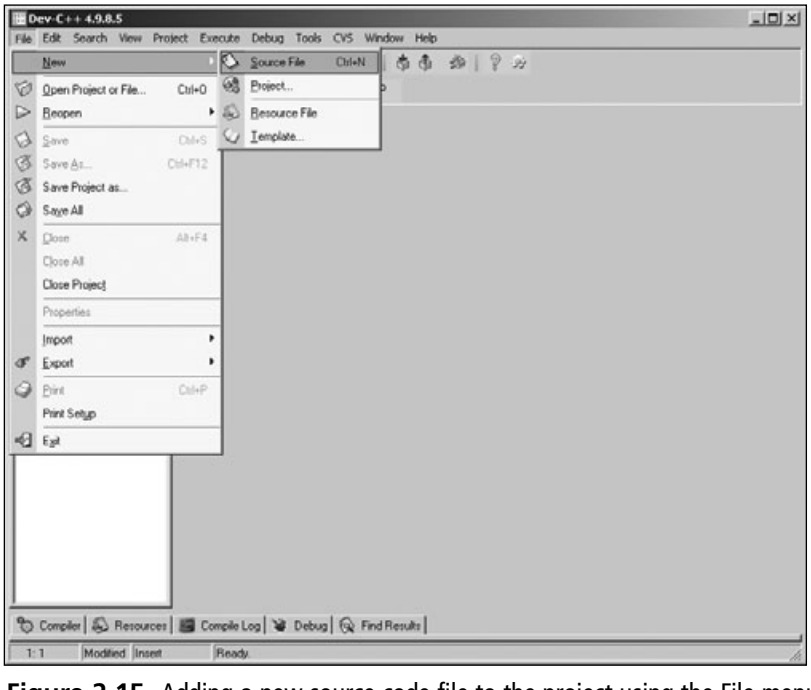


Figure 2.15 Adding a new source code file to the project using the File menu.

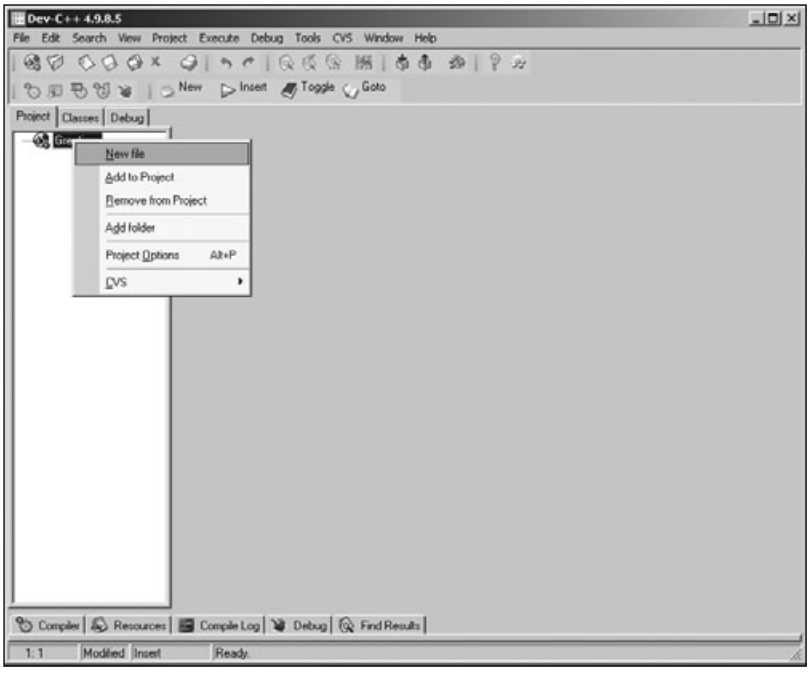


Figure 2.16 Adding a new source code file to the project using the right-click menu.

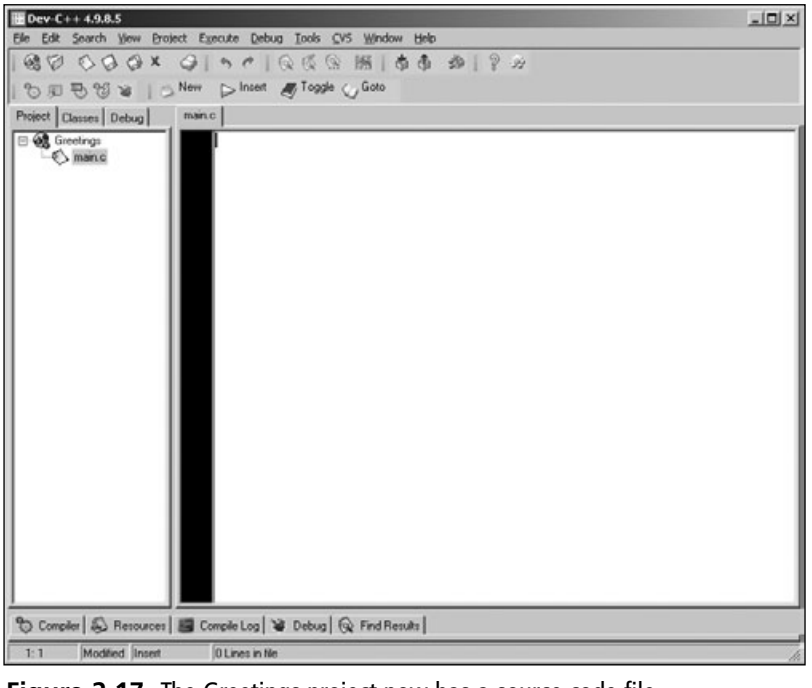


Figure 2.17 The Greetings project now has a source code file.

The Greetings Source Code

Now that you have a source code file, type in some source code to make sure Dev-C++ is configured properly. Here is a short program that you can type in:

```
#include <conio.h>
#include <stdio.h>
int main()
{
    printf("Greetings Earthlings.\n");
    printf("All your base are belong to us!\n");
    getch();
}
```

You can compile and run the program using several methods. Note that this program doesn't require Allegro to run at this point. (I'll stick to basic C right now.) The easiest way to compile and run the program is by pressing F9. You can also click on the Compile & Run (F9) icon on the toolbar or you can open the Execute menu and select Compile & Run. While you are browsing the toolbar and menus, note some of the other options available, such as Compile, Run, and Rebuild All. These options are occasionally helpful, although the compiler is so fast that I typically just hit F9. Because this is not an introductory book on C programming and I assume you have some experience writing C programs, I won't get into the basics of debugging and correcting syntax errors.

However, there is one thing that might prevent this program from running. If you look at the code listing, you'll notice that it doesn't include any header files and it is about as simple as things can get for a C program. This program assumes that it will be run on a console (such as a DOS prompt or shell prompt). Therefore, the project must be configured as a console project. The terminology will differ based on your OS, but for Windows the two most common project types are Windows Application and Console Application. Open the Project menu and select Project Options. The Project Options dialog box will appear, as shown in Figure 2.18.

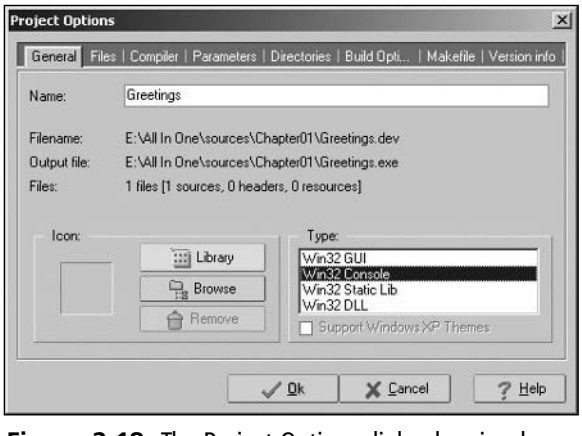


Figure 2.18 The Project Options dialog box is where you can change the project settings.

Pay special attention to the list of project types and make sure that Win32 Console is selected. (This should have been the default when you created a new blank project; however, future versions of Dev-C++ may change the default option or any other feature deemed necessary to improve the IDE.) If Win32 Console is selected, then you are ready to run the program. Close the dialog box, and then press F9 to compile and run the program.

note

Feel free to open multiple instances of Dev-C++ if you are working on several C or C++ projects at the same time or if you would like to copy code from one source listing to another. Dev-C++ has a small footprint of only around 12 MB of memory, and multiple instances of it run off the first memory instance.

Running the Greetings Program

If all goes well you should see the program run as in Figure 2.19, which shows the console window superimposed over Dev-C++. As the source code indicates (note the `getch()` function), press a key to end the program.

If the compile process failed, first check to make sure there are no typos in the source code you entered. If the code looks good, you might want to refer back to the "Installing and Configuring Dev-C++ and Allegro" section to see whether you might have missed a step

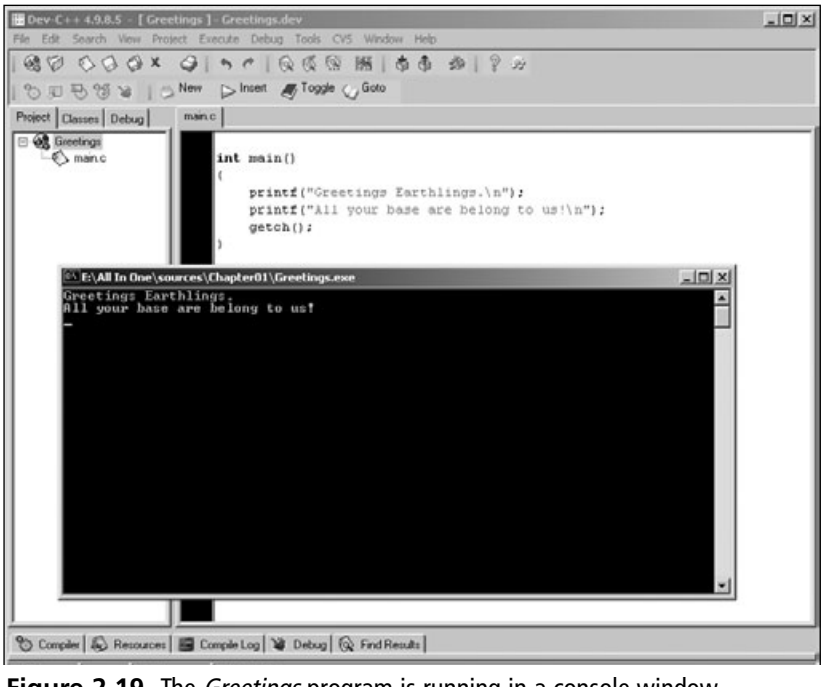


Figure 2.19 The *Greetings* program is running in a console window.

that is preventing the compiler from running as it should. The install process is fairly simple and straightforward (ignoring the update process, at any rate), so if you continue to have problems, you might seek help at the Dev-C++ Web site at http://www.bloodshed.net/devcpp.html. A program this simple should compile and run without any problem, so any error at this point is an installation problem if anything.

Testing Allegro: The GetInfo Program

Now you should give Allegro a spin and make sure it was compiled and installed correctly. The next program you'll write will be similar to the last one because it will be a console program. The difference is that this program will include the Allegro library. Go ahead and open a new instance of Dev-C++ (or close the current project). Open the File menu and select New, Project as before. This time, however, instead of creating an empty project, select Console Application (see Figure 2.20). For the project name, type **GetInfo**.

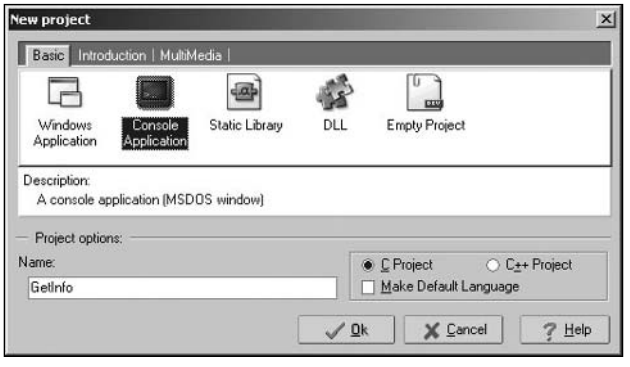


Figure 2.20 Creating a new console application in Dev-C++

When the new project is created, Dev-C++ will add a main.c file for you and fill it with some basic code. Delete the template code because you'll be typing in your own code.

Introducing Some of Allegro's Features

The first function that you need to know is `allegro_init`, which has this syntax:

```
int allegro_init();
```

This function is required because it initializes the Allegro library. If you do not call this function, the program will probably crash (at worst) or simply not work (at best). In addition to initializing the library, `allegro_init` also fills a number of global string and number variables that you can use to display information about Allegro. One such variable is a string called `allegro_id`; it is declared like this:

```
extern char allegro_id[];
```

You can use `allegro_id` to display the version number for the Allegro library you have installed. That is a good way to check whether Allegro has been installed correctly, so you should write some code to display `allegro_id`. Referring to the *GetInfo* project you just created, type in the following code:

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

int main()
{
        allegro_init();
        printf("Allegro version = %s\n", allegro_id);
        printf("\nPress any key...\n");
        getch();
        return 0;
}
END_OF_MAIN();
```

You are probably wondering what the heck that `END_OF_MAIN` function at the bottom of the source listing is. This is actually a macro that is used by Allegro and helps with the multi-platform nature of the library. This is odd, but the macro simply must follow the `main` function in every program that uses Allegro. You'll get used to it (and quickly begin to ignore it after a while).

Including the Allegro Library File

One more thing. Before you can run the program, you must add the Allegro library file to the *GetInfo* project. The library file is called liballeg.a and can be found in the \allegro\lib folder. (Depending on where you installed it, that might be C:\allegro\lib.) To add the library file, open the Project menu and select Project Options. There are a number of tabs in the Project Options dialog box. Locate the Parameters tab, which is shown in Figure 2.21.

You now want to add an entry into the third column (labeled Linker) so the Allegro library file will be linked into the executable program. You can type in the path and filename directly or you can click on the Add Library or Object button to search for the file. Navigate to your root Allegro folder and look inside a folder called lib.

If you installed Allegro using the Dev-C++ WebUpdate or by installing the DevPak off the CD-ROM, then Allegro will be installed to C:\Dev-Cpp\Allegro by default. If you are at all confused about this issue, then I recommend you visit Appendix F to get a better feel for how Allegro and Dev-C++ work together.

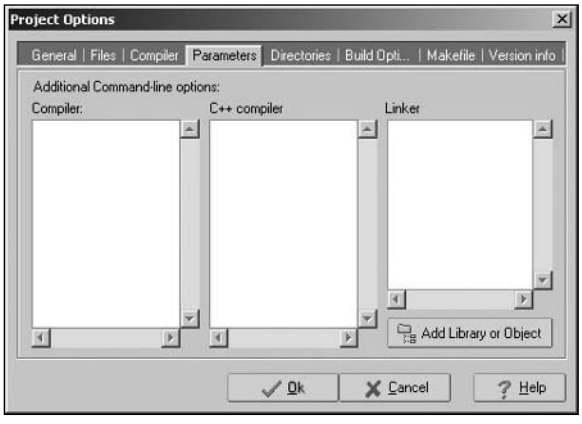


Figure 2.21 The Parameters tab in the Project Options dialog box

You should see eight compiler-specific folders inside lib:

- bcc32
- beos
- djgpp
- mingw32
- msvc
- qnx
- unix
- watcom

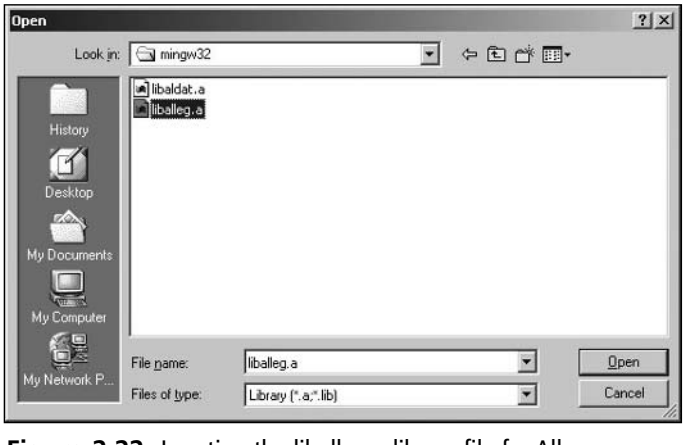


Figure 2.22 Locating the liballeg.a library file for Allegro

As you might recall from the "Installing and Configuring Dev-C++ and Allegro" section, the version you want to use for Windows is mingw32, so go ahead and open that folder. If you have compiled Allegro for mingw32 you should see two files inside—libaldat.a and liballeg.a (see Figure 2.22).

Select the liballeg.a file and click on Open to load the path name for the file into the Linker list. If you look at the path name that was inserted, you'll notice that it includes a lot of folder redirection (../../../../allegro/lib/mingw32/liballeg.a). This will probably look different on your system due to where you saved the project file. (Mine is stored several folders deep in the source code folder.) For future reference, note that you only need to refer to the absolute path name for liballeg.a (or any other library file). Therefore, you can edit the Linker text so it looks like this:

```
/allegro/lib/mingw32/liballeg.a
```

The result should look like Figure 2.23.

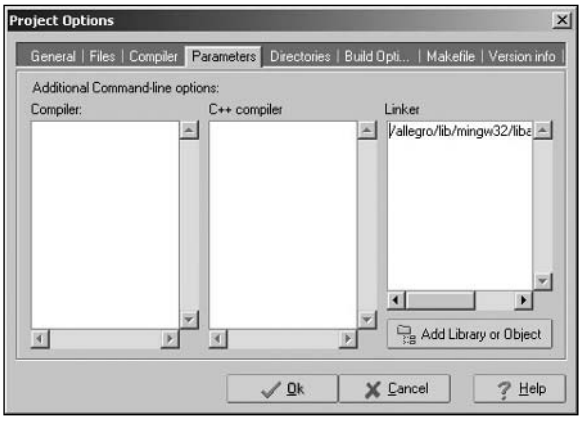


Figure 2.23 You can also type the path name to a library file directly into the Linker list.

Before you get too comfortable with this plan, let me give you a heads up on an even easier way to include the Allegro library! Regardless of whether you installed Allegro.DevPak or compiled the Allegro source code, the liballeg.a file will be installed at \Dev-Cpp\lib. So you really don't need to reference the file in \allegro\lib directly. Referring back to Figure 2.23, you can substitute the path to liballeg.a with a simple linker command (-lalleg) and that will suffice! I will remind you how to set up the projects as we go along, using both methods. This chapter is really thorough in these explanations because future chapters will skim over these details. If you ever have trouble configuring a new project for Allegro, this is the chapter you will want to refer back to as a reference.

tip

If you are using Visual C++, you will want to reference alleg.lib (and no other library files) in the linker options field. See Appendix E for details.

Running the GetInfo Program

If you haven't already, press F9 to compile and run the program. If all goes well, you should be rewarded with a console window that looks like the one in Figure 2.24. If you have problems running the program, aside from syntax errors due to typos you might want to double-check that you have the correct path to the liballeg.a library file.

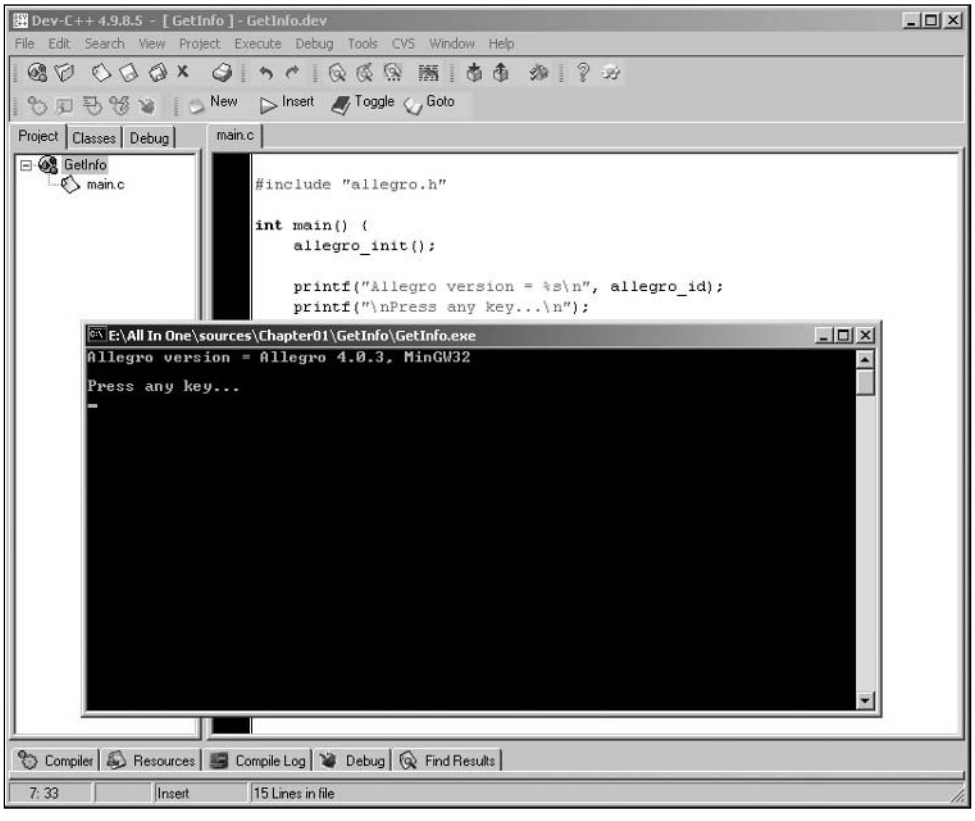


Figure 2.24 The *GetInfo* program displays information about the Allegro library.

Adding to the GetInfo Program

Now you can add some more functionality to the *GetInfo* program to explore more of the functions available with Allegro. First, let me introduce you to a variable called `os_type`, which has this declaration:

```
extern int os_type;
```

This variable returns a value for the operating system that Allegro detected, and may be one of the values listed in Table 2.1.

To display the operating system name, you'll need to use the `switch` statement to determine which OS it is. This would be easier using a string array, but unfortunately the list might not be in consecutive order within Allegro, so it is safer to use a `switch`. Add the following function above the `int main()` line:

Table 2.1 Operating Systems Recognized by Allegro

Identifier	Description
OSTYPE_UNKNOWN	Unknown (may be MS-DOS)
OSTYPE_WIN3	Windows 3.1 or earlier
OSTYPE_WIN95	Windows 95
OSTYPE_WIN98	Windows 98
OSTYPE_WINME	Windows Me
OSTYPE_WINNT	Windows NT
OSTYPE_WIN2000	Windows 2000
OSTYPE_WINXP	Windows XP
OSTYPE_OS2	OS/2
OSTYPE_WARP	OS/2 Warp 3
OSTYPE_DOSEMU	Linux DOSEMU
OSTYPE_OPENDOS	Caldera OpenDOS
OSTYPE_LINUX	Linux
OSTYPE_FREEBSD	FreeBSD
OSTYPE_QNX	QNX
OSTYPE_UNIX	UNIX variant
OSTYPE_BEOS	BeOS
OSTYPE_MACOS	Mac OS

```
char *OSName(int number)
{
     switch (number)
     {
          case OSTYPE_UNKNOWN: return "Unknown or MS-DOS";
          case OSTYPE_WIN3:    return "Windows";
          case OSTYPE_WIN95:   return "Windows 95";
          case OSTYPE_WIN98:   return "Windows 98";
          case OSTYPE_WINME:   return "Windows ME";
          case OSTYPE_WINNT:   return "Windows NT";
          case OSTYPE_WIN2000: return "Windows 2000";
          case OSTYPE_WINXP:   return "Windows XP";
          case OSTYPE_OS2:     return "OS/2";
          case OSTYPE_WARP:    return "OS/2 Warp 3";
          case OSTYPE_DOSEMU:  return "Linux DOSEMU";
          case OSTYPE_OPENDOS: return "Caldera OpenDOS";
          case OSTYPE_LINUX:   return "Linux";
          case OSTYPE_FREEBSD: return "FreeBSD";
```

```
        case OSTYPE_QNX:      return "QNX";
        case OSTYPE_UNIX:     return "Unix variant";
        case OSTYPE_BEOS:     return "BeOS";
        case OSTYPE_MACOS:    return "MacOS";
    }
}
```

Now you can modify the main routine to display the name of the operating system. Add the following line of code following the first `printf` line:

```
printf("Operating system = %s\n", OSName(os_type));
```

When you run the program (F9), you should see a console window with output that looks like the following lines. (Note that your operating system should be displayed if you are not running Windows 2000.)

```
Allegro version  = Allegro 4.0.3, MinGW32
Operating system = Windows 2000

Press any key...
```

Now you can use a few more of Allegro's very useful global variables to retrieve the operating system version, desktop resolution, multitasking flag, color depth, and some details about the processor. Since you are already raring to go, I'll just list the definitions for these functions and variables, and then you can add them to the *GetInfo* program. (Remember that none of these variables and functions will work unless you have called `allegro_init()` first.)

```
extern int os_version;
extern int os_revision;
extern int os_multitasking;
int desktop_color_depth();
int get_desktop_resolution(int *width, int *height);
extern char cpu_vendor[];
extern int cpu_family;
extern int cpu_model;
extern int cpu_capabilities;
```

The first three variables provide information about the operating system version, revision, and whether it is multitasking. The following lines of code will display those values:

```
printf("OS version          = %i.%i\n", os_version, os_revision);
printf("Multitasking        = %s\n", YesNo(os_multitasking));
```

I wrote a short function called `YesNo()` to display the appropriate word (yes = 1, no = 0); this function should be typed in above `int main()`:

```
char *YesNo(int number)
{
    if (number==0)
        return "No";
    else
        return "Yes";
}
```

Next are the desktop resolution and color depth values, which you can add to the program with the following lines:

```
int width, height;
get_desktop_resolution(&width, &height);
printf("Desktop resolution = %i x %i\n", width, height);
printf("Color depth        = %i bits\n", desktop_color_depth());
```

Notice how you must pass the `width` and `height` variables to `get_desktop_resolution()`? The variables are passed by reference to this function so you can then use the variables to display the desktop resolution. Color depth is a direct function call.

Next come the functions associated with the processor. I don't know about you, but I personally find this information very interesting. You could use these values to directly affect how a game runs by enabling or disabling certain features based on system specifications. When it comes to a multi-platform library, this can be essential because there are many older PCs running Linux and other OSs that perform well on older hardware (whereas Windows typically puts a high demand on resources). Here are the processor-specific variables and functions:

```
extern char cpu_vendor[];
extern int cpu_family;
extern int cpu_model;
extern int cpu_capabilities;
```

The first three variables are easy enough to read, although they are manufacturer-specific values. For instance, a `cpu_family` value of 6 indicates a Pentium Pro for the Intel platform, while it refers to an Athlon for the AMD platform. The `cpu_capabilities` variable is a little more complicated because it contains packed values specifying the special features of the processor. Table 2.2 presents a rundown of those capabilities.

I have always enjoyed system decoding programs like this one, so it is great that this is built into Allegro. To decode the `cpu_capabilities` variable, you can `AND` one of the identifiers with `cpu_capabilities` to see whether it is available. If the `AND` operation equals the identifier value, then you know that identifier has been bit-packed into `cpu_capabilities`. Here is the code to display these capabilities. (Note that spacing is not critical—I just wanted all of the equal signs to line up.)

Table 2.2 Processor Features Identified by Allegro

Identifier	Description
CPU_ID	cpuid is available.
CPU_FPU	x87 FPU is available.
CPU_MMX	MMX is available.
CPU_MMXPLUS	MMX+ is available.
CPU_SSE	SSE is available.
CPU_SSE2	SSE2 is available.
CPU_3DNOW	3DNow! is available.
CPU_ENH3DNOW	Enhanced 3DNow! is available.

```
int caps = cpu_capabilities;
printf("Processor ID           = %s\n",
    YesNo((caps & CPU_ID)==CPU_ID));
printf("x87 FPU                = %s\n",
    YesNo((caps & CPU_FPU)==CPU_FPU));
printf("MMX                    = %s\n",
    YesNo((caps & CPU_MMX)==CPU_MMX));
printf("MMX+                   = %s\n",
    YesNo((caps & CPU_MMXPLUS)==CPU_MMXPLUS));
printf("SSE                    = %s\n",
    YesNo((caps & CPU_SSE)==CPU_SSE));
printf("SSE2                   = %s\n",
    YesNo((caps & CPU_SSE2)==CPU_SSE2));
printf("3DNOW                  = %s\n",
    YesNo((caps & CPU_3DNOW)==CPU_3DNOW));
printf("Enhanced 3DNOW         = %s\n",
    YesNo((caps & CPU_ENH3DNOW)==CPU_ENH3DNOW));
```

For reference, here is the complete listing for the `main` function of *GetInfo*, with some additional comments to clarify what each section of code is doing.

```
int main() {
    //initialize Allegro
    allegro_init();

    //display version info
    printf("Allegro version    = %s\n", allegro_id);
    printf("Operating system   = %s\n", OSName(os_type));
```

```
    printf("OS version          = %i.%i\n", os_version, os_revision);
    printf("Multitasking       = %s\n", YesNo(os_multitasking));

    //display system info
    int width, height;
    get_desktop_resolution(&width, &height);
    printf("Desktop resolution = %i x %i\n", width, height);
    printf("Color depth         = %i bits\n", desktop_color_depth());
    printf("Processor vendor   = %s\n", cpu_vendor);
    printf("Processor family   = %i\n", cpu_family);
    printf("Processor model    = %i\n", cpu_model);

    //display processor capabilities
    int caps = cpu_capabilities;
    printf("Processor ID       = %s\n",
        YesNo((caps & CPU_ID)==CPU_ID));
    printf("x87 FPU             = %s\n",
        YesNo((caps & CPU_FPU)==CPU_FPU));
    printf("MMX                 = %s\n",
        YesNo((caps & CPU_MMX)==CPU_MMX));
    printf("MMX+                = %s\n",
        YesNo((caps & CPU_MMXPLUS)==CPU_MMXPLUS));
    printf("SSE                 = %s\n",
        YesNo((caps & CPU_SSE)==CPU_SSE));
    printf("SSE2                = %s\n",
        YesNo((caps & CPU_SSE2)==CPU_SSE2));
    printf("3DNOW               = %s\n",
        YesNo((caps & CPU_3DNOW)==CPU_3DNOW));
    printf("Enhanced 3DNOW     = %s\n",
        YesNo((caps & CPU_ENH3DNOW)==CPU_ENH3DNOW));

    printf("\nPress any key...\n");
    getch();
    return 0;
}
```

Running the program now produces the following results. (Note that it will reflect the hardware in your own system.)

```
Allegro version    = Allegro 4.0.3, MinGW32
Operating system   = Windows 2000
OS version         = 5.0
```

```
Multitasking         = Yes
Desktop resolution   = 1280 x 1024
Color depth          = 32 bits
Processor vendor     = AuthenticAMD
Processor family     = 6
Processor model      = 4
Processor ID         = Yes
x87 FPU              = Yes
MMX                  = Yes
MMX+                 = Yes
SSE                  = No
SSE2                 = No
3DNOW                = Yes
Enhanced 3DNOW       = Yes

Press any key...
```

Gaining More Experience with Allegro

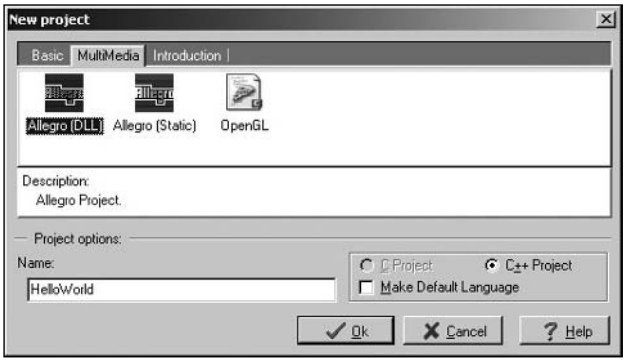
Now that you have learned how to set up the compiler to use Allegro by manually adding the library file to the project, I will show you the easy way to do it! I believe it's always best to know how to set up a project first, but the Allegro.DevPak includes two project templates you can use, so it's a cinch to create a new Allegro project and get started writing code without having to go into the Project Options at all.

I mention this after the fact because a default installation of Dev-C++ and Allegro does not include these Allegro project templates. Only after you install Allegro via WebUpdate will you find these templates installed (which is one good reason for using the WebUpdate tool).

The only drawback to using these Allegro project templates is that they are specifically limited to C++ code, not C (which is mainly what this book focuses on). That is not a limitation really, because you can still write straight C code and it won't make any difference (due to the way C++ headers are handled in the project template). However, you can feel free to write C or C++ code as you wish, so this might be a better solution than limiting the project to C by default.

The Hello World Demo

Now let's see how easy it is to create a new Allegro project. Fire up Dev-C++ and open the File menu. Select New, Project and click on the MultiMedia tab, and you should see three project types—Allegro (DLL), Allegro (Static), and OpenGL—as shown in Figure 2.25.



Select the Allegro (DLL) project template, type a new name for the project, and click on OK. A new project will be created in Dev-C++, and you will be asked to choose a location for the project file. After the project has been created, you should see the sample source code shown in Figure 2.26.

Figure 2.25 Creating a new Allegro (DLL) project in Dev-C++

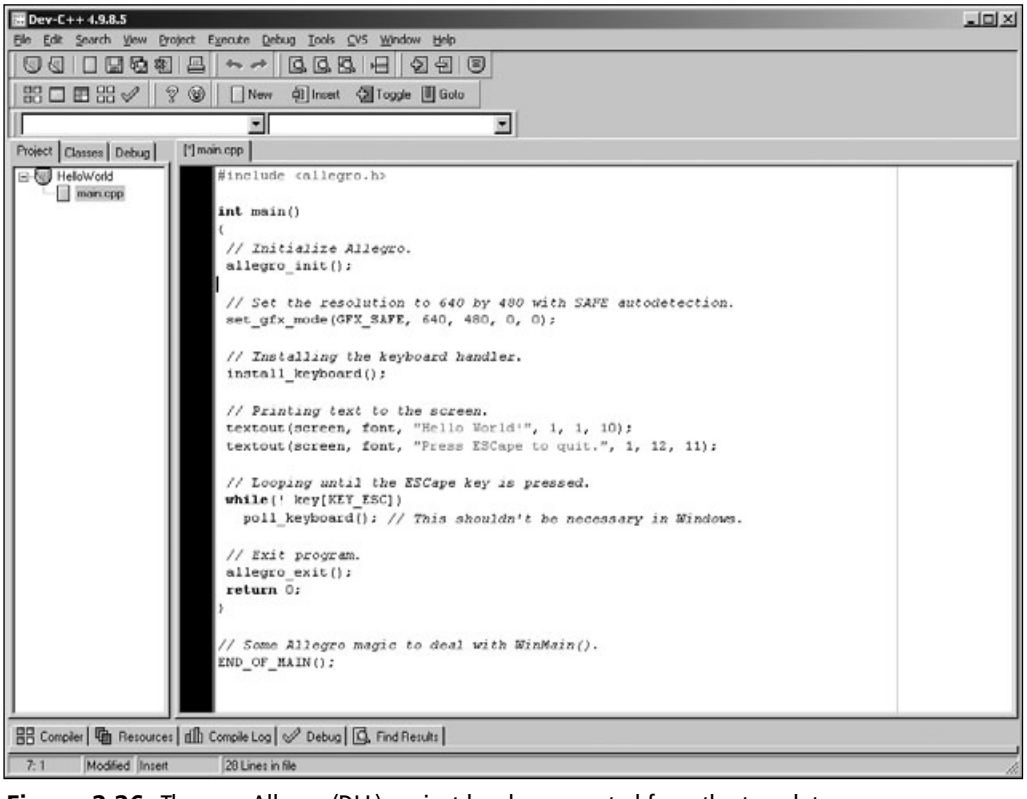


Figure 2.26 The new Allegro (DLL) project has been created from the template.

If you run the program by pressing F9, you should see the program run full-screen with the message "Hello World" displayed (see Figure 2.27).

Figure 2.27 The *HelloWorld* program runs in full-screen 640×480 mode.

This is a fully-functional Allegro program that didn't require any configuration. If you are using a Windows system, Allegro automatically supports DirectX and takes advantage of hardware acceleration with DirectDraw. On other platforms (such as Linux), Allegro will use whatever library has been compiled with it to make the most out of that platform (in other words, the DirectX equivalent on each system).

If for any reason you are not able to run the program as shown, go back to the "Installing Allegro" section to make sure it is installed correctly.

Allegro Sample Programs

Allegro comes with a large number of sample programs that demonstrate all of the various features of the library. If you installed Allegro as described in this chapter using the DevPak, you will find these sample programs in the main Dev-Cpp folder on your hard drive. Assuming you have installed it at C:\Dev-Cpp, the example programs are located in C:\Dev-Cpp\Examples\Allegro.

Before you can run an individual C source program in Dev-C++, you need to copy it into an existing project that has been configured with the Allegro library. Otherwise, the compiler will complain that one or more Allegro functions could not be found. The easiest way to do this is to create a new Allegro (DLL) project, as you did with *HelloWorld* a few minutes ago. Then you can open one of the sample programs in Dev-C++ and paste the

new code into main.c to run it. That way the project template is configured for Allegro and you can repeatedly paste sample code into main.c to see the sample programs run. The alternative is to create a separate project for each program or compile them all using a make file or by running GCC from the command line (which is probably easier than using Dev-C++, but not as convenient).

note

The Allegro sample programs are contributions from many Allegro developers and fans. They are not all guaranteed to work, especially considering that new versions of Allegro are released frequently and the examples are not always kept up to date.

For an example, take a look at the ex3buf.c example program. You can load this program from \Dev-Cpp\Examples\Allegro (assuming you have installed Dev-C++ to this folder), as shown in Figure 2.28. This program is a triple-buffer demonstration written by Shawn Hargreaves. Although it was primarily written for MS-DOS (as evidenced by the 320×200 video resolution), you can still run the program in Windows or any other system in full-screen mode.

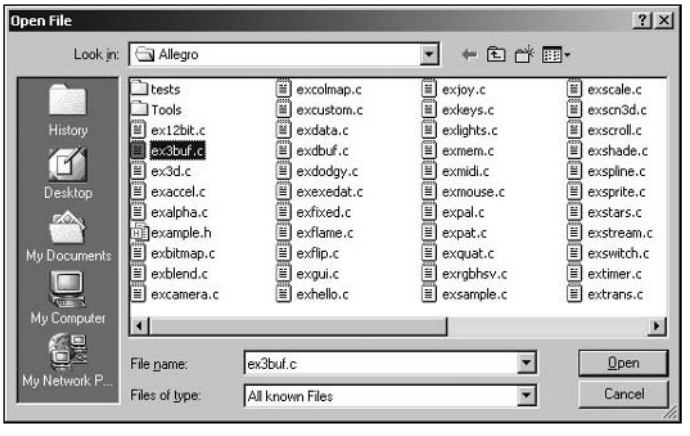


Figure 2.28 Opening one of the sample programs installed with Allegro

This is actually a very interesting program because it uses an early polygon rendering routine that was written before the 3D features were added to Allegro. You could adapt this code to produce a shaded 3D game, but that would be hard work—better to wait until I cover the 3D functionality built into Allegro first! Figure 2.29 shows the program running.

Another interesting program is called exblend.c; it is also found in \Dev-Cpp\Examples\Allegro. This was also written by Shawn Hargreaves, and it shows an interesting alpha-blend effect that I will cover in the next chapter (see Figure 2.30).

There are many more sample programs just like these in \Dev-Cpp\Examples\Allegro that I encourage you to load and run to see some of the things that Allegro is capable of doing.

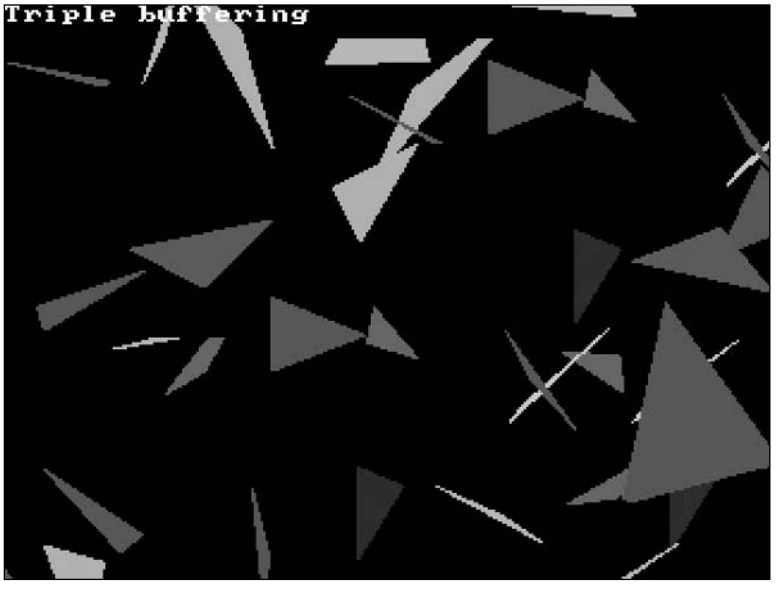


Figure 2.29 The ex3buf.c program (written by Shawn Hargreaves) is one of the many sample programs included with Allegro.

Figure 2.30 The exblend.c program shows how two bitmaps can be displayed with translucency.

These aren't full programs or games per se, but they do a good job of demonstrating simple concepts.

Summary

That sums up the introduction to Dev-C++ and Allegro. I hope that by this time you are at least familiar with the IDE and have a good understanding of how the compiler works, as well as what Allegro is capable of (with a little effort on your part). This chapter has given you few tools for building a game as of yet, but it was necessary along that path. Installing and configuring the dev tools is always a daunting task for those who are new to programming, and even experienced programmers get lost when trying to get up and running with a new IDE, compiler, and game library. Not only did you learn to configure a new IDE and open-source compiler, you also got started writing programs using an open-source game library. But now that the logistics are out of the way, you can focus on learning Allegro and writing a few sample programs in the following chapters.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What game features an Avatar and takes place in the land of Brittania?
   A. *Baldur's Gate: Dark Alliance*
   B. *Ultima VII: The Black Gate*
   C. *The Elder Scrolls III: Morrowind*
   D. *Wizardry 8*

2. GNU is an acronym for which of the following phrases?
   A. GNU is Not UNIX
   B. Great Northern University
   C. Central Processing Unit
   D. None of the above

3. What is the primary Web site for Dev-C++?
   A. http://www.microsoft.com
   B. http://www.bloodshed.net
   C. http://www.borland.com
   D. http://www.fsf.org

4. What is the name of the compiler used by Dev-Pascal?
   A. GNU Pascal
   B. Turbo Pascal
   C. Object Pascal
   D. Microsoft Pascal

5. What is the name of the powerful automated update utility for Dev-C++?
   A. DevUpdate
   B. AutoUpdate
   C. Windows Update
   D. WebUpdate

6. What are the Dev-C++ update packages called?
   A. DevPacks
   B. DevPaks
   C. DevPackages
   D. DevSpanks

7. What distinctive feature of Dev-C++ sets it apart from commercial development tools?
   A. Dev-C++ is open-source
   B. Dev-C++ is free
   C. Dev-C++ is multi-platform
   D. All of the above

8. What is the name of the game programming library featured in this chapter?
   A. DirectX
   B. Gnome
   C. GTK+
   D. Allegro

9. What function must be called before you use the Allegro library?
   A. `main()`
   B. `byte_me()`
   C. `allegro_init()`
   D. `lets_get_started()`

10. What statement must be included at the end of `main()` in an Allegro program?
    A. `END_OF_THE_WORLD()`
    B. `END_OF_MAIN()`
    C. `END_OF_FREEDOM()`
    D. `AH_DONUTS()`

This page intentionally left blank

CHAPTER 3

Basic 2D Graphics Programming with Allegro

This hands-on chapter introduces you to the powerful graphics features built into Allegro. In the early years of personal computers, at a time when 3D accelerators with 256 MB of DDR memory were inconceivable, vector graphics provided a solid solution to the underpowered PC. For most games, a scrolling background was not even remotely possible due to the painfully slow performance of the early IBM PC. While competing systems from Atari, Amiga, Commodore, Apple, and others provided some of the best gaming available at the time with performance that would not be matched in consoles for many years, these PCs fell to the wayside as the IBM PC (and its many clone manufacturers) gained market dominance—not without the help of Microsoft and Intel. It is a shame that Apple is really the only contender that survived the personal computer revolution of the 1980s and 1990s, but it was mainly that crucible that launched gaming forward with such force.

This chapter is somewhat a lesson in progressive programming, starting with basic concepts that grow in complexity over time. Because we are pushing the 2D envelope to the limit throughout this book, it is fitting that we should start at the beginning and cover vector graphics. The term *vector* describes the CRT (*Cathode Ray Tube)* monitors of the past and the vector graphics hardware built into the computers that used this early technology.

A more descriptive term for the subject of this chapter would be "programming graphics primitives." A *graphics primitive* is a function that draws a simple geometric shape, such as a point, line, rectangle, or circle. This chapter covers the graphics primitives built into Allegro with complete sample programs for each function so you will have a solid understanding of how these functions work. I should point out also that these graphics primitives form the basis of all 3D graphics, past and present; after all, the mantra of the 3D card is the holy polygon. But above all, I want you to have some fun with this chapter.

Whether you are a skilled programmer or a beginner, try to have some fun in everything you do. I believe even an old hand will find something of interest in this chapter.

Here is a breakdown of the major topics in this chapter:

- Understanding graphics fundamentals
- Drawing graphics primitives
- Printing text on the screen

Introduction

I don't know about you, but I was drawn to graphics programming before I became interested in actually writing games. The subject of computer graphics is absolutely fascinating and is at the forefront of computer technology. The high-end graphics accelerator cards featuring graphics processors with high-speed video memory, such as the NVIDIA GeForce FX and ATI Radeon 9800, are built specifically to render graphics insanely fast. The silicon is not designed merely to satisfy a marketing initiative or to best the competition (although that would seem to be the case). The graphics chips are designed to render graphics with great efficiency using hardware-accelerated functions that were once calculated in software. I emphasize the word "graphics" because we often take it for granted after hearing it used so often. Figure 3.1 shows a typical monitor.

The fact of the matter is that video cards are not designed to render games; they are designed to render geometric primitives with special effects. As far as the video card is concerned, there is only one triangle on the screen. It is the programmer who tells the video card to move from one triangle to the next. The video card does this so quickly (on the order of 100 million or more polygons per second) that it fools the viewer into believing that the video card is rendering an entire scene on its own. The triangles are hidden away in the matrix of the scene (so to speak), and it is becoming more and more difficult to discern reality from virtual reality due to the advanced features built into the latest graphics chips (see Figure 3.2).

Figure 3.1 A typical monitor displays whatever it is sent by the video card.

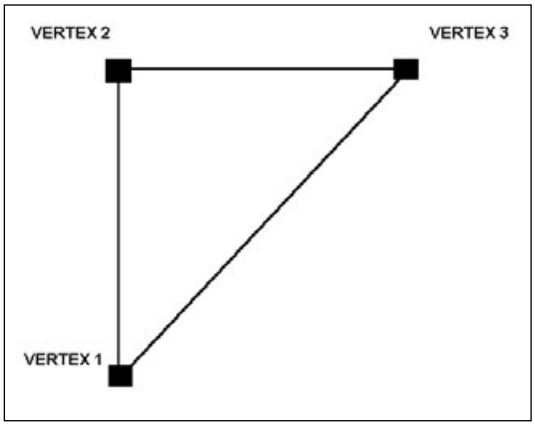


Figure 3.2 A typical 3D accelerator card sees only one triangle at a time.

Taken a step closer, one would notice that each triangle is made up of three points, or *vertices*, which is really all the graphics chip cares about. Filling pixels between the three points and applying varying effects (such as lighting) are tasks that the graphics chip has been designed to do quickly and efficiently.

Years ago, when a new video card was produced, the manufacturer would hire a programmer to write the device driver software for the new hardware, usually for Windows and Linux. That device driver was required to provide a specific set of common functions to the operating system for the new video card to work correctly. The early graphics chips were very immature (so to speak); they were only willing to switch video modes and provide access to the video memory (or frame buffer), usually in banks—another issue of immaturity. As graphics chips improved, silicon designers began to incorporate some of the software's functionality right into the silicon, resulting in huge speed increases (orders of greater magnitude) over functions that had previously existed only in software.

The earliest *Windows accelerators*, as they were known, produced for Windows 3.1 and Windows 95 provided hardware blitting. *Blit* is a term that means bit-block transfer, a method of transferring a chunk of memory from one place to another. In the case of a graphical blit, the process involves copying a chunk of data from system memory through the bus to the memory present on the video card. In the early years of the PC, video cards were lucky to have 1 MB of memory. My first VGA card had 256 KB (see Figure 3.3)!

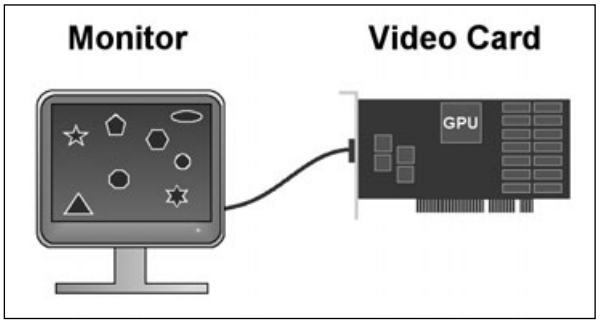


Figure 3.3 The modern video card has taken over the duties of the software driver.

Contrast this with the latest 3D cards that have 256 MB of DDR (*Double Data Rate*) memory and are enhanced with direct access to the AGP bus! The latest DDR memory at the time of this writing is PC-4000, also called DDR-500. This type of memory comes on a 184-pin socket with a throughput of 4 gigabytes per second. Although the latest video cards don't use this type of high-speed memory yet, they are

close, using DDR-333. The point is, this is insanely fast memory! It simply must be as fast as possible to keep feeding the ravenous graphics chip, which eats textures in video memory and spews them out into the frame buffer, which is sent directly to the screen (see Figure 3.4).

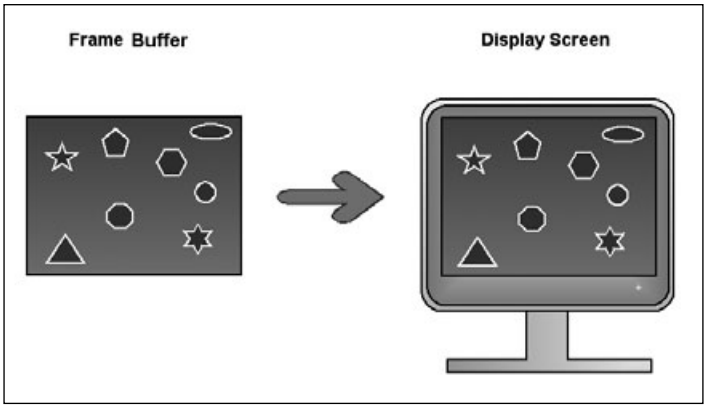


Figure 3.4 The frame buffer, located in video memory, is transferred directly to the screen.

In a very real sense, the graphics card is a small computer on its own. When you consider that the typical high-end PC also has a high-performance sound processing card (such as the Sound Blaster Audigy 2 by Creative Labs) capable of Dolby DTS and Dolby Digital 5.1 surround sound, what we are really talking about here is a multiprocessor system. If your first impression is to scoff at the idea or shrug it off like an old joke, think about it again. The typical $200 graphics card or sound card has more processing power than a Cray supercomputer had in the mid-1980s. Considering that a gaming rig has these two major subsystems in addition to an insanely fast central processor, is it unfounded to say that such a PC is a three-processor system? All three chips are sharing the bus and main memory and are running in parallel. The difference between this setup and a symmetric multiprocessing system (SMP) is that an SMP divides a single task between two or more processors, while the CPU, graphics chip, and sound chip work on different sets of data. The case made in this respect is valid, I think. If you want to put forth the argument that the motherboard chipset and memory controller are also processors, I would point out that these are logistical chips with a single task of providing low-level system communication. But consider a high-speed 3D game featuring multiplayer networking, advanced 3D rendering, and surround sound. This is a piece of software that uses multiple processors unlike any business application or Web browser.

This short overview of computer graphics was interesting, but how does the information translate to writing a game? Read on….

Graphics Fundamentals

The basis of this entire chapter can be summarized in a single word: pixel. The word *pixel* is short for "picture element," sort of the atomic element of the screen. The pixel is the

smallest unit of measurement in a video system. But like the atom you know from physics, even the smallest building block is comprised of yet smaller things. In the case of a pixel, those quantum elements of the pixel are red, green, and blue electron streams that give each pixel a specific color. This is not mere theory or analogy; each pixel is comprised of three small streams of electrons of varying shades of red, green, and blue (see Figure 3.5).

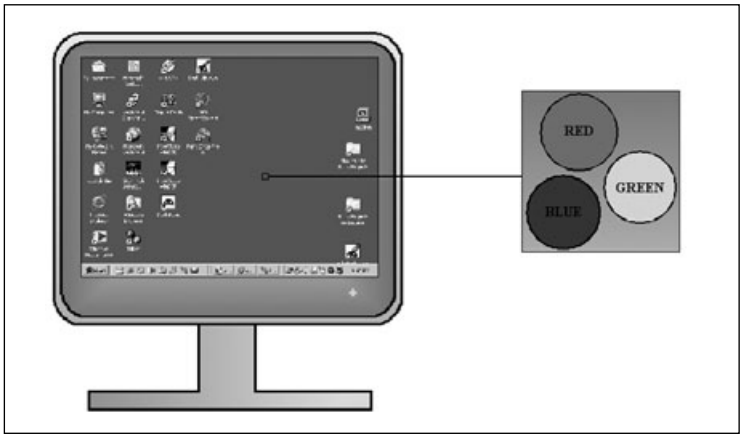


Figure 3.5 The pixel is the smallest unit of measurement in a video system.

Starting with this most basic building block, you can construct an entire game one pixel at a time (something you will do in the next chapter). Allegro creates a global screen pointer when you call `allegro_init`. This simple pointer is called `screen`, and you can pass it to all of the drawing functions in this chapter. A technique called *double-buffering* (which uses offscreen rendering for speed) works like this: Drawing routines must draw out to a memory bitmap, which is then blitted to the screen in a single function call. Until you start using a double-buffer, you'll just work with the global `screen` object.

The InitGraphics Program

As you saw in the last chapter, Allegro is useful even in a text-based console, such as the command prompt in Windows (or a shell in Linux). But there is only so much you can do with a character-based video mode. You could fire up one of the two dozen or so text adventure games from the 1970s and 1980s. (*Zork* comes to mind.) But let's get started on the really useful stuff and stop fooling around with text mode, shall we? I have written a program called *InitGraphics* that simply shows how to initialize a full-screen video mode or window of a particular resolution. Figure 3.6 shows the program running.

The first function you'll learn about in this chapter is `set_gfx_mode`, which sets the graphics mode (or what I prefer to call "video mode"). This function is really loaded, although you would not know that just from calling it. What I mean is that `set_gfx_mode` does a lot of work when called—detecting the graphics card, identifying and initializing the graphics

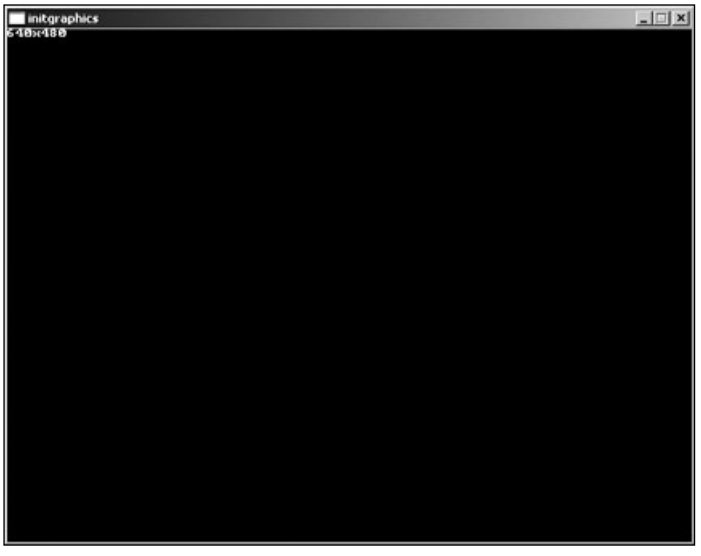


Figure 3.6 The *InitGraphics* program

system, verifying or setting the color depth, entering full-screen or windowed mode, and setting the resolution. As you can see, it does a lot of work for you! A comparable DirectX initialization is 20 to 30 lines of code. This function has the following declaration:

```
int set_gfx_mode(int card, int w, int h, int v_w, int v_h);
```

If an error occurs setting a particular video mode, `set_gfx_mode` will return a non-zero value (where a return value of zero means success) and store an error message in `allegro_error`, which you can then print out. For an example, try using an invalid resolution for a full-screen display, like this:

```
ret = set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 645, 485, 0, 0);
```

However, if you specify `GFX_AUTODETECT` and send an invalid width and height to `set_gfx_mode`, it will actually run in a window with the resolution you wanted! Running in windowed mode is a good idea when you are testing a game and you don't want it to jump into and out of full-screen mode every time you run the program.

The first parameter, `int card`, specifies the display mode (or the video card in a dual-card configuration) and will usually be `GFX_AUTODETECT`. If you want a full-screen display, you can use `GFX_AUTODETECT_FULLSCREEN`, while you can invoke a windowed display using `GFX_AUTODETECT _WINDOWED`. Both modes work equally well, but I find it easier to use windowed mode for demonstration purposes. A window is easier to handle when you are editing code, and some video cards really don't handle mode changes well. Depending on the quality of a video card, it can take several seconds to switch from full-screen back to the Windows desktop, but a windowed program does not have this problem.

The next two parameters, `int w` and `int h`, specify the desired resolution, such as 640×480, 800×600, or 1024×768. To maintain compatibility with as many systems as possible, I am using 640×480 for most of the sample programs in this book (with a few exceptions where demonstration is needed).

The final two parameters, `int v_w` and `int v_h`, specify the virtual resolution and are used to create a large virtual screen for hardware scrolling or page flipping.

After you have called `set_gfx_mode` to change the video mode, Allegro populates the variables `SCREEN_W`, `SCREEN_H`, `VIRTUAL_W`, and `VIRTUAL_H` with the appropriate values, which come in handy when you prefer not to hard-code the screen resolution in your programs.

The *InitGraphics* program source code listing follows. Several new functions in this program are included for convenience; I will go over them shortly.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void main(void)
{
    //initialize Allegro
    allegro_init();

    //initialize the keyboard
    install_keyboard();

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, makecol(255, 255, 255),
        "%dx%d", SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC]);

    //end program
    allegro_exit();
}

END_OF_MAIN();
```

In addition to the `set_gfx_mode` function, there are several more Allegro functions in this program that you probably noticed. Although they are self-explanatory, I will give you a brief overview of them.

The `allegro_message` function is handy when you want to display an error message in a pop-up dialog box (also called a *message box*). Usually you will not want to use this function in production code, although it is helpful when you are debugging (when you will want to run your program in windowed mode rather than full-screen mode). Note that some operating systems will simply output an `allegro_message` response to the console. It is fairly common to get stuck debugging a part of any game, especially when it has grown to a fair size and the source code has gotten rather long, so this function might prove handy.

You might also have noticed a variable called `allegro_error` in this program. This is one of the global variables created by Allegro when `allegro_init` is called, and it is populated with a string whenever an error occurs within Allegro. As a case in point for *not* using pop-ups, Allegro will not display any error messages. It's your job to deal with errors the way you see fit.

Another interesting function is `textprintf`, which, as you might have guessed, displays a message in any video mode. I will be going over all of the text output functions later in this chapter, but for now it is helpful to note how this one is called. Because this is one of the more complex functions, here is the declaration:

```
void textprintf(BITMAP *bmp, const FONT *f, int x, y, color,
          const char *fmt, ...);
```

The first parameter specifies the destination, which can be the physical display screen or a memory bitmap. The next parameter specifies the font to be used for output. The `x` and `y` parameters specify where the text should be drawn on the screen, while `color` denotes the color used for the text. The last parameter is a string containing the text to display along with formatting information that is comparable to the formatting in the standard `printf` function (for instance, `%s` for string, `%i` for integer, and so on).

You might have noticed a function called `makecol` within the `textprintf` code line. This function creates an RGB color using the component colors passed to it. However, Allegro also specifies 16 default colors you can use, which is a real convenience for simple text output needs. If you want to define custom colors beyond these mere 16 default colors, you can create your own colors like this:

```
#define COLOR_BROWN makecol(174,123,0)
```

This is but one out of 16 million possible colors in a 32-bit graphics system. Table 3.1 displays the colors pre-defined for your use.

The last function that you should be aware of is `allegro_exit`, which shuts down the graphics system and destroys the memory used by Allegro. In theory, the destructors will take care of removing everything from memory, but it's a good idea to call this function explicitly. One very important reason why is for the benefit of restoring the video display. (Failure to call `allegro_exit` might leave the desktop in an altered resolution or color depth depending on the graphics card being used.)

All of the functions and variables presented in this program will become familiar to you in time because they are frequently used in the example programs in this book.

Table 3.1 Standard Colors for Allegro Graphics (8-Bit Only)

Color #	Color Name
0	Black
1	Dark Blue
2	Dark Green
3	Dark Cyan
4	Dark Red
5	Dark Magenta
6	Orange
7	Gray
8	Dark Gray
9	Blue
10	Green
11	Cyan
12	Red
13	Magenta
14	Yellow
15	White

The DrawBitmap Program

Now that you have an idea of how to initialize one of the graphics modes available in Allegro, you have the ability to draw on the screen (or in the main window of your program). But before I delve into some of the graphics primitives built into Allegro, I want to show you a simple program that loads a bitmap file (the supported formats are BMP, PCX, TGA, and LBM) and draws it to the screen using a method called *bit-block transfer* (or *blit,* for short). This program will be a helpful introduction to the functions for initializing the graphics system—setting the video mode, color depth, and so on.

While I'm holding off on bitmap and sprite programming for the next two chapters, I believe you will appreciate the simplicity of this program, shown in Figure 3.7. It is always a significant first step to writing a game when you are able to load and display a bitmap image on the screen because that is the basis for sprite-based games. First, create a new project so you can get started on the first of many exciting projects in the graphical realm.

Fire up Dev-C++, open the File menu, and select New, Project. Click on the MultiMedia tab. If you have installed the Allegro DevPak as described in Chapter 2, you should see two Allegro project templates—Allegro (DLL) and Allegro (Static). Hold off on the static projects for now; you'll have plenty of time to delve into that later. For now, stick to the simple

DLL-type projects. Name this project DrawBitmap (see Figure 3.8). If you prefer, you can load the project from \sources\chapter03\DrawBitmap on the CD-ROM. After you have created the new project, you'll have a sample code listing in main.c. Delete most of that code and enter the following code in its place.

Figure 3.7 The *DrawBitmap* program

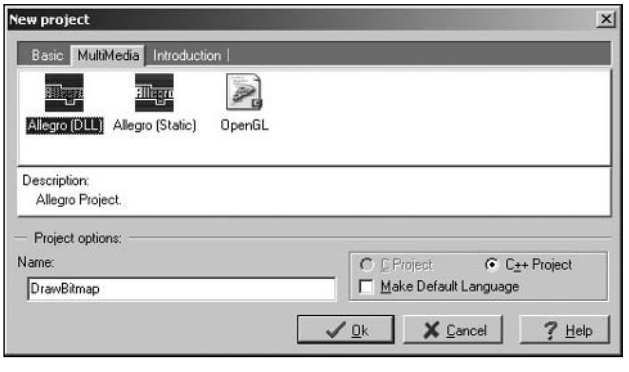


Figure 3.8 The New Project dialog box in Dev-C++

```
#include "allegro.h"

void main(void)
{
```

```
    char *filename = "allegro.pcx";
    int colordepth = 32;
    BITMAP *image;
    int ret;

    allegro_init();
    install_keyboard();

    set_color_depth(colordepth);
    ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //load the image file
    image = load_bitmap(filename, NULL);
    if (!image) {
        allegro_message("Error loading %s", filename);
        return;
    }

    //display the image
    blit(image, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

    //done drawing--delete bitmap from memory
    destroy_bitmap(image);

    //draw font with transparency
    text_mode(-1);

    //display video mode information
    textprintf(screen, font, 0, 0, makecol(255, 255, 255),
        "%dx%d %ibpp", SCREEN_W, SCREEN_H, colordepth);

    //wait for keypress
    while (!key[KEY_ESC]);

    //exit program
    allegro_exit();
}

END_OF_MAIN();
```

As you can see from the source code for *DrawBitmap*, the program loads a file called allegro.pcx. Obviously you'll need a PCX file to run this program. However, you can just as easily use a BMP, PNG, GIF, or JPG for the graphics file if you want because Allegro supports all of these formats! That alone is reason enough to use a game library like Allegro! Do you know what a pain it is to write loaders for these file formats yourself? Even if you find code on the Web somewhere, it is never quite satisfactory. Not only does Allegro support these file formats, it allows you to use them for storing sprites—and you can load different file formats all in the same program because Allegro does all the work for you. Feel free to substitute allegro.pcx with a file of your choosing; just be sure it has a resolution of 640×480! Allegro determines the file type from the extension and header information within the file. (Yeah, it's a pretty smart library.)

Drawing Graphics Primitives

While the first two programs in this chapter might have only whetted your appetite for graphics, this section will satisfy your hunger for more! Vector graphics are always fun, in my opinion, because you are able to see every pixel or line in a vector-based program. The term "vector" goes back to the early days of computer graphics, when primitive monitors were only able to display lines of varying sizes (where a vector represents a line segment from one point to another).

All of the graphics in a vector system are comprised of lines (including circles, rectangles, and arcs, which are made up of small lines). Vector displays are contrasted with bitmapped displays, in which the screen is a bitmap array (the video buffer). On the contrary, a vector system does not have a linear video buffer.

At any rate, that is what a vector system is as a useful comparison, but you have far more capabilities with Allegro. I always prefer to start at the beginning and work my way up into a subject of interest, and Allegro is definitely interesting. So I'm going to start with the vector-based graphics primitives built into Allegro and work up from there into bitmap- and sprite-based games in the next few chapters.

Drawing Pixels

The simplest graphics primitive is obviously the pixel-drawing function, and Allegro provides one:

```
void putpixel(BITMAP *bmp, int x, int y, int color);
```

Figure 3.9 shows the output of the *Pixels* program, which draws random pixels on the screen using whatever video mode and resolution you prefer.

```
#include <conio.h>
#include <stdlib.h>
```

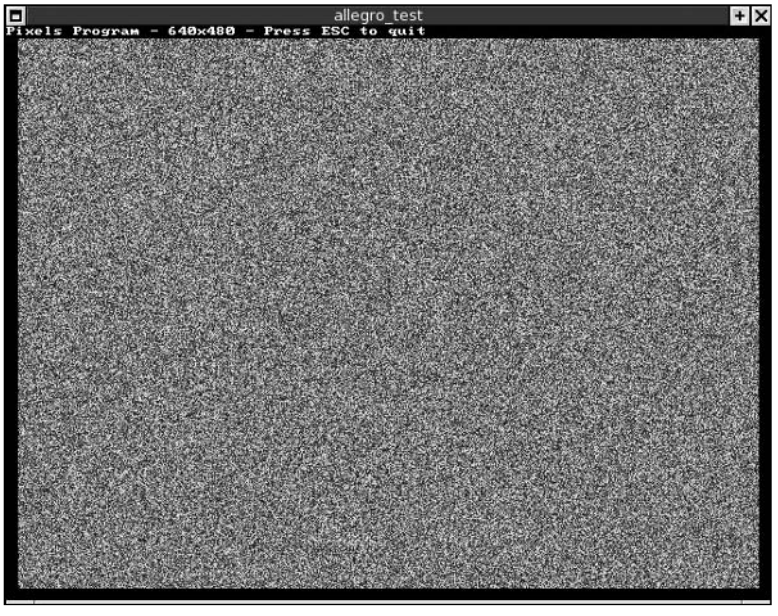


Figure 3.9 The *Pixels* program fills the screen with dots. (The Linux version is shown.)

```
#include "allegro.h"

void main(void)
{
    int x,y,x1,y1,x2,y2;
    int red, green, blue, color;

    //initialize Allegro
    allegro_init();

    //initialize the keyboard
    install_keyboard();

    //initialize the random number seed
    srand(time(NULL));

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
```

```
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Pixels Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x = 10 + rand() % (SCREEN_W-20);
        y = 10 + rand() % (SCREEN_H-20);

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the pixel
        putpixel(screen, x, y, color);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

This program should be clear to you, although it uses a C function called `srand` to initialize the random-number seed. This program performs a `while` loop continually until the ESC key is pressed, during which time a pixel of random color and location is drawn using the `putpixel` function.

Drawing Lines and Rectangles

The next step up from the lowly pixel is the line, and Allegro provides several line-drawing functions. To keep things as efficient as possible, Allegro divides line drawing among three functions—one for horizontal lines, one for vertical lines, and a third for every other type of line. Drawing horizontal and vertical lines can be an extremely optimized process using a simple high-speed memory copy, but non-aligned lines must be drawn using an algorithm to fill in the pixels between two points specified for the line (see Figure 3.10).

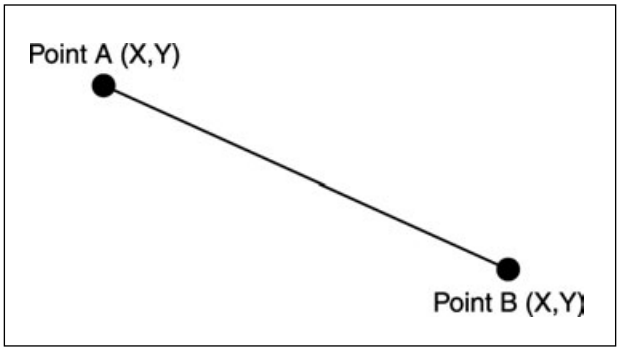


Figure 3.10 A line is comprised of pixels filled in between point A and point B.

Horizontal Lines

The horizontal line-drawing function is called `hline`:

```
void hline(BITMAP *bmp, int x1, int y, int x2, int color);
```

Because this is your first function for drawing lines, allow me to elaborate. The first parameter, `BITMAP *bmp`, is the destination bitmap for the line, which can be `screen` if you want to draw directly to the screen. The next three paramters, `int x1`, `int y`, and `int x2`, specify the two points on the single horizontal Y-axis where the line should be drawn. The *HLines* program (shown in Figure 3.11) demonstrates how to use this function.

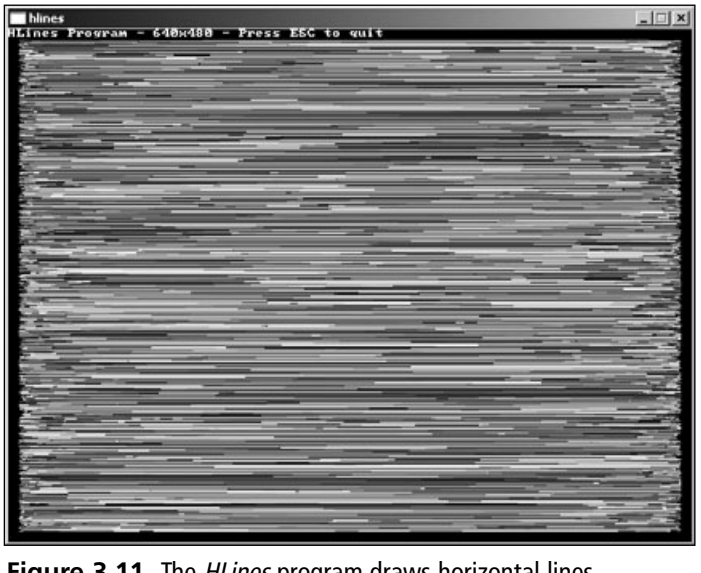


Figure 3.11 The *HLines* program draws horizontal lines.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void main(void)
{
    int x,y,x1,y1,x2,y2;
    int red,green,blue,color;

    //initialize Allegro
    allegro_init();

    //initialize the keyboard
    install_keyboard();

    //initialize random seed
    srand(time(NULL));

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "HLines Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x1 = 10 + rand() % (SCREEN_W-20);
        y = 10 + rand() % (SCREEN_H-20);
        x2 = 10 + rand() % (SCREEN_W-20);

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
```

```
        color = makecol(red,green,blue);

        //draw the horizontal line
        hline(screen, x1,y,x2,color);
    }

    //end program
    allegro_exit();
}

END_OF_MAIN();
```

You have probably noticed that the *HLines* program is very similar to the *Pixels* program, with only a few lines that differ inside the `while` loop. I'll just show the differences from this point forward, rather than listing the entire source code for each program, because in most cases you simply need to replace a few lines inside `main`. It is pretty obvious that just a few lines inside the `while` loop need to be changed. The programs are available on the CD-ROM in complete form, but I will provide only partial listings where such changes are needed to demonstrate each of these graphics primitives.

Vertical Lines

Vertical lines are drawn with the `vline` function:

```
void vline(BITMAP *bmp, int x, int y1, int y2, int color);
```

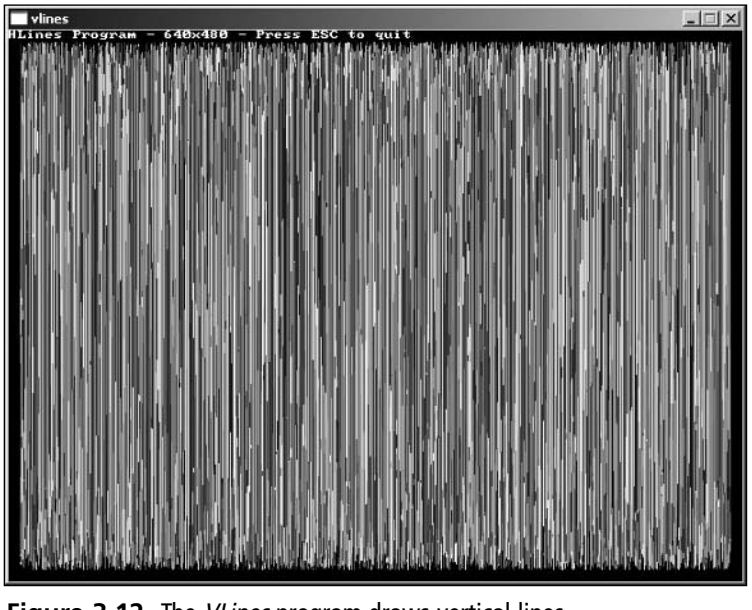


Figure 3.12 The *VLines* program draws vertical lines.

The *VLines* program (see Figure 3.12) is the same as the *HLines* program except for a single function call inside the `while` loop. Also note that this program uses a single X variable and two Y variables, `y1` and `y2`. Here is the listing:

```
//display screen resolution
textprintf(screen, font, 0, 0, 15,
    "VLines Program - %dx%d - Press ESC to quit",
    SCREEN_W, SCREEN_H);

//wait for keypress
while(!key[KEY_ESC])
{
    //set a random location
    x = 10 + rand() % (SCREEN_W-20);
    y1 = 10 + rand() % (SCREEN_H-20);
    y2 = 10 + rand() % (SCREEN_H-20);

    //set a random color
    red = rand() % 255;
    green = rand() % 255;
    blue = rand() % 255;
    color = makecol(red,green,blue);

    //draw the vertical line
    vline(screen,x,y1,y2,color);
}
```

Regular Lines

The special-case lines functions for drawing horizontal and vertical lines are not used often. The following `line` function will simply call `hline` or `vline` if the slope of the line is perfectly horizontal or vertical:

```
void line(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
```

The *Lines* program uses two complete sets of points—(`x1`,`y1`) and (`x2`,`y2`)—to draw an arbitrary line on the screen (see Figure 3.13).

```
//display screen resolution
textprintf(screen, font, 0, 0, 15,
    "Lines Program - %dx%d - Press ESC to quit",
    SCREEN_W, SCREEN_H);

//wait for keypress
while(!key[KEY_ESC])
```

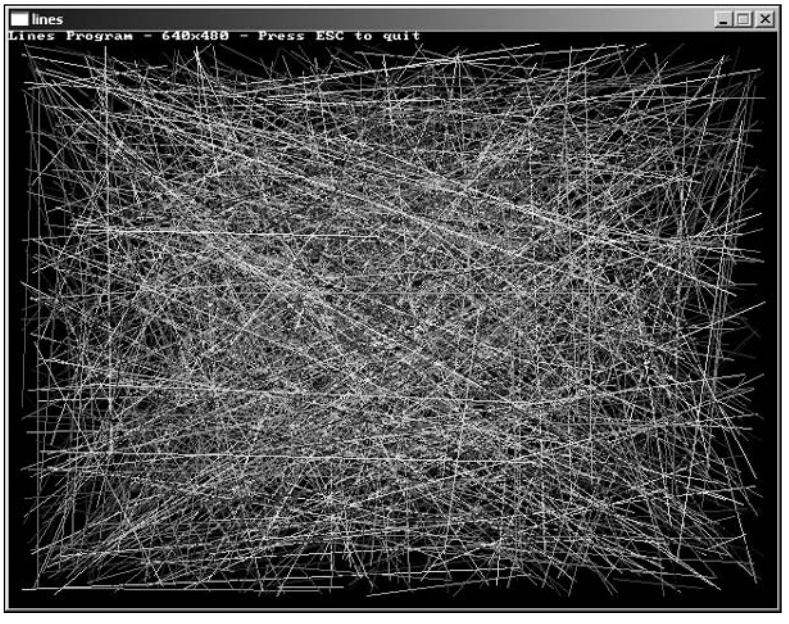


Figure 3.13 The *Lines* program draws random lines on the screen.

```
{
    //set a random location
    x1 = 10 + rand() % (SCREEN_W-20);
    y1 = 10 + rand() % (SCREEN_H-20);
    x2 = 10 + rand() % (SCREEN_W-20);
    y2 = 10 + rand() % (SCREEN_H-20);

    //set a random color
    red = rand() % 255;
    green = rand() % 255;
    blue = rand() % 255;
    color = makecol(red,green,blue);

    //draw the line
    line(screen, x1,y1,x2,y2,color);
}
```

Rectangles

Yet again there is another logical step forward in geometry that is mimicked by a primitive graphics function. While a single pixel might be thought of as a geometric point with no mass, a line is a one-dimensional object that theoretically goes off in two directions

toward infinity. Fortunately for us, computer graphics engineers are not as abstract as mathematicians. The next logical step is a two-dimensional object containing points in both the X-axis and the Y-axis. Although a triangle would be the next best thing, I believe the rectangle is easier to deal with at this stage because triangles carry with them the connotation of the mighty polygon, and we aren't quite there yet. Here is the `rect` function:

```
void rect(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
```

As you might have guessed, a rectangle is comprised strictly of two horizontal and two vertical lines; therefore, the `rect` function simply calls `hline` and `vline` to render its shape (see Figure 3.14).

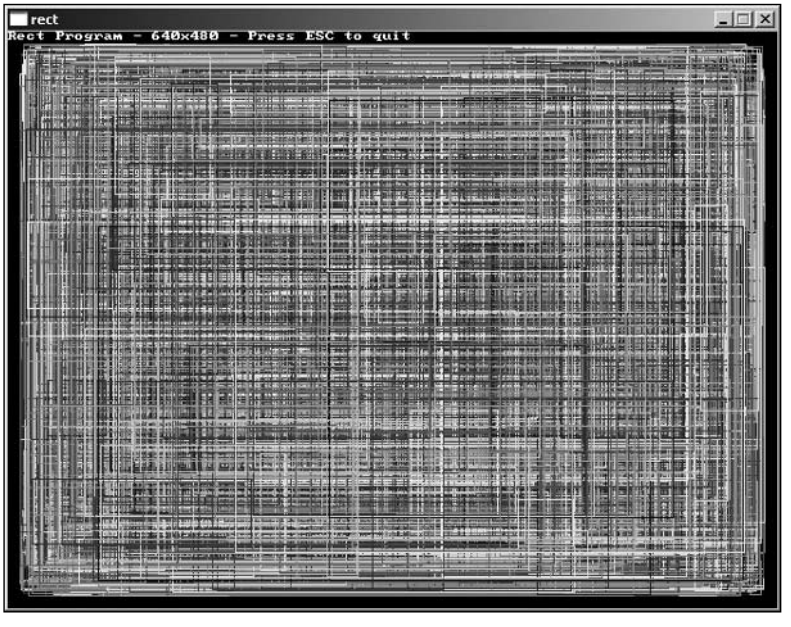


Figure 3.14 The *Rect* program draws random rectangles.

```
//display screen resolution
textprintf(screen, font, 0, 0, 15,
    "Rect Program - %dx%d - Press ESC to quit",
    SCREEN_W, SCREEN_H);

//wait for keypress
while(!key[KEY_ESC])
{
    //set a random location
    x1 = 10 + rand() % (SCREEN_W-20);
    y1 = 10 + rand() % (SCREEN_H-20);
```

```
        x2 = 10 + rand() % (SCREEN_W-20);
        y2 = 10 + rand() % (SCREEN_H-20);

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the rectangle
        rect(screen,x1,y1,x2,y2,color);
    }
```

Filled Rectangles

Outlined rectangles are boring, if you ask me. They are almost too thin to be noticed when drawn. On the other hand, a true rectangle is filled in with a specific color! That is where the `rectfill` function comes in handy:

```
void rectfill(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
```

This function draws a filled rectangle, but one that otherwise has the exact same parameters as `rect`. Figure 3.15 shows the output from the *RectFill* program.

Figure 3.15 The *RectFill* program draws filled rectangles.

```
//display screen resolution
 textprintf(screen, font, 0, 0, 15,
     "Rect Program - %dx%d - Press ESC to quit",
     SCREEN_W, SCREEN_H);

 //wait for keypress
 while(!key[KEY_ESC])
 {
     //set a random location
     x1 = 10 + rand() % (SCREEN_W-20);
     y1 = 10 + rand() % (SCREEN_H-20);
     x2 = 10 + rand() % (SCREEN_W-20);
     y2 = 10 + rand() % (SCREEN_H-20);

     //set a random color
     red = rand() % 255;
     green = rand() % 255;
     blue = rand() % 255;
     color = makecol(red,green,blue);

     //draw the filled rectangle
     rectfill(screen,x1,y1,x2,y2,color);
 }
```

The Line-Drawing Callback Function

Allegro provides a really fascinating feature in that it will draw an abstract line by firing off a call to a callback function of your making (in which, presumably, you would want to draw a pixel at the specified (x,y) location, although it's up to you to do what you will with the coordinate). To use the callback, you must call the `do_line` function, which looks like this:

```
void do_line(BITMAP *bmp, int x1, y1, x2, y2, int d, void (*proc))
```

The callback function has this format:

```
void doline_callback(BITMAP *bmp, int x, int y, int d)
```

To use the callback, you want to call the `do_line` function as you would call the normal `line` function, with the addition of the callback pointer as the last parameter. To fully demonstrate how useful this can be, I wrote a short program that draws random lines on the screen. But before drawing each pixel of the line, a check is performed on the new position to determine whether a pixel is already present. This indicates an intersection or collision. When this occurs, the line is ended and a small circle is drawn to indicate the intersection. The result is shown in Figure 3.16.

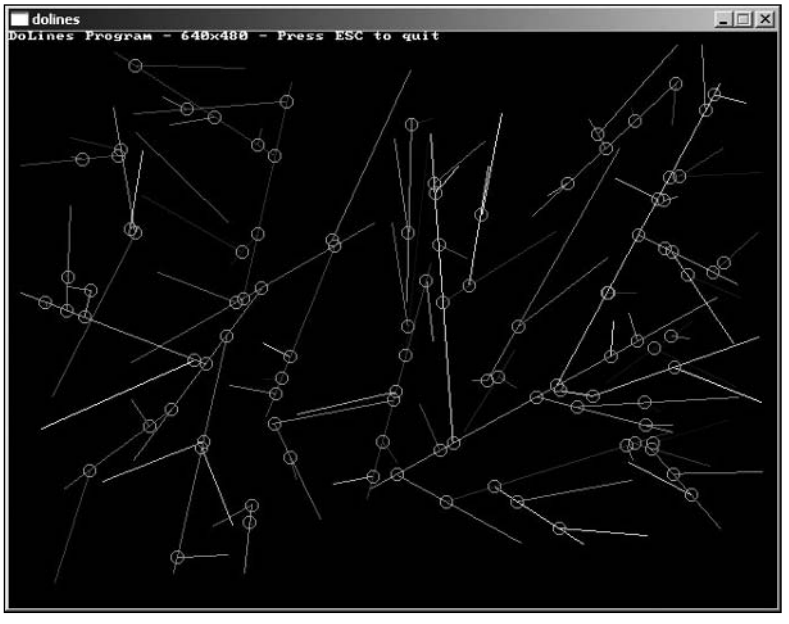


Figure 3.16 The *DoLines* program shows how to use the line-drawing callback function.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

int stop = 0;

//doline is the callback function for do_line
void doline(BITMAP *bmp, int x, int y, int color)
{
    if (!stop)
    {
        if (getpixel(bmp,x,y) == 0)
        {
            putpixel(bmp, x, y, color);
            rest(5);
        }
        else
        {
            stop = 1;
            circle(bmp, x, y, 5, 7);
```

```
        }
    }
}

void main(void)
{
    int x1,y1,x2,y2;
    int red,green,blue,color;
    long n;

    //initialize Allegro
    allegro_init();

    install_timer();
    srand(time(NULL));

    //initialize the keyboard
    install_keyboard();

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "DoLines Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x1 = 10 + rand() % (SCREEN_W-20);
        y1 = 10 + rand() % (SCREEN_H-20);
        x2 = 10 + rand() % (SCREEN_W-20);
        y2 = 10 + rand() % (SCREEN_H-20);

        //set a random color
        red = rand() % 255;
```

```
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the line using the callback function
        stop = 0;
        do_line(screen,x1,y1,x2,y2,color,*doline);

        rest(200);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Drawing Circles and Ellipses

Allegro also provides functions for drawing circles and ellipses, as you will see. The circle-drawing function is called `circle`, surprisingly enough. This function takes a set of parameters very similar to those you have seen already—the destination bitmap, x, y, the radius, and the color.

Circles

The `circle` function has this declaration:

```
void circle(BITMAP *bmp, int x, int y, int radius, int color);
```

To demonstrate, the *Circles* program draws random circles on the screen, as shown in Figure 3.17.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void main(void)
{
    int x,y,radius;
    int red,green,blue,color;

    //initialize some stuff
    allegro_init();
    install_keyboard();
```

```
    install_timer();
    srand(time(NULL));

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Circles Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x = 30 + rand() % (SCREEN_W-60);
        y = 30 + rand() % (SCREEN_H-60);
        radius = rand() % 30;

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the pixel
        circle(screen, x, y, radius, color);

        rest(25);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

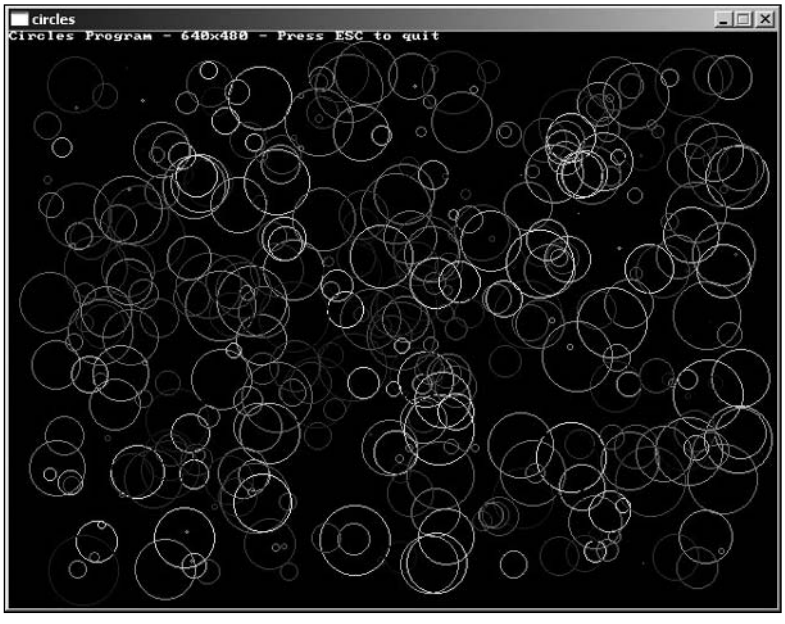


Figure 3.17 The *Circles* program draws random circles on the screen.

Filled Circles

The hollow circle function is interesting, but really seeing the full effect of circles requires the `circlefill` function:

```
void circlefill(BITMAP *bmp, int x, int y, int radius, int color);
```

The following program (shown in Figure 3.18) demonstrates the solid-filled circle function.

```
//display screen resolution
textprintf(screen, font, 0, 0, 15,
    "CircleFill Program - %dx%d - Press ESC to quit",
    SCREEN_W, SCREEN_H);

//wait for keypress
while(!key[KEY_ESC])
{
    //set a random location
    x = 30 + rand() % (SCREEN_W-60);
    y = 30 + rand() % (SCREEN_H-60);
    radius = rand() % 30;
```

```
        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the filled circle
        circlefill(screen, x, y, radius, color);

        rest(25);
    }
```

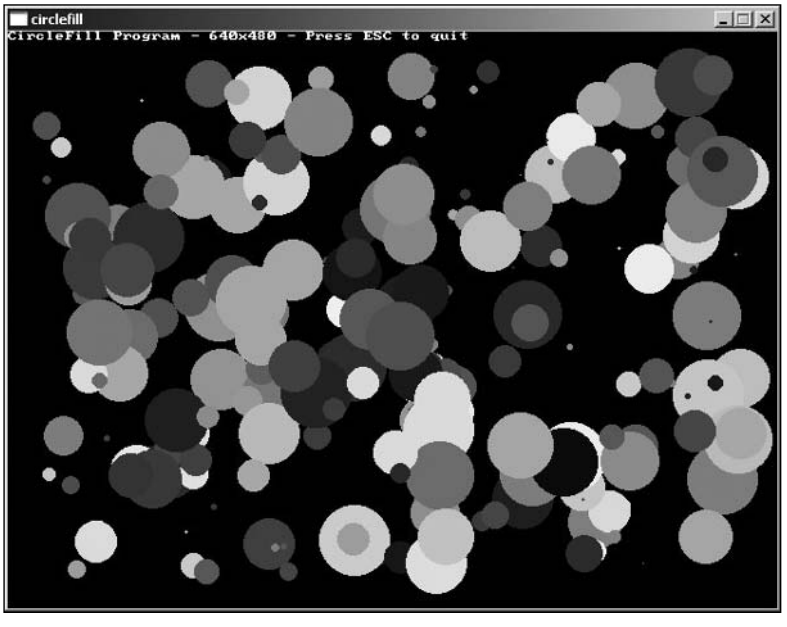


Figure 3.18 The *CircleFill* program draws filled circles.

Ellipses

The `ellipse` function is similar to the `circle` function, although the radius is divided into two parameters—one for the horizontal and another for the vertical—as indicated:

```
void ellipse(BITMAP *bmp, int x, int y, int rx, int ry, int color);
```

The *Ellipses* program draws random ellipses on the screen using two parameters—`radiusx` and `radiusy`.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void main(void)
{
    int x,y,radiusx,radiusy;
    int red,green,blue,color;

    //initialize everything
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Ellipses Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x = 30 + rand() % (SCREEN_W-60);
        y = 30 + rand() % (SCREEN_H-60);
        radiusx = rand() % 30;
        radiusy = rand() % 30;

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);
```

```
        //draw the ellipse
        ellipse(screen, x, y, radiusx, radiusy, color);

        rest(25);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Filled Ellipses

You can draw filled ellipses using the `ellipsefill` function, which takes the same parameters as the `ellipse` function but simply renders each ellipse with a solid fill color:

```
void ellipsefill(BITMAP *bmp, int x, int y, int rx, int ry, int color);
```

Figure 3.19 shows the output from the *EllipseFill* program.

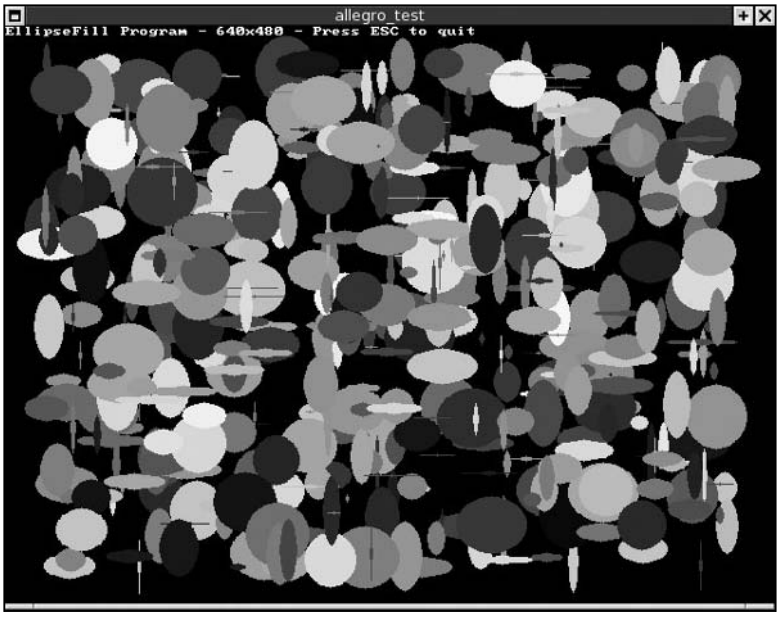


Figure 3.19 The *EllipseFill* program draws filled ellipses. (The Linux version is shown.)

```
    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "EllipseFill Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);
```

```
    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x = 30 + rand() % (SCREEN_W-60);
        y = 30 + rand() % (SCREEN_H-60);
        radiusx = rand() % 30;
        radiusy = rand() % 30;

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the ellipse
        ellipsefill(screen, x, y, radiusx, radiusy, color);

        sleep(25);
    }
```

Circle Drawing Callback Function

Surprisingly enough, Allegro provides a circle-drawing callback function just as it did with the line callback function. The only difference is, this one uses the `do_circle` function:

```
void do_circle(BITMAP *bmp, int x, int y, int radius, int d);
```

To use `do_circle`, you must declare a callback function with the format `void docircle(BITMAP *bmp, int x, int y, int d)` and pass a pointer to this function to `do_circle`, as the following sample program demonstrates.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void docircle(BITMAP *bmp, int x, int y, int color)
{
    putpixel(bmp, x, y, color);
    putpixel(bmp, x+1, y+1, color);
    rest(1);
}

void main(void)
{
```

```
    int x,y,radius;
    int red,green,blue,color;

    //initialize Allegro
    allegro_init();

    //initialize the keyboard
    install_keyboard();
    install_timer();

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "DoCircles Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x = 40 + rand() % (SCREEN_W-80);
        y = 40 + rand() % (SCREEN_H-80);
        radius = rand() % 40;

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the circle
        do_circle(screen, x, y, radius, color, *docircle);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Drawing Splines, Triangles, and Polygons

I have now covered all of the basic graphics primitives built into Allegro except for three, which might be thought of as the most important ones, at least where a game is involved. The `spline` function is valuable for creating dynamic trajectories for objects in a game that needs various curving paths. Triangles and other types of polygons are the basis for 3D graphics, so I will show you how to draw them.

Splines

The `spline` function draws a set of curves based on a set of four input points stored in an array. The function calculates a smooth curve from the first set of points, through the second and third, toward the fourth point:

```
void spline(BITMAP *bmp, const int points[8], int color);
```

The *Splines* program draws an animated spline based on shifting points, as shown in Figure 3.20.

Figure 3.20 The *Splines* program draws an animated spline curve. (The Linux version is shown.)

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"
```

```
void main(void)
{
    int red,green,blue,color;

    //initialize Allegro
    allegro_init();
    install_keyboard();
    install_timer();

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Splines Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    int points[8] = {0,240,300,0,200,0,639,240};
    int y1 = 0;
    int y2 = SCREEN_H;
    int dir1 = 10;
    int dir2 = -10;

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //modify the first spline point
        y1 += dir1;
        if (y1 > SCREEN_H)
        {
            dir1 = -10;
        }
        if (y1 < 0)
            dir1 = 10;
        points[3] = y1;
```

```
        //modify the second spline point
        y2 += dir2;
        if (y2++ > SCREEN_H)
        {
            dir2 = -10;
        }
        if (y2 < 0)
            dir2 = 10;
        points[5] = y2;

        //draw the spline, pause, then erase it
        spline(screen, points, 15);
        rest(30);
        spline(screen, points, 0);

    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Triangles

You can draw triangles using the `triangle` function, which takes three `(x,y)` points and a `color` parameter:

```
void triangle(BITMAP *bmp, int x1, y1, x2, y2, x3, y3, int color);
```

The *Triangles* program (shown in Figure 3.21) draws random triangles on the screen.

```
#include "allegro.h"

void main(void)
{
    int x1,y1,x2,y2,x3,y3;
    int red,green,blue,color;

    //initialize Allegro
    allegro_init();

    //initialize the keyboard
    install_keyboard();
    install_timer();
```

```
    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Triangles Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        x1 = 10 + rand() % (SCREEN_W-20);
        y1 = 10 + rand() % (SCREEN_H-20);
        x2 = 10 + rand() % (SCREEN_W-20);
        y2 = 10 + rand() % (SCREEN_H-20);
        x3 = 10 + rand() % (SCREEN_W-20);
        y3 = 10 + rand() % (SCREEN_H-20);

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the triangle
        triangle(screen,x1,y1,x2,y2,x3,y3,color);

        rest(100);
    }

    //end program
    allegro_exit();
}

END_OF_MAIN();
```

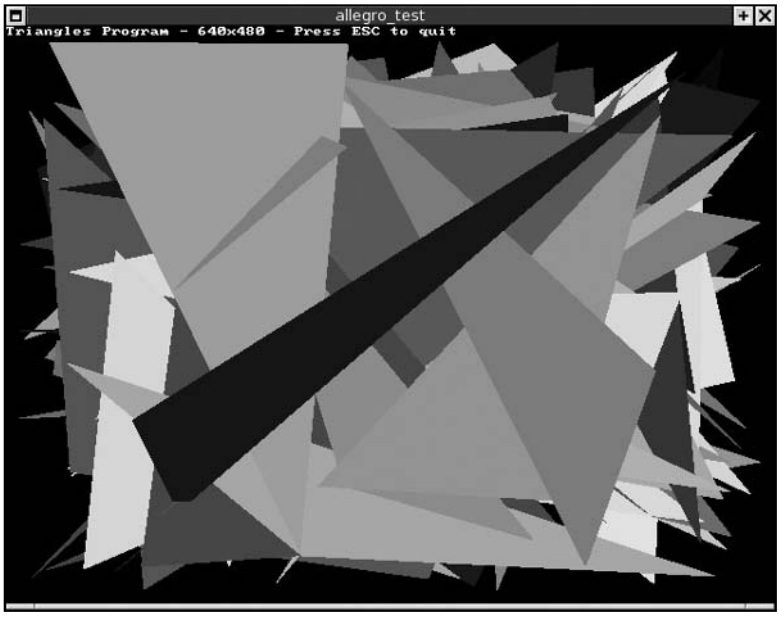


Figure 3.21 The *Triangles* program draws random triangles on the screen. (The Linux version is shown.)

Polygons

You have already seen polygons in action with the *Triangles* program, because any geometric shape with three or more points comprises a polygon. To draw polygons in Allegro, you use the `polygon` function with a pointer to an array of points:

```
void polygon(BITMAP *bmp, int vertices, const int *points, int color);
```

In most cases you will want to simply use the `triangle` function, but in unusual cases when you need to draw polygons with more than three points, this function can be helpful (although it is more difficult to set up because the points array must be set up prior to calling the polygon function). The best way to demonstrate this function is with a sample program that sets up the points array and calls the `polygon` function (see Figure 3.22).

There is more to the subject of polygon rendering than I have time for in this chapter. Rest assured, you will have several more opportunities in later chapters to exercise the polygon functions built into Allegro.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"
```

```
void main(void)
{
    int vertices[8];
    int red,green,blue,color;

    //initialize everything
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //display screen resolution
    textprintf(screen, font, 0, 0, 15,
        "Polygons Program - %dx%d - Press ESC to quit",
        SCREEN_W, SCREEN_H);

    //wait for keypress
    while(!key[KEY_ESC])
    {
        //set a random location
        vertices[0] = 10 + rand() % (SCREEN_W-20);
        vertices[1] = 10 + rand() % (SCREEN_H-20);
        vertices[2] = vertices[0] + (rand() % 30)+50;
        vertices[3] = vertices[1] + (rand() % 30)+50;
        vertices[4] = vertices[2] + (rand() % 30)-100;
        vertices[5] = vertices[3] + (rand() % 30)+50;
        vertices[6] = vertices[4] + (rand() % 30);
        vertices[7] = vertices[5] + (rand() % 30)-100;

        //set a random color
        red = rand() % 255;
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);
```

```
            //draw the polygon
            polygon(screen,4,vertices,color);

            rest(50);
        }
        //end program
        allegro_exit();
}
END_OF_MAIN();
```

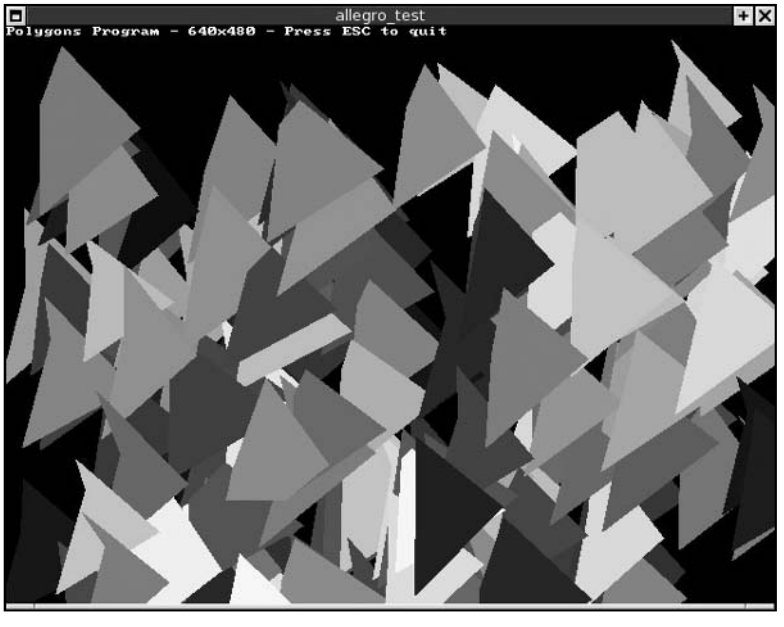


Figure 3.22 The *Polygons* program draws random polygons on the screen. (The Linux version is shown.)

Filling in Regions

The last function I want to introduce to you in this chapter is `floodfill`, which fills in a region on the destination bitmap (which can be the screen) with the color of your choice:

```
void floodfill(BITMAP *bmp, int x, int y, int color);
```

To demonstrate, the *FloodFill* program draws a circle on the screen and fills it in using the `floodfill` function while the "ball" is moving around on the screen. I will be the first to admit that this program could have simply called the `circlefill` function (which is very likely faster, too), but the object of this program is to demonstrate `floodfill` with a basic circle shape that has historically been difficult to fill efficiently (see Figure 3.23).

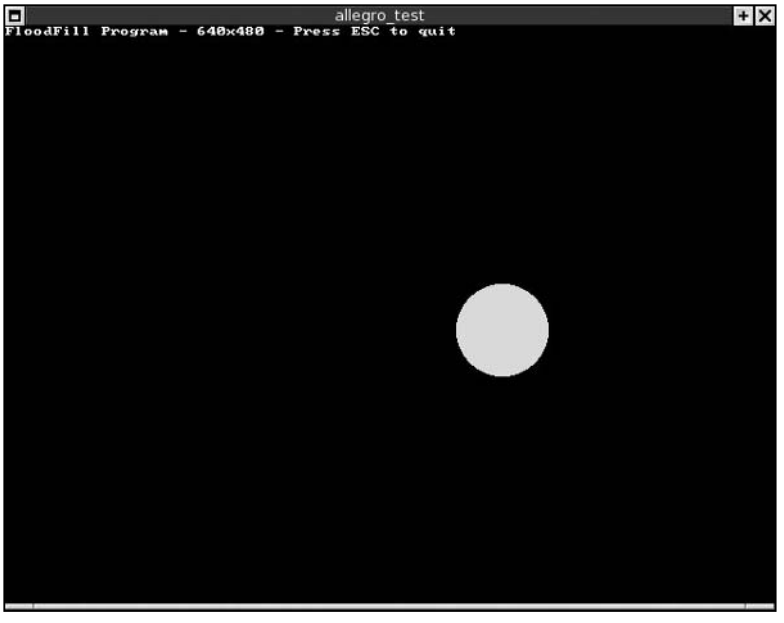


Figure 3.23 The *FloodFill* program moves a filled circle around on the screen. (The Linux version is shown.)

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

void main(void)
{
    int x = 100, y = 100;
    int xdir = 10, ydir = 10;
    int red,green,blue,color;
    int radius = 50;

    //initialize some things
    allegro_init();
    install_keyboard();
    install_timer();

    //initialize video mode to 640x480
    int ret = set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
```

```
}

//display screen resolution
textprintf(screen, font, 0, 0, 15,
    "FloodFill Program - %dx%d - Press ESC to quit",
    SCREEN_W, SCREEN_H);

//wait for keypress
while(!key[KEY_ESC])
{
    //update the x position, keep within screen
    x += xdir;
    if (x > SCREEN_W-radius)
    {
        xdir = -10;
        radius = 10 + rand() % 40;
        x = SCREEN_W-radius;
    }
    if (x < radius)
    {
        xdir = 10;
        radius = 10 + rand() % 40;
        x = radius;
    }

    //update the y position, keep within screen
    y += ydir;
    if (y > SCREEN_H-radius)
    {
        ydir = -10;
        radius = 10 + rand() % 40;
        y = SCREEN_H-radius;
    }
    if (y < radius+20)
    {
        ydir = 10;
        radius = 10 + rand() % 40;
        y = radius+20;
    }

    //set a random color
    red = rand() % 255;
```

```
        green = rand() % 255;
        blue = rand() % 255;
        color = makecol(red,green,blue);

        //draw the circle, pause, then erase it
        circle(screen, x, y, radius, color);
        floodfill(screen, x, y, color);
        rest(20);
        rectfill(screen, x-radius, y-radius, x+radius, y+radius, 0);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Printing Text on the Screen

Allegro provides numerous useful text output functions for drawing on a console or graphical display. Allegro's text functions support plug-in fonts that you can create with a utility bundled with Allegro, but I'll reserve that discussion for later. For now I just want to give you a heads-up on the basic text output functions included with Allegro (some of which you have already used).

Constant Text Output

There are four primary text output functions in Allegro. The `text_mode` function sets text output to draw with an opaque or transparent background. Passing a value of –1 will set the background to transparent, while passing any other value will set the background to a specific color. Here is what the function looks like:

```
int text_mode(int mode);
```

The `textout` function is the basic text output function for Allegro. It has the syntax:

```
void textout(BITMAP *bmp, const FONT *f, const char *s, int x, y, int color);
```

The `BITMAP *bmp` parameter specifies the destination bitmap. (You can use `screen` to output directly to the screen.) `FONT *f` specifies the font, which is just `font` if you are using the default font. `const char *s` is the text to display, `int x, y` is the position on the screen, and `int color` specifies the color of the font to use. (Passing –1 will use the colors built into any custom font.) Here is an example usage for `textout`:

```
textout(screen, font, "Hello World!", 1, 1, 10);
```

This line draws directly on the screen using the default font at the position (1,1), using the color 10 (which can also be a custom color with `makecol`).

The other three text output functions are based on `textout` but provide justification. The `textout_centre` function has the same parameter list as `textout`, but the position is based on the center of the text rather than at the left.

```
void textout_centre(BITMAP *bmp, const FONT *f, const char *s,
        int x, y, color);
```

The `textout_right` function is also similar to `textout`, but the text position (x,y) specifies the right edge of the text rather than the left or center.

```
void textout_right(BITMAP *bmp, const FONT *f, const char *s,
        int x, y, color);
```

A slightly different take on the matter of text output is `textout_justify`, which includes two X coordinates—one for the left edge of the text and one for the right edge—along with the Y position. In effect, this function tries to draw the text between the two points. You want to set the `diff` parameter to a fairly high value for justification to work; otherwise, it is automatically left-justified. This really is more useful when you are using custom fonts.

```
void textout_justify(BITMAP *bmp, const FONT *f, const char *s,
        int x1, int x2, int y, int diff, int color);
```

Variable Text Output

Allegro provides several very useful text output functions that mimic the standard C `printf` function, providing the capability of formatting the text and displaying variables. The base function is `textprintf`, and it looks like this:

```
void textprintf(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

The syntax for `textprintf` is slightly different than the syntax for the `textout` functions. As you can see, `textprintf` has the character string passed as the last parameter, with support for numerous additional parameters. If you are familiar with `printf` (and you certainly should be if you call yourself a C programmer!), then you should feel right at home with `textprintf` because it supports the usual `%i` (integer), `%f` (float), `%s` (string), and other formatting elements. Here is an example:

```
float ver = 4.9;
textprintf(screen, font, 0, 100, 12, "Version %.2f", ver);
```

This code displays:

```
Version 4.90
```

There are three additional functions that share functionality with `textprintf`. The `textprintf_centre` produces the same output as `textprintf`, but the (x,y) position is based on the center of the text output (comparable to `textout_centre`). Here is the syntax:

```
void textprintf_centre(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

As you might have guessed, there is also a `textprintf_right`, which looks like this:

```
void textprintf_right(BITMAP *bmp, const FONT *f, int x, y, color,
        const char *fmt, ...);
```

Likewise, `textprintf_justify` mimics the functionality of `textout_justify` but adds the formatting capabilities. Here is the function:

```
void textprintf_justify(BITMAP *bmp, const FONT *f, int x1, int x2,
        int y, int diff, int color, const char *fmt, ...);
```

Testing Text Output

To put these functions to use, let's write a short demonstration program (see Figure 3.24). Open your favorite IDE (I am using Dev-C++ in Windows and KDevelop in Linux) and create a new project called TextOutput. In Dev-C++, you can click on the MultiMedia tab in the New Project dialog box and choose Allegro (DLL) to configure the project automatically. In KDevelop and other IDEs, you'll want to add a reference "-lalleg" to the linker options to incorporate the Allegro library file.

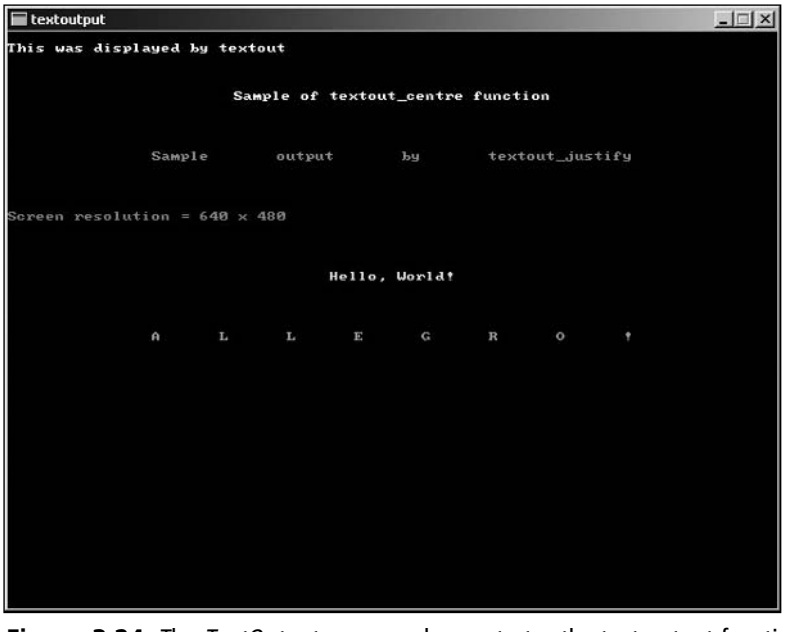


Figure 3.24 The *TextOutput* program demonstrates the text output functions of Allegro.

```
#include <allegro.h>

int main()
{
    //initialize Allegro
    allegro_init();
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();
    text_mode(-1);

    //test the text output functions
    textout(screen, font, "This was displayed by textout", 0, 10, 15);

    textout_centre(screen, font, "Sample of textout_centre function",
        SCREEN_W/2, 50, 14);

    textout_justify(screen, font, "Sample output by textout_justify",
        SCREEN_W/2 - 200, SCREEN_W/2 + 200, 100, 200, 13);

    textprintf(screen, font, 0, 150, 12, "Screen resolution = %i x %i",
        SCREEN_W, SCREEN_H);

    textprintf_centre(screen, font, SCREEN_W/2, 200, 10,
        "%s, %s!", "Hello", "World");

    textprintf_justify(screen, font, SCREEN_W/2 - 200,
        SCREEN_W/2 + 200, 250, 400, 7, "A L L E G R O !");


    //main loop
    while(! key[KEY_ESC]) { }

    allegro_exit();
    return 0;
}
END_OF_MAIN();
```

Summary

This chapter has been a romp through the basic graphics functions built into Allegro. You learned to draw pixels, lines, circles, ellipses, and other geometric shapes in various colors, with wireframe and solid filled color. I also covered text output in Allegro, and you learned about the different text functions and how to use them. This chapter included many sample programs to demonstrate all of the new functionality presented.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the term used to describe line-based graphics?
   A. Vector
   B. Bitmap
   C. Polygon
   D. Pixel

2. What does CRT stand for?
   A. Captain Ron Teague
   B. Corporate Resource Training
   C. Cathode Ray Tube
   D. Common Relativistic Torch

3. What describes a function that draws a simple geometric shape, such as a point, line, rectangle, or circle?
   A. `putpixel`
   B. `Graphics Primitive`
   C. `triangle`
   D. `polygon`

4. How many polygons does the typical 3D accelerator chip process at a time?
   A. 16
   B. 8
   C. 1
   D. 256

5. What is comprised of three small streams of electrons of varying shades of red, green, and blue?
   A. Superstring
   B. Quantum particle
   C. Electron gun
   D. Pixel

6. What function is used to create a custom 24- or 32-bit color?
   A. `makecol`
   B. `rgb`
   C. `color`
   D. `truecolor`

7. What function is used to draw filled rectangles?
   A. `fill_rect`
   B. `fillrect`
   C. `filledrectangle`
   D. `rectfill`

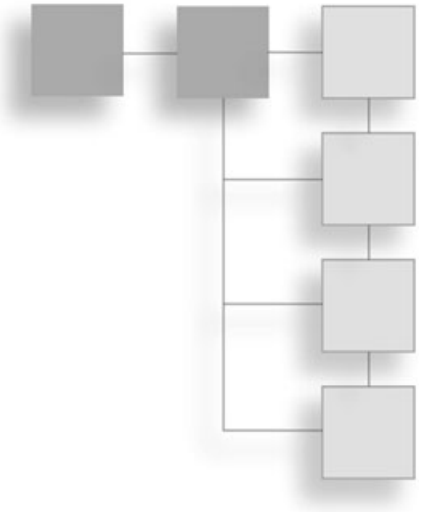
8. Which of the following is the correct definition of the circle function?
   A. `void circle(BITMAP *bmp, int x, int y, int radius, int color);`
   B. `void draw_circle(BITMAP *bmp, int x, int y, int radius);`
   C. `int circle(BITMAP *bmp, int y, int x, int radius, int color);`
   D. `bool circle(BITMAP *bmp, int x, int y, int color);`

9. What function draws a set of curves based on a set of four input points stored in an array?
   A. `jagged`
   B. `draw_curves`
   C. `spline`
   D. `polygon`

10. Which text output function draws a formatted string with justification?
    A. `textout_justify`
    B. `textprintf_right`
    C. `textout_centre`
    D. `textprintf_justify`

This page intentionally left blank

CHAPTER 4



Writing Your First Allegro Game

This chapter forges ahead with a lot of things I haven't discussed yet, such as collision detection and keyboard input, but the *Tank War* game that is created in this chapter will help you absorb all the information presented thus far. You'll see how you can use the graphics primitives you learned in Chapter 3 to create a complete game with support for two players. You will learn how to draw and move a tank around on the screen using nothing but simple pixel and rectangle drawing functions. You will learn how to look at the video screen to determine when a projectile strikes a tank or another object, how to read the keyboard, and how to process a game loop. The goal of this chapter is to show you that you can create an entire game using the meager resources provided thus far (in the form of the Allegro functions you have already learned) and to introduce some new functionality that will be covered in more detail in later chapters.

Here is a breakdown of the major topics in this chapter:

- Creating the tanks
- Firing weapons
- Moving the tanks
- Detecting collisions
- Understanding the complete source code

Tank War

If this is your first foray into game programming, then *Tank War* is likely your very first game! There is always a lot of joy involved in seeing your first game running on the screen. In the mid-1980s I subscribed to several of the popular computer magazines, such as *Family Computing* and *Compute!*, which provided small program listings in the BASIC

language, most often games. I can still remember some of the games I painstakingly typed in from the magazine using Microsoft GW-BASIC on my old Tandy 1000. The games never ran on the first try! I would often miss entire lines of code, even with the benefit of line numbers in the old style of BASIC.

Today there are fantastic development tools that quite often cost nothing and yet incorporate some of the most advanced compiler technology available. The Free Software Foundation (http://www.fsf.org) has done the world a wonderful service by inspiring and funding the development of free software. Perhaps the most significant contribution by the FSF is the GNU Compiler Collection, fondly known as GCC. Oddly enough, this very same compiler is used on both Windows and Linux platforms by the Dev-C++ and KDevelop tools, respectively. The format of structured and object-oriented code is much easier to read and follow than in the numbered lines of the past.

Tank War is a two-player game that is played on a single screen using a shared keyboard. The first player uses the W, A, S, and D keys to move his tank, and the Spacebar to fire the main cannon on the tank. The second player uses the arrow keys for movement and the Enter key to fire. The game is shown in Figure 4.1.

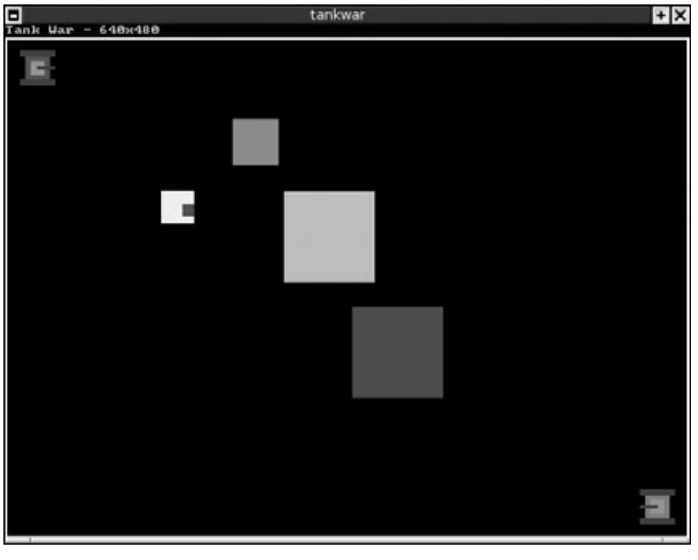


Figure 4.1 *Tank War* is a two-player game in the classic style.

Creating the Tanks

The graphics in *Tank War* are created entirely with the drawing functions included in Allegro. Figure 4.2 shows the four angles of the tank that are drawn based on the tank's direction of travel.

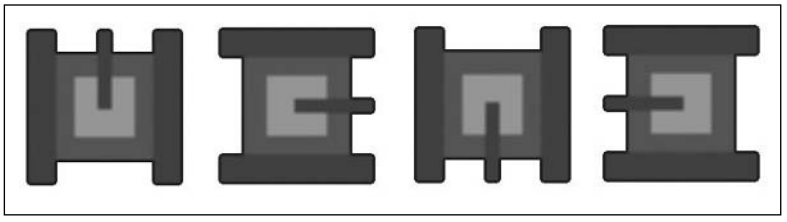


Figure 4.2 The tanks are rendered on the screen using a series of filled rectangles.

The `drawtank` function is called from the main loop to draw each tank according to its current direction. The `drawtank` function looks like this:

```
void drawtank(int num)
{
    int x = tanks[num].x;
    int y = tanks[num].y;
    int dir = tanks[num].dir;

    //draw tank body and turret
    rectfill(screen, x-11, y-11, x+11, y+11, tanks[num].color);
    rectfill(screen, x-6, y-6, x+6, y+6, 7);

    //draw the treads based on orientation
    if (dir == 0 || dir == 2)
    {
        rectfill(screen, x-16, y-16, x-11, y+16, 8);
        rectfill(screen, x+11, y-16, x+16, y+16, 8);
    }
    else
    if (dir == 1 || dir == 3)
    {
        rectfill(screen, x-16, y-16, x+16, y-11, 8);
        rectfill(screen, x-16, y+16, x+16, y+11, 8);
    }

    //draw the turret based on direction
    switch (dir)
    {
        case 0:
            rectfill(screen, x-1, y, x+1, y-16, 8);
            break;
        case 1:
            rectfill(screen, x, y-1, x+16, y+1, 8);
```

```
            break;
        case 2:
            rectfill(screen, x-1, y, x+1, y+16, 8);
            break;
        case 3:
            rectfill(screen, x, y-1, x-16, y+1, 8);
            break;
    }
}
```

Did you notice how the entire tank is constructed with `rectfill` statements? This is one example of improvisation where better technology is not available. For instance, bitmaps and sprites are not yet available because I haven't covered that subject yet, so this game actually draws the tank sprite used in the game. Don't underestimate the usefulness of rendered graphics to enhance a sprite-based game or to create a game entirely. To erase the tank, you simply call the `erasetank` function, which looks like this:

```
void erasetank(int num)
{
    //calculate box to encompass the tank
    int left = tanks[num].x - 17;
    int top = tanks[num].y - 17;
    int right = tanks[num].x + 17;
    int bottom = tanks[num].y + 17;

    //erase the tank
    rectfill(screen, left, top, right, bottom, 0);
}
```

The `erasetank` function is calculated based on the center of the tank (which is how the tank is drawn as well, from the center). Because the tank is 32×32 pixels in size, the `erasetank` function draws a black filled rectangle a distance of 17 pixels in each direction from the center (for a total of 34×34 pixels, to include a small border around the tank, which helps to keep the tank from getting stuck in obstacles).

Firing Weapons

The projectiles fired from each tank are drawn as small rectangles (four pixels total) that move in the current direction the tank is facing until they strike the other tank, an object, or the edge of the screen. You can increase the size of the projectile by increasing the size in the `updatebullet` function (coming up next). To determine whether a hit has occurred, you use the `getpixel` function to "look" at the pixel on the screen right in front of the bullet. If that pixel is black (color 0 or RGB 0,0,0), then the bullet is moved another space.

If that color is anything other than black, then it is a sure hit! The `fireweapon` function gets the bullet started in the right direction.

```
void fireweapon(int num)
{
    int x = tanks[num].x;
    int y = tanks[num].y;

    //ready to fire again?
    if (!bullets[num].alive)
    {
        bullets[num].alive = 1;

        //fire bullet in direction tank is facing
        switch (tanks[num].dir)
        {
            //north
            case 0:
                bullets[num].x = x;
                bullets[num].y = y-22;
                bullets[num].xspd = 0;
                bullets[num].yspd = -BULLETSPEED;
                break;
            //east
            case 1:
                bullets[num].x = x+22;
                bullets[num].y = y;
                bullets[num].xspd = BULLETSPEED;
                bullets[num].yspd = 0;
                break;
            //south
            case 2:
                bullets[num].x = x;
                bullets[num].y = y+22;
                bullets[num].xspd = 0;
                bullets[num].yspd = BULLETSPEED;
                break;
            //west
            case 3:
                bullets[num].x = x-22;
                bullets[num].y = y;
                bullets[num].xspd = -BULLETSPEED;
                bullets[num].yspd = 0;
```

```
        }
    }
}
```

The `fireweapon` function looks at the direction of the current tank to set the X and Y movement values for the bullet. Once it is set up, the bullet will move in that direction until it strikes something or reaches the edge of the screen. The important variable here is `alive`, which determines whether the bullet is moved accordingly using this `updatebullet` function:

```
void updatebullet(int num)
{
    int x = bullets[num].x;
    int y = bullets[num].y;

    if (bullets[num].alive)
    {
        //erase bullet
        rect(screen, x-1, y-1, x+1, y+1, 0);

        //move bullet
        bullets[num].x += bullets[num].xspd;
        bullets[num].y += bullets[num].yspd;
        x = bullets[num].x;
        y = bullets[num].y;

        //stay within the screen
        if (x < 5 || x > SCREEN_W-5 || y < 20 || y > SCREEN_H-5)
        {
            bullets[num].alive = 0;
            return;
        }

        //draw bullet
        x = bullets[num].x;
        y = bullets[num].y;
        rect(screen, x-1, y-1, x+1, y+1, 14);

        //look for a hit
        if (getpixel(screen, bullets[num].x, bullets[num].y))
        {
            bullets[num].alive = 0;
            explode(num, x, y);
        }
```

```
        //print the bullet's position
        textprintf(screen, font, SCREEN_W/2-50, 1, 2,
            "B1 %-3dx%-3d  B2 %-3dx%-3d",
            bullets[0].x, bullets[0].y,
            bullets[1].x, bullets[1].y);
    }
}
```

Tank Movement

To move the tank, each player uses the appropriate keys to move forward, backward, left, right, and to fire the weapon. The first player uses W, A, S, and D to move and the Spacebar to fire, while player two uses the arrow keys to move and Enter to fire. The main loop looks for a key press and calls on the `getinput` function to see which key has been pressed. I will discuss keyboard input in a later chapter; for now all you need to be aware of is an array called `key` that stores the values of each key press.

```
void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])
        gameover = 1;

    //WASD / SPACE keys control tank 1
    if (key[KEY_W])
        forward(0);
    if (key[KEY_D])
        turnright(0);
    if (key[KEY_A])
        turnleft(0);
    if (key[KEY_S])
        backward(0);
    if (key[KEY_SPACE])
        fireweapon(0);

    //arrow / ENTER keys control tank 2
    if (key[KEY_UP])
        forward(1);
    if (key[KEY_RIGHT])
        turnright(1);
    if (key[KEY_DOWN])
        backward(1);
    if (key[KEY_LEFT])
```

```
        turnleft(1);
    if (key[KEY_ENTER])
        fireweapon(1);

    //short delay after keypress
    rest(10);
}
```

Collision Detection

I have already explained how the bullets use `getpixel` to determine when a collision has occurred (when the bullet hits a tank or obstacle). But what about collision detection when you are moving the tanks themselves? There are several obstacles on the battlefield to add a little strategy to the game; they offer a place to hide or maneuver around (or straight through if you blow up the obstacles). The `clearpath` function is used to determine whether the ship can move. The function checks the screen boundaries and obstacles on the screen to clear a path for the tank or prevent it from moving any further in that direction. The function also takes into account reverse motion because the tanks can move forward or backward. `clearpath` is a bit lengthy, so I'll leave it for the main code listing later in the chapter. The `clearpath` function calls the `checkpath` function to actually see whether the tank's pathway is clear for movement. (`checkpath` is called multiple times for each tank.)

```
int checkpath(int x1,int y1,int x2,int y2,int x3,int y3)
{
    if (getpixel(screen, x1, y1) ||
        getpixel(screen, x2, y2) ||
        getpixel(screen, x3, y3))
        return 1;
    else
        return 0;
}
```

All that remains of the program are the logistical functions for setting up the screen, modifying the speed and direction of each tank, displaying the score, placing the random debris, and so on.

The Complete Tank War Source Code

The code listing for *Tank War* is included here in its entirety. Despite having already shown you many of the functions in this program, I think it's important at this point to show you the entire listing in one fell swoop so there is no confusion. Of course, you can open the *Tank War* project that is located on the CD-ROM that accompanies this book; look inside a folder called chapter04 for the complete project for Visual C++, Dev-C++, or KDevelop. If you are using some other operating system, you can still compile this code

for your favorite compiler by typing it into your text editor and including the Allegro library. (If you need some pointers, refer to Appendix E, "Configuring Allegro for Microsoft Visual C++ and Other Compilers.")

The Tank War Header File

The first code listing is for the header file, which includes the variables, structures, constants, and function prototypes for the game. You will want to add a new file to the project called tankwar.h. The main source code file (main.c) will try to include the header file by this filename. If you need help configuring your compiler to link to the Allegro game library, refer to Appendix E. If you have not yet installed Allegro, you might want to go back and read Chapter 2 and refer to Appendix F, "Compiling the Allegro Source Code."

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Chapter 4 - Tank War Game
////////////////////////////////////////////////////////////////////////

#ifndef _TANKWAR_H
#define _TANKWAR_H

#include "allegro.h"

//define some game constants
#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define BLOCKS 5
#define BLOCKSIZE 100
#define MAXSPEED 2
#define BULLETSPEED 10
#define TAN makecol(255,242,169)
#define CAMO makecol(64,142,66)
#define BURST makecol(255,189,73)

//define tank structure
struct tagTank
{
    int x,y;
    int dir,speed;
    int color;
    int score;
```

```
} tanks[2];

//define bullet structure
struct tagBullet
{
    int x,y;
    int alive;
    int xspd,yspd;

} bullets[2];

int gameover = 0;

//function prototypes
void drawtank(int num);
void erasetank(int num);
void movetank(int num);
void explode(int num, int x, int y);
void updatebullet(int num);
int checkpath(int x1,int y1,int x2,int y2,int x3,int y3);
void clearpath(int num);
void fireweapon(int num);
void forward(int num);
void backward(int num);
void turnleft(int num);
void turnright(int num);
void getinput();
void setuptanks();
void score(int);
void print(const char *s, int c);
void setupdebris();
void setupscreen();

#endif
```

The Tank War Source File

The primary source code file for *Tank War* includes the tankwar.h header file (which in turn includes allegro.h). Included in this code listing are all of the functions needed by the game in addition to the `main` function (containing the game loop). You can type this code in as-is for whatever OS and IDE you are using; if you have included the Allegro library, it will run without issue. This game is wonderfully easy to get to work because it requires no bitmap files, uses no backgrounds, and simply draws directly to the primary screen buffer (which can be full-screen or windowed).

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Chapter 4 - Tank War Game
////////////////////////////////////////////////////////////////////////

#include "tankwar.h"

////////////////////////////////////////////////////////////////////////
// drawtank function
// construct the tank using drawing functions
////////////////////////////////////////////////////////////////////////
void drawtank(int num)
{
    int x = tanks[num].x;
    int y = tanks[num].y;
    int dir = tanks[num].dir;

    //draw tank body and turret
    rectfill(screen, x-11, y-11, x+11, y+11, tanks[num].color);
    rectfill(screen, x-6, y-6, x+6, y+6, 7);

    //draw the treads based on orientation
    if (dir == 0 || dir == 2)
    {
        rectfill(screen, x-16, y-16, x-11, y+16, 8);
        rectfill(screen, x+11, y-16, x+16, y+16, 8);
    }
    else
    if (dir == 1 || dir == 3)
    {
        rectfill(screen, x-16, y-16, x+16, y-11, 8);
        rectfill(screen, x-16, y+16, x+16, y+11, 8);
    }

    //draw the turret based on direction
    switch (dir)
    {
        case 0:
            rectfill(screen, x-1, y, x+1, y-16, 8);
            break;
        case 1:
            rectfill(screen, x, y-1, x+16, y+1, 8);
```

```
            break;
        case 2:
            rectfill(screen, x-1, y, x+1, y+16, 8);
            break;
        case 3:
            rectfill(screen, x, y-1, x-16, y+1, 8);
            break;
    }
}

////////////////////////////////////////////////////////////////////////
// erasetank function
// erase the tank using rectfill
////////////////////////////////////////////////////////////////////////
void erasetank(int num)
{
    //calculate box to encompass the tank
    int left = tanks[num].x - 17;
    int top = tanks[num].y - 17;
    int right = tanks[num].x + 17;
    int bottom = tanks[num].y + 17;

    //erase the tank
    rectfill(screen, left, top, right, bottom, 0);
}

////////////////////////////////////////////////////////////////////////
// movetank function
// move the tank in the current direction
////////////////////////////////////////////////////////////////////////
void movetank(int num)
{
    int dir = tanks[num].dir;
    int speed = tanks[num].speed;

    //update tank position based on direction
    switch(dir)
    {
        case 0:
            tanks[num].y -= speed;
            break;
        case 1:
            tanks[num].x += speed;
```

```
            break;
        case 2:
            tanks[num].y += speed;
            break;
        case 3:
            tanks[num].x -= speed;
    }

    //keep tank inside the screen
    if (tanks[num].x > SCREEN_W-22)
    {
        tanks[num].x = SCREEN_W-22;
        tanks[num].speed = 0;
    }
    if (tanks[num].x < 22)
    {
        tanks[num].x = 22;
        tanks[num].speed = 0;
    }
    if (tanks[num].y > SCREEN_H-22)
    {
        tanks[num].y = SCREEN_H-22;
        tanks[num].speed = 0;
    }
    if (tanks[num].y < 22)
    {
        tanks[num].y = 22;
        tanks[num].speed = 0;
    }
}

//////////////////////////////////////////////////////////////////////////
// explode function
// display random boxes to simulate an explosion
//////////////////////////////////////////////////////////////////////////
void explode(int num, int x, int y)
{

    int n;

    //retrieve location of enemy tank
    int tx = tanks[!num].x;
    int ty = tanks[!num].y;
```

```
    //is bullet inside the boundary of the enemy tank?
    if (x > tx-16 && x < tx+16 && y > ty-16 && y < ty+16)
        score(num);

    //draw some random circles for the "explosion"
    for (n = 0; n < 10; n++)
    {
        rectfill(screen, x-16, y-16, x+16, y+16, rand() % 16);
        rest(1);
    }

    //clear the area of debris
    rectfill(screen, x-16, y-16, x+16, y+16, 0);

}

////////////////////////////////////////////////////////////////////////////
// updatebullet function
// update the position of a bullet
////////////////////////////////////////////////////////////////////////////
void updatebullet(int num)
{
    int x = bullets[num].x;
    int y = bullets[num].y;

    if (bullets[num].alive)
    {
        //erase bullet
        rect(screen, x-1, y-1, x+1, y+1, 0);

        //move bullet
        bullets[num].x += bullets[num].xspd;
        bullets[num].y += bullets[num].yspd;
        x = bullets[num].x;
        y = bullets[num].y;

        //stay within the screen
        if (x < 5 || x > SCREEN_W-5 || y < 20 || y > SCREEN_H-5)
        {
            bullets[num].alive = 0;
            return;
        }
```

```
        //draw bullet
        x = bullets[num].x;
        y = bullets[num].y;
        rect(screen, x-1, y-1, x+1, y+1, 14);

        //look for a hit
        if (getpixel(screen, bullets[num].x, bullets[num].y))
        {
            bullets[num].alive = 0;
            explode(num, x, y);
        }

        //print the bullet's position
        textprintf(screen, font, SCREEN_W/2-50, 1, 2,
            "B1 %-3dx%-3d  B2 %-3dx%-3d",
            bullets[0].x, bullets[0].y,
            bullets[1].x, bullets[1].y);

    }
}

////////////////////////////////////////////////////////////////////////
// checkpath function
// check to see if a point on the screen is black
////////////////////////////////////////////////////////////////////////
int checkpath(int x1,int y1,int x2,int y2,int x3,int y3)
{
    if (getpixel(screen, x1, y1) ||
        getpixel(screen, x2, y2) ||
        getpixel(screen, x3, y3))
        return 1;
    else
        return 0;
}

////////////////////////////////////////////////////////////////////////
// clearpath function
// verify that the tank can move in the current direction
////////////////////////////////////////////////////////////////////////
void clearpath(int num)
{
    //shortcut vars
    int dir = tanks[num].dir;
```

```
int speed = tanks[num].speed;
int x = tanks[num].x;
int y = tanks[num].y;

switch(dir)
{
    //check pixels north
    case 0:
        if (speed > 0)
        {
            if (checkpath(x-16, y-20, x, y-20, x+16, y-20))
                tanks[num].speed = 0;
        }
        else
            //if reverse dir, check south
            if (checkpath(x-16, y+20, x, y+20, x+16, y+20))
                tanks[num].speed = 0;
        break;

    //check pixels east
    case 1:
        if (speed > 0)
        {
            if (checkpath(x+20, y-16, x+20, y, x+20, y+16))
                tanks[num].speed = 0;
        }
        else
            //if reverse dir, check west
            if (checkpath(x-20, y-16, x-20, y, x-20, y+16))
                tanks[num].speed = 0;
        break;

    //check pixels south
    case 2:
        if (speed > 0)
        {
            if (checkpath(x-16, y+20, x, y+20, x+16, y+20 ))
                tanks[num].speed = 0;
        }
        else
            //if reverse dir, check north
            if (checkpath(x-16, y-20, x, y-20, x+16, y-20))
                tanks[num].speed = 0;
```

```
            break;

        //check pixels west
        case 3:
            if (speed > 0)
            {
                if (checkpath(x-20, y-16, x-20, y, x-20, y+16))
                    tanks[num].speed = 0;
            }
            else
                //if reverse dir, check east
                if (checkpath(x+20, y-16, x+20, y, x+20, y+16))
                    tanks[num].speed = 0;
            break;
    }
}

////////////////////////////////////////////////////////////////////////
// fireweapon function
// configure a bullet's direction and speed and activate it
////////////////////////////////////////////////////////////////////////
void fireweapon(int num)
{
    int x = tanks[num].x;
    int y = tanks[num].y;

    //ready to fire again?
    if (!bullets[num].alive)
    {
        bullets[num].alive = 1;

        //fire bullet in direction tank is facing
        switch (tanks[num].dir)
        {
            //north
            case 0:
                bullets[num].x = x;
                bullets[num].y = y-22;
                bullets[num].xspd = 0;
                bullets[num].yspd = -BULLETSPEED;
                break;
            //east
            case 1:
```

```
                bullets[num].x = x+22;
                bullets[num].y = y;
                bullets[num].xspd = BULLETSPEED;
                bullets[num].yspd = 0;
                break;
            //south
            case 2:
                bullets[num].x = x;
                bullets[num].y = y+22;
                bullets[num].xspd = 0;
                bullets[num].yspd = BULLETSPEED;
                break;
            //west
            case 3:
                bullets[num].x = x-22;
                bullets[num].y = y;
                bullets[num].xspd = -BULLETSPEED;
                bullets[num].yspd = 0;
        }
    }
}

////////////////////////////////////////////////////////////////////////
// forward function
// increase the tank's speed
////////////////////////////////////////////////////////////////////////
void forward(int num)
{
    tanks[num].speed++;
    if (tanks[num].speed > MAXSPEED)
        tanks[num].speed = MAXSPEED;
}

////////////////////////////////////////////////////////////////////////
// backward function
// decrease the tank's speed
////////////////////////////////////////////////////////////////////////
void backward(int num)
{
    tanks[num].speed--;
    if (tanks[num].speed < -MAXSPEED)
        tanks[num].speed = -MAXSPEED;
}
```

```
////////////////////////////////////////////////////////////////////////
// turnleft function
// rotate the tank counter-clockwise
////////////////////////////////////////////////////////////////////////
void turnleft(int num)
{
    tanks[num].dir--;
    if (tanks[num].dir < 0)
        tanks[num].dir = 3;
}

////////////////////////////////////////////////////////////////////////
// turnright function
// rotate the tank clockwise
////////////////////////////////////////////////////////////////////////
void turnright(int num)
{
    tanks[num].dir++;
    if (tanks[num].dir > 3)
        tanks[num].dir = 0;
}

////////////////////////////////////////////////////////////////////////
// getinput function
// check for player input keys (2 player support)
////////////////////////////////////////////////////////////////////////
void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])
        gameover = 1;

    //WASD / SPACE keys control tank 1
    if (key[KEY_W])
        forward(0);
    if (key[KEY_D])
        turnright(0);
    if (key[KEY_A])
        turnleft(0);
    if (key[KEY_S])
        backward(0);
    if (key[KEY_SPACE])
        fireweapon(0);
```

```
    //arrow / ENTER keys control tank 2
    if (key[KEY_UP])
        forward(1);
    if (key[KEY_RIGHT])
        turnright(1);
    if (key[KEY_DOWN])
        backward(1);
    if (key[KEY_LEFT])
        turnleft(1);
    if (key[KEY_ENTER])
        fireweapon(1);

    //short delay after keypress
    rest(10);
}

////////////////////////////////////////////////////////////////////////////
// score function
// add a point to the specified player's score
////////////////////////////////////////////////////////////////////////////
void score(int player)
{
    //update score
    int points = ++tanks[player].score;

    //display score
    textprintf(screen, font, SCREEN_W-70*(player+1), 1, BURST,
        "P%d: %d", player+1, points);
}

////////////////////////////////////////////////////////////////////////////
// setuptanks function
// set up the starting condition of each tank
////////////////////////////////////////////////////////////////////////////
void setuptanks()
{
    //player 1
    tanks[0].x = 30;
    tanks[0].y = 40;
    tanks[0].dir = 1;
    tanks[0].speed = 0;
    tanks[0].color = 9;
    tanks[0].score = 0;
```

```
    //player 2
    tanks[1].x = SCREEN_W-30;
    tanks[1].y = SCREEN_H-30;
    tanks[1].dir = 3;
    tanks[1].speed = 0;
    tanks[1].color = 12;
    tanks[1].score = 0;
}

////////////////////////////////////////////////////////////////////////////
// setupdebris function
// set up the debris on the battlefield
////////////////////////////////////////////////////////////////////////////
void setupdebris()
{
    int n,x,y,size,color;

    //fill the battlefield with random debris
    for (n = 0; n < BLOCKS; n++)
    {
        x = BLOCKSIZE + rand() % (SCREEN_W-BLOCKSIZE*2);
        y = BLOCKSIZE + rand() % (SCREEN_H-BLOCKSIZE*2);
        size = (10 + rand() % BLOCKSIZE)/2;
        color = makecol(rand()%255, rand()%255, rand()%255);
        rectfill(screen, x-size, y-size, x+size, y+size, color);
    }

}

////////////////////////////////////////////////////////////////////////////
// setupscreen function
// set up the graphics mode and game screen
////////////////////////////////////////////////////////////////////////////
void setupscreen()
{
    //set video mode
    int ret = set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }
```

```
    //print title
    textprintf(screen, font, 1, 1, BURST,
        "Tank War - %dx%d", SCREEN_W, SCREEN_H);

    //draw screen border
    rect(screen, 0, 12, SCREEN_W-1, SCREEN_H-1, TAN);
    rect(screen, 1, 13, SCREEN_W-2, SCREEN_H-2, TAN);

}

////////////////////////////////////////////////////////////////////////
// main function
// start point of the program
////////////////////////////////////////////////////////////////////////
void main(void)
{
    //initialize everything
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    setupscreen();
    setupdebris();
    setuptanks();

    //game loop
    while(!gameover)
    {
        //erase the tanks
        erasetank(0);
        erasetank(1);

        //check for collisions
        clearpath(0);
        clearpath(1);

        //move the tanks
        movetank(0);
        movetank(1);

        //draw the tanks
        drawtank(0);
        drawtank(1);
```

```
        //update the bullets
        updatebullet(0);
        updatebullet(1);

        //check for keypresses
        if (keypressed())
            getinput();

        //slow the game down (adjust as necessary)
        rest(30);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Summary

Congratulations on completing your first game with Allegro! It has been a short journey thus far—we're only in the fourth chapter of the book. Contrast this with the enormous amount of information that would have been required in advance to compile even a simple game, such as *Tank War*, using standard graphics libraries, such as DirectX or SVGAlib! It would have taken this amount of source code just to set up the screen and prepare the program for the actual game. That is where Allegro truly shines—by abstracting the logistical issues into a common set of library functions that work regardless of the underlying operating system.

This also concludes Part I of the book and sends you venturing into Part II, which covers the core functionality of Allegro in much more detail. You will learn how to use animated sprites and create scrolling backgrounds, and we'll discuss the next upgrade to *Tank War*. That's right, this isn't the end of *Tank War*! From this point forward, we'll be improving the game with each new chapter. For starters, this game really needs some design and direction (the focus of Chapter 5). By the time you're finished, the game will feature a scrolling background, a tile-based battlefield, sound effects…the whole works!

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the primary graphics drawing function used to draw the tanks in *Tank War*?
   A. `rectfill`
   B. `fillrect`
   C. `drawrect`
   D. `rectangle`

2. What function in *Tank War* sets up a bullet to fire it in the direction of the tank?
   A. `pulltrigger`
   B. `launchprojectile`
   C. `fireweapon`
   D. `firecannon`

3. What function in *Tank War* updates the position and draws each projectile?
   A. `updatecannon`
   B. `movebullet`
   C. `moveprojectile`
   D. `updatebullet`

4. What is the name of the organization that produced GCC?
   A. Free Software Foundation
   B. GNU
   C. Freeware
   D. Open Source

5. How many players are supported in *Tank War* at the same time?
   A. 1
   B. 2
   C. 3
   D. 4

6. What is the technical terminology for handling two objects that crash in the game?
   A. Crash override
   B. Sprite insurance
   C. Collision detection
   D. Handling the crash

7. What function in *Tank War* keeps the tanks from colliding with other objects?
   A. `makepath`
   B. `clearpath`
   C. `buildpath`
   D. `dontcollide`

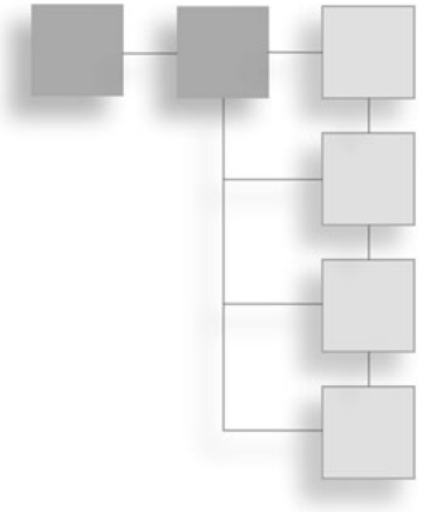
8. Which function in *Tank War* helps to find out whether a point on the screen is black?
   A. `getpixel`
   B. `findcolor`
   C. `getcolor`
   D. `checkpixel`

9. What is the standard constant used to run Allegro in windowed mode?
   A. `GFX_RUNINA_WINDOW`
   B. `GFX_DETECT_WINDOWED`
   C. `GFX_AUTODETECT_WINDOWS`
   D. `GFX_AUTODETECT_WINDOWED`

10. What function in Allegro is used to slow the game down?
    A. `pause`
    B. `slow`
    C. `rest`
    D. `stop`

This page intentionally left blank

CHAPTER 5



Programming the Keyboard, Mouse, and Joystick

Welcome to the input chapter, focusing on programming the keyboard, mouse, and joystick! This chapter is a lot of fun, and I know you will enjoy learning about these three input devices because there are some great example programs here to demonstrate how to get a handle on this subject. By the time you have finished this chapter, you will be able to scan for individual keys, read their scan codes, and detect multiple button presses. You will learn about Allegro's buffered keyboard input routines and discover ASCII. (See Appendix B, "Useful Tables," for a table of ASCII values.) You will learn how to read the mouse position, create a custom graphical mouse pointer, check up on the mouse wheel, and discover something called mickeys. You will also learn how to read the joystick, find out what features the currently installed joystick provides (such as analog/digital sticks, buttons, hats, sliders, and so on), and read the joystick values to provide input for a game. As you go through this chapter, you will discover several sample programs that make the subjects easy to understand, including a stargate program, a missile defense system, a hyperspace teleportation program, and a joystick program that involves bouncing balls. Are you ready to dig into the fun subject of device input? I thought so! Let's do it.

Here is a breakdown of the major topics in this chapter:

- Handling keyboard input
- Detecting key presses
- Dealing with buffered keyboard input
- Handling mouse input
- Reading the mouse position
- Working with relative mouse motion
- Handling joystick input

- Handling joystick controller movement
- Handling joystick button presses

Handling Keyboard Input

Allegro provides functions for handling buffered input and individual key states. Keyboard input might seem strange to gamers who have dedicated their lives to console games, but the keyboard has been the mainstay of PC gaming for two dozen years and counting, and it is not likely to be replaced anytime soon. The joystick has had only limited acceptance on the PC, but the mouse has had a larger influence on games, primarily due to modern operating systems. Allegro supports both ANSI (one-byte) and Unicode (two-byte) character systems. (By the way, ANSI stands for *American National Standards Institute* and ASCII stands for *American Standard Code for Information Interchange.*)

The Keyboard Handler

Allegro abstracts the keyboard from the operating system so the generic keyboard routines will work on any computer system you have targeted for your game (Windows, Linux, Mac, and so on). However, that abstraction does not take anything away from the inherent capabilities of any system because the library is custom-written for each platform. The Windows version of Allegro utilizes DirectInput for the keyboard handler. Since there really is no magic to the subject, let's just jump right in and work with the keyboard.

Before you can start using the keyboard routines in Allegro, you must initialize the keyboard handler with the `install_keyboard` function.

```
int install_keyboard();
```

If you try to use the keyboard routines before initializing, the program will likely crash (or at best, it won't respond to the keyboard). Once you have initialized the keyboard handler, there is no need to uninitialize it—that is handled by Allegro via the `allegro_exit` function (which is called automatically before Allegro stops running). But if you do find a need to remove the keyboard handler, you can use `remove_keyboard`.

```
void remove_keyboard();
```

Some operating systems, such as those with preemptive multitasking, do not support the keyboard interrupt handler that Allegro uses. You can use the `poll_keyboard` function to poll the keyboard if your program will need to be run on systems that don't support the keyboard interrupt service routine. Why would this be the case? Allegro is a multi-threaded library. When you call `allegro_init` and functions such as `install_keyboard`, Allegro creates several threads to handle events, scroll the screen, draw sprites, and so on.

```
int poll_keyboard();
```

When you first call `poll_keyboard`, Allegro switches to polled mode, after which the keyboard *must* be polled even if an interrupt or a thread is available. To determine when polling mode is active, use the `keyboard_needs_poll` function.

```
int keyboard_needs_poll();
```

Detecting Key Presses

Allegro makes it very easy to detect key presses. To check for an individual key, you can use the `key` array that is populated with values when the keyboard is polled (or during regular intervals, when run as a thread).

```
extern volatile char key[KEY_MAX];
```

Most of the keys on computer systems are supported by name using constant key values defined in the Allegro library header files. If you want to see all of the key definitions yourself, look in the Allegro library folder for a header file called keyboard.h, in which all the keys are defined. Note also that Allegro defines individual keys, not ASCII codes, so the main numeric keys are not the same as the numeric keypad keys, and the Ctrl, Alt, and Shift keys are treated individually. Pressing Shift+A results in two key presses, not just the "A" key. The buffered keyboard routines (covered next) will differentiate lowercase "a" from uppercase "A." Table 5.1 lists a few of the most common key codes.

Table 5.1 Common Key Codes

Key	Description
KEY_A…KEY_Z	Standard alphabetic keys
KEY_0…KEY_9	Standard numeric keys
KEY_0_PAD…KEY_9_PAD	Numeric keypad keys
KEY_F1…KEY_F12	Function keys
KEY_ESC	Esc key
KEY_BACKSPACE	Backspace key
KEY_TAB	Tab key
KEY_ENTER	Enter key
KEY_SPACE	Space key
KEY_INSERT	Insert key
KEY_DEL	Delete key
KEY_HOME	Home key
KEY_END	End key
KEY_PGUP	Page Up key
KEY_PGDN	Page Down key
KEY_LEFT	Left arrow key
KEY_RIGHT	Right arrow key
KEY_UP	Up arrow key
KEY_DOWN	Down arrow key
KEY_LSHIFT	Left Shift key
KEY_RSHIFT	Right Shift key

The sample programs in the chapters thus far have used the keyboard handler without fully explaining it because it's difficult to demonstrate anything without some form of keyboard input. The typical game loop looks like this:

```
while (!key[KEY_ESC])
{
    //do some stuff
}
```

This loop continues to run until the Esc key is pressed, at which point the loop is exited. Direct access to the key codes means the program does not use the keyboard buffer; rather, it checks each key individually, bypassing the keyboard buffer entirely. You can still check the key codes while also processing key presses in the keyboard buffer using the buffered input functions, such as `readkey`.

The Stargate Program

The *Stargate* program demonstrates how to use the keyboard scan codes to detect when specific keys have been pressed. You will use this technology to decipher the ancient hieroglyphs on the gate and attempt to open a wormhole to Abydos. If all scholarly attempts fail, you can resort to trying random dialing sequences using the keys on the keyboard. Our scientists have thus far failed in their attempt to decipher the gate symbols, as you can see in Figure 5.1. What this program really needs are some sound effects, but that will have to wait for Chapter 15, "Mastering the Audible Realm: Allegro's Sound Support."

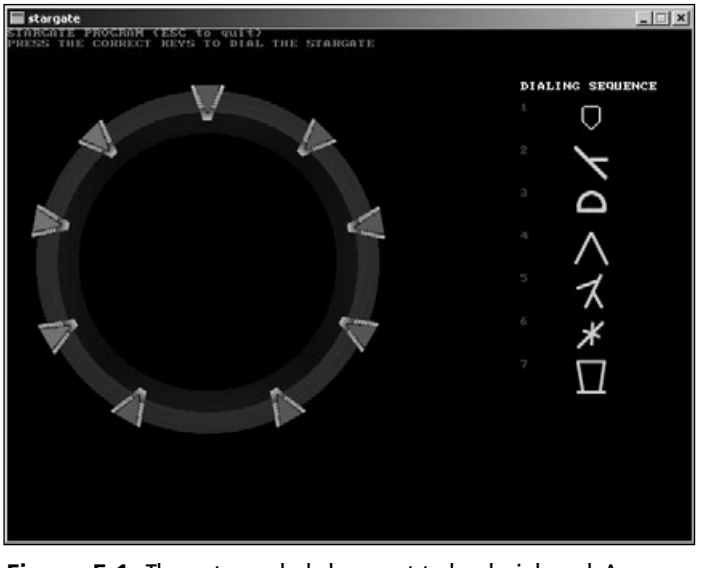


Figure 5.1 The gate symbols have yet to be deciphered. Are you up to the challenge?

Should you successfully crack the gate codes, the result will look like Figure 5.2.

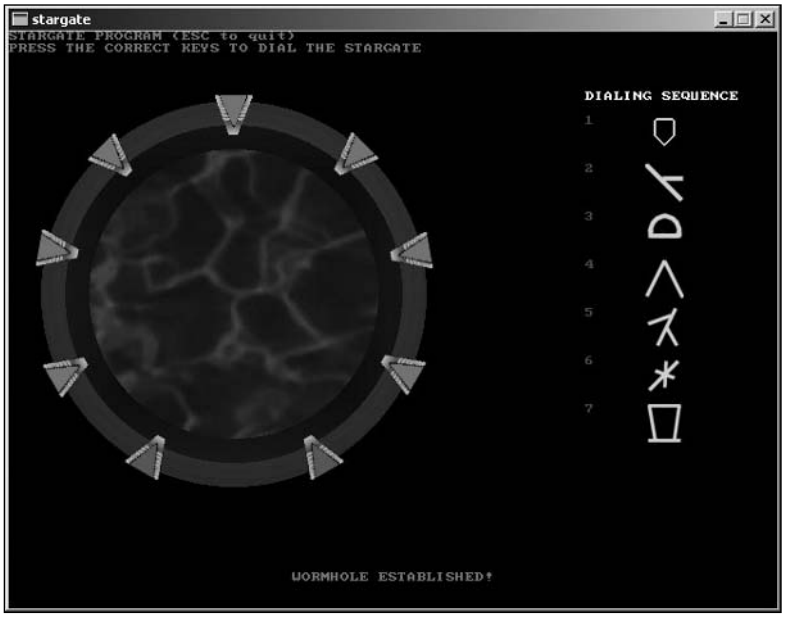


Figure 5.2 Opening a gateway to another world—speculative fantasy or a real possibility?

```
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLUE makecol(64,64,255)
#define RED makecol(255,64,64)

typedef struct POINT
{
    int x, y;
} POINT;

POINT coords[] = {{25,235},
                  {15,130},
                  {60,50},
                  {165,10},
                  {270,50},
                  {325,135},
                  {315,235}};

BITMAP *stargate;
BITMAP *water;
```

```
BITMAP *symbols[7];
int count = 0;

//helper function to highlight each shevron
void shevron(int num)
{
    floodfill(screen, 20+coords[num].x, 50+coords[num].y, RED);

    if (++count > 6)
    {
        masked_blit(water,screen,0,0,67,98,water->w,water->h);
        textout_centre(screen,font,"WORMHOLE ESTABLISHED!",
            SCREEN_W/2, SCREEN_H-30, RED);
    }
}

//main function
void main(void)
{
    int n;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0);
    install_keyboard();

    //load the stargate image
    stargate = load_bitmap("stargate.bmp", NULL);
    blit(stargate,screen,0,0,20,50,stargate->w,stargate->h);

    //load the water image
    water = load_bitmap("water.bmp", NULL);

    //load the symbol images
    symbols[0] = load_bitmap("symbol1.bmp", NULL);
    symbols[1] = load_bitmap("symbol2.bmp", NULL);
    symbols[2] = load_bitmap("symbol3.bmp", NULL);
    symbols[3] = load_bitmap("symbol4.bmp", NULL);
    symbols[4] = load_bitmap("symbol5.bmp", NULL);
    symbols[5] = load_bitmap("symbol6.bmp", NULL);
    symbols[6] = load_bitmap("symbol7.bmp", NULL);
```

```
//display the symbols
textout(screen,font,"DIALING SEQUENCE", 480, 50, WHITE);
for (n=0; n<7; n++)
{
    textprintf(screen,font,480,70+n*40,BLUE,"%d", n+1);
    blit(symbols[n],screen,0,0,530,70+n*40,32,32);
}

//display title
textout(screen,font,"STARGATE PROGRAM (ESC to quit)", 0, 0, RED);
textout(screen,font,"PRESS THE CORRECT KEYS (A-Z) "\
    "TO DIAL THE STARGATE", 0, 10, RED);

//main loop
while (!key[KEY_ESC])
{
    //check for proper sequence
    switch (count)
    {
        case 0:
            if (key[KEY_A]) shevron(0);
            break;
        case 1:
            if (key[KEY_Y]) shevron(1);
            break;
        case 2:
            if (key[KEY_B]) shevron(2);
            break;
        case 3:
            if (key[KEY_A]) shevron(3);
            break;
        case 4:
            if (key[KEY_B]) shevron(4);
            break;
        case 5:
            if (key[KEY_T]) shevron(5);
            break;
        case 6:
            if (key[KEY_U]) shevron(6);
            break;
    }
}
```

```
    //clean up
    destroy_bitmap(stargate);
    destroy_bitmap(water);
    for (n=0; n<7; n++)
        destroy_bitmap(symbols[n]);

    allegro_exit();
}
END_OF_MAIN();
```

Buffered Keyboard Input

Buffered keyboard input is a less direct way of reading keyboard input in which individual key codes are not scanned; instead, the ASCII code is returned by one of the buffered keyboard input functions, such as `readkey`.

```
int readkey();
```

The `readkey` function returns the ASCII code of the next character in the keyboard buffer. If no key has been pressed, then `readkey` waits for the next key press. There is a similar function for handling Unicode keys called `ureadkey`, which returns the Unicode value (a two-byte value similar to ASCII) while returning the scan code as a pointer. (I have often wondered why Allegro doesn't simply return these values as a four-byte long.)

```
int ureadkey(int *scancode);
```

The `readkey` function actually returns two values using a two-byte integer value. The low byte of the return value contains the ASCII code (which changes based on Ctrl, Alt, and Shift keys), while the high byte contains the scan code (which is always the same regardless of the control keys). Because the scan code is included in the upper byte, you can use the predefined `key` array to detect buffered key presses by shifting the bits. Shifting the value returned by `readkey` by eight results in the scan code. For instance:

```
if ((readkey() >> 8) == KEY_TAB)
    printf("You pressed Tab\n");
```

Of course, it is easier to use just the `key` array unless you need to read both the scan code and the ASCII code at the same time, which is where `readkey` comes in handy.

As an alternative, you can also check the ASCII code and detect control key sequences at the same time using the `key_shifts` value.

```
extern volatile int key_shifts;
```

This integer contains a bitmask with the following possible values:

```
KB_SHIFT_FLAGKB_CTRL_FLAGKB_ALT_FLAG
KB_LWIN_FLAG
KB_RWIN_FLAG
KB_MENU_FLAG
KB_SCROLOCK_FLAG
KB_NUMLOCK_FLAG
KB_CAPSLOCK_FLAG
KB_INALTSEQ_FLAG
KB_ACCENT1_FLAG
KB_ACCENT2_FLAG
KB_ACCENT3_FLAG
KB_ACCENT4_FLAG
```

For instance:

```
if ((key_shifts & KB_CTRL_FLAG) && (readkey() == 13))
    printf("You pressed CTRL+Enter\n");
```

Of course, I personally find it easier to simply write the code this way:

```
if ((key[KEY_CTRL] && key[KEY_ENTER])
    printf("You pressed CTRL+Enter\n");
```

You can also use a support function provided by Allegro to convert the scan code to an ASCII value with the `scancode_to_ascii` function.

```
int scancode_to_ascii(int scancode);
```

One more support function that you might want to use is `set_keyboard_rate`, which changes the key repeat rate of the keyboard (in milliseconds). You can disable the key repeat by passing zeros to this function.

```
void set_keyboard_rate(int delay, int repeat);
```

Simulating Key Presses

Suppose you have written a game and you want to create a game demo, but you don't want to write a complicated program just to demonstrate a "proof of concept." There is an elegant solution to the problem—simulating key presses. Allegro provides two functions you can use to insert keys into the keyboard buffer so it will appear as if those keys were actually pressed.

The function is called `simulate_keypress`, and it has a similar support function for Unicode called `simulate_ukeypress`. Here are the definitions:

```
void simulate_keypress(int key);
void simulate_ukeypress(int key, int scancode);
```

In addition to inserting keys into the keyboard buffer, you can also clear the keyboard buffer entirely using the `clear_keybuf` function.

```
void clear_keybuf();
```

The KeyTest Program

I would be remiss if I didn't provide a sample program to demonstrate buffered keyboard input, although this small sample program is not as interesting as the last one. Nevertheless, it always helps to see the theory of a particular subject in action. Figure 5.3 shows the *KeyTest* program. This is a convenient program to keep handy because you'll frequently need keyboard scan codes, and this program makes it easy to look them up (knowing that you are free to use Allegro's predefined keys or the scan codes directly).

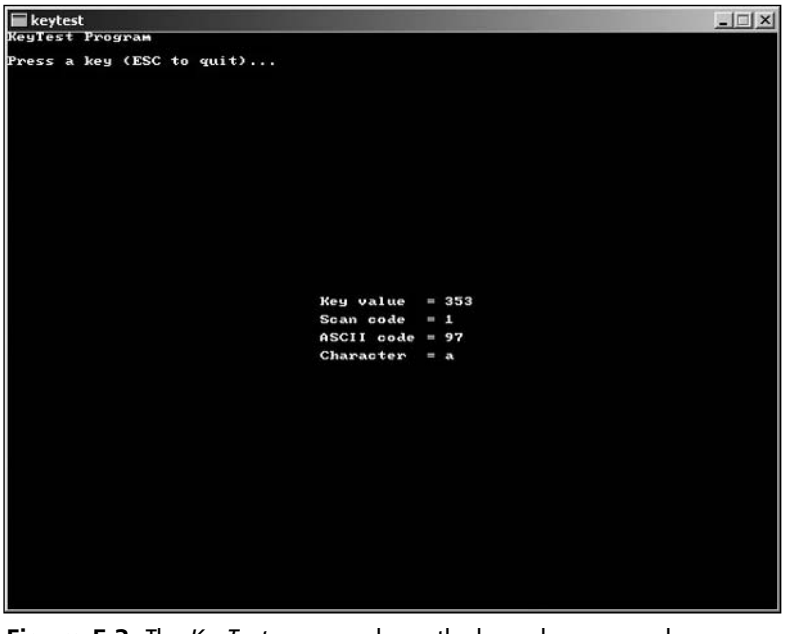


Figure 5.3 The *KeyTest* program shows the key value, scan code, ASCII code, and character.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

void main(void)
{
```

```
    int k, x, y;
    int scancode, ascii;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();

    //display title
    textout(screen,font,"KeyTest Program", 0, 0, WHITE);
    textout(screen,font,"Press a key (ESC to quit)...", 0, 20, WHITE);

    //set starting position for text
    x = SCREEN_W/2 - 60;
    y = SCREEN_H/2 - 20;

    while (!key[KEY_ESC])
    {
        //get and convert scan code
        k = readkey();
        scancode = (k >> 8);
        ascii = scancode_to_ascii(scancode);

        //display key values
        textprintf(screen, font, x, y, WHITE,
            "Key value  = %-6d", k);
        textprintf(screen, font, x, y+15, WHITE,
            "Scan code  = %-6d", scancode);
        textprintf(screen, font, x, y+30, WHITE,
            "ASCII code = %-6d", ascii);
        textprintf(screen, font, x, y+45, WHITE,
            "Character  = %-6c", (char)ascii);
    }
    allegro_exit();
}
END_OF_MAIN();
```

Handling Mouse Input

Mouse input is probably even more vital to a modern game than keyboard input, so support for the mouse is not just an option, it is an assumption, a requirement (unless you are planning to develop a text game).

The Mouse Handler

Allegro is consistent with the input routines, so it is fairly easy to explain how to enable the mouse handler. The one thing you must remember is that the mouse routines (which I'll go over shortly) must only be used after the mouse handler has been installed with the `install_mouse` function.

```
int install_mouse();
```

Although it is not required because `allegro_exit` handles this aspect for you, you can use the `remove_mouse` function to remove the mouse handler.

```
void remove_mouse();
```

Another similarity between the mouse and keyboard handlers is the ability to poll the mouse rather than using the asynchronous interrupt handler to feed values to the mouse variables and functions at your disposal.

```
int poll_mouse();
```

When you have forced mouse polling by calling this function, or when your program is running under an operating system that doesn't support asynchronous interrupt handlers, you can check the polled state using `mouse_needs_poll`. If you suspect that polling might be necessary (based on the operating system you are targeting for the game), it's a good idea to call this function to determine whether polling is indeed needed.

```
int mouse_needs_poll();
```

Reading the Mouse Position

After you install the mouse handler, you automatically have access to the mouse values and functions without much ado (or any more effort). The `mouse_x` and `mouse_y` variables are defined and populated with the mouse position by Allegro.

```
extern volatile int mouse_x;
extern volatile int mouse_y;
```

The `mouse_z` variable contains the current value of the mouse wheel (if supported by the mouse driver and the operating system). I think it's a great idea to support the mouse wheel in a game whenever possible because it's a frequent and popular option, and most new mice have mouse wheels.

```
extern volatile int mouse_z;
```

Detecting Mouse Buttons

Obviously you can't do much with just the mouse position, so wouldn't it be helpful to also have the ability to detect mouse button clicks? You can do just that by using the `mouse_b` variable.

```
extern volatile int mouse_b;
```

This single integer variable contains the button values in packed bit format, where the first bit is button one, the second bit is button two, and the third bit is button three. If you want to check for a specific button, you can just use the `&` (logical and) operator to compare a bit inside `mouse_b`.

```
if (mouse_b & 1)
    printf("Left button was pressed");
if (mouse_b & 2)
    printf("Right button was pressed");
if (mouse_b & 4)
    printf("Center button was pressed");
```

Showing and Hiding the Mouse Pointer

Since an Allegro game will usually run in full-screen mode (or at least take over the entire window in windowed mode), you need a way to display a graphical mouse pointer. Anything other than the default operating system pointer is needed to really personalize a game. To facilitate this, Allegro provides the `set_mouse_sprite` function.

```
void set_mouse_sprite(BITMAP *sprite);
```

As you can see from the function definition, `show_mouse` needs a bitmap to display as the mouse pointer. Although I won't cover bitmaps and sprites until later (see Chapter 7, "Basic Bitmap Handling and Blitting," and Chapter 8, "Basic Sprite Programming"), you'll have to make some assumptions at this point and just go with the code. I will show you how to load a bitmap image and display it as the mouse pointer shortly, in the *Strategic Defense* game.

You can use a helper function after you call `set_mouse_sprite` to draw a graphical mouse pointer. The `set_mouse_sprite_focus` function adjusts the center point of the mouse cursor, with a default at the upper-left corner. If you are using a mouse pointer with another focal point, you can use this function to set that point within the mouse pointer.

```
void set_mouse_sprite_focus(int x, int y);
```

Of course, you are free to continue using the system mouse in windowed mode. Even in full-screen mode the mouse position is polled, but no mouse pointer is displayed.

When you are using a graphical mouse, you must tell the mouse handler where the mouse should be displayed. Remember that the pointer is just an image treated as a transparent sprite, so you have the option to draw the mouse directly to the screen or to any other bitmap (such as a secondary image used for double-buffering the screen). Use the `show_mouse` function to tell the mouse handler where you want the mouse pointer drawn.

```
void show_mouse(BITMAP *bmp);
```

Now what about hiding a graphical mouse once it's been drawn? This is actually a very important consideration because the mouse is basically treated as a transparent sprite, so it will interfere with the objects being drawn on the screen. Therefore, the mouse pointer needs to be hidden during screen updates, and then enabled again after drawing is completed. It's a bit of a pun that the function to hide the mouse pointer is called `scare_mouse`, and the function to show the mouse again is called `unscare_mouse`.

```
void scare_mouse();
void unscare_mouse();
```

There is also a version of this function that hides the mouse only if the mouse is within a certain part of the screen. If you know what part of the screen is being updated, you can use `scare_mouse_area` instead of `scare_mouse`, in which case the mouse simply will be frozen until you call `unscare_mouse` to re-enable it.

```
void scare_mouse_area(int x, int y, int w, int h);
```

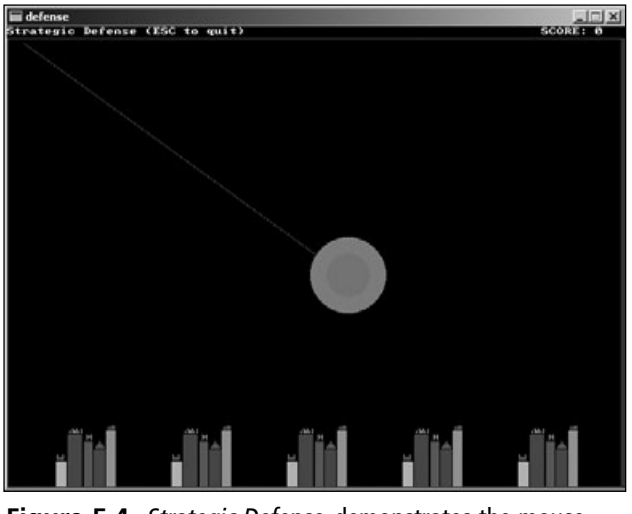


Figure 5.4 *Strategic Defense* demonstrates the mouse handler.

The Strategic Defense Game

I have written a short game to demonstrate how to use the basic mouse handler functions covered so far. This game is a derivation of the classic *Missile Command* and it is called *Strategic Defense*. The game uses the mouse position and the left mouse button to control a defense weapon to destroy incoming enemy missiles. Figure 5.4 shows a missile being destroyed.

The game features a graphical mouse pointer that is used as a targeting reticule, as shown in Figure 5.5.

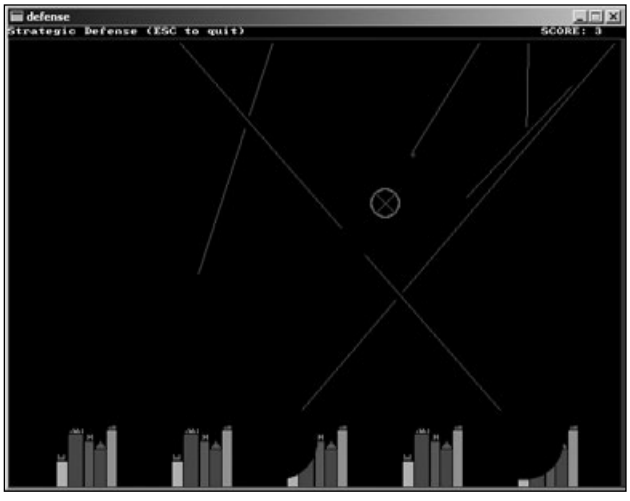


Figure 5.5 A graphical mouse pointer is used for targeting enemy missiles.

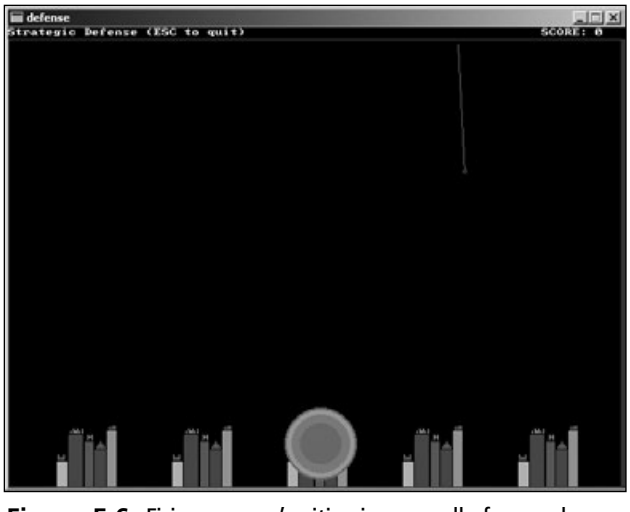


Figure 5.6 Firing on one's cities is generally frowned upon, but it is not destructive in this game.

When enemy missiles reach the ground (represented by the bottom of the screen) they will explode, taking out any nearby enemy cities.

One interesting thing about the game is how it uses a secondary screen buffer. Rather than writing extensive code to erase explosions and restore the mouse cursor, the game simply draws explosions directly on the screen rather than to the buffer (which contains the background image, including the game title and the cities). Thus, when the player fires directly on a city (as shown in Figure 5.6), that city remains intact because the explosion was drawn to the screen, while the buffer image remained intact. Perhaps it's not as realistic, but we don't want to destroy our own cities!

You might also notice in the figures that the game keeps track of the score in the upper-right corner of the screen. You gain a point for every enemy missile you destroy. Unfortunately, there is no ending to this game; it will keep running with an endless barrage of enemy missiles until you hit Esc to quit.

Type in the game's source code, and then have some fun! If you'd like to load the project off the CD-ROM, it is in the chapter08 folder and the file is called defense. This game might be overkill just to demonstrate the mouse, but it has some features that are helpful for the learning process, such as a real-time game loop, the use of bitmaps and sprites, and basic game logic. This game is far more complex than *Tank War*, but it is not without flaws. For one thing, the original *Missile Command* had multiple incoming enemy missiles

and allowed the player to target them, after which a missile would fire from turrets on the ground to take out the enemy missiles. These features would really make the game a lot more fun, so I encourage you to add them if you are so inclined.

Want a hint? You can add a dimension to the points array to support many "lines" for incoming missiles. How would you fire anti-ballistic missiles from the ground up to the mouse-click spot? Reverse-engineer the enemy missile code, add another array (perhaps something like `mypoints`), add another line callback function that doesn't interfere with the existing one, and reverse the direction (with the starting position at the bottom, moving upward toward the mouse click). When the friendly missile reaches the end of its line, it will explode. It's like adding an intermediate step between the time you press the mouse button and when the explosion occurs.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

//create some colors
#define WHITE makecol(255,255,255)
#define BLACK makecol(0,0,0)
#define RED makecol(255,0,0)
#define GREEN makecol(0,255,0)
#define BLUE makecol(0,0,255)
#define SMOKE makecol(140,130,120)

//point structure used to draw lines
typedef struct POINT
{
    int x,y;
}POINT;

//points array holds do_line points for drawing a line
POINT points[2000];
int curpoint,totalpoints;

//bitmap images
BITMAP *buffer;
BITMAP *crosshair;
BITMAP *city;

//misc variables
int x1,y1,x2,y2;
int done=0;
```

```
int destroyed=1;
int n;
int mx,my,mb;
int score = -1;

void updatescore()
{
    //update and display the score
    score++;
    textprintf_right(buffer,font,SCREEN_W-5,1,WHITE,
        "SCORE: %d  ", score);
}

void explosion(BITMAP *bmp, int x,int y,int finalcolor)
{
    int color,size;

    for (n=0; n<20; n++)
    {
        //generate a random color
        color = makecol(rand()%255,rand()%255,rand()%255);
        //random explosion size
        size = 20+rand()%20;
        //draw the random filled circle
        circlefill(bmp, x, y, size, color);
        //short pause
        rest(2);
    }
    //missile tracker looks for this explosion color
    circlefill(bmp, x, y, 40, finalcolor);
}


void doline(BITMAP *bmp, int x, int y, int d)
{
    //line callback function...fills the points array
    points[totalpoints].x = x;
    points[totalpoints].y = y;
    totalpoints++;
}

void firenewmissile()
{
```

```
    //activate the new missile
    destroyed=0;
    totalpoints = 0;
    curpoint = 0;

    //random starting location
    x1 = rand() % (SCREEN_W-1);
    y1 = 20;

    //random ending location
    x2 = rand() % (SCREEN_W-1);
    y2 = SCREEN_H-50;

    //construct the line point-by-point
    do_line(buffer,x1,y1,x2,y2,0,&doline);
}

void movemissile()
{
    //grab a local copy of the current point
    int x = points[curpoint].x;
    int y = points[curpoint].y;

    //hide mouse pointer
    scare_mouse();

    //erase missile
    rectfill(buffer,x-6,y-3,x+6,y+1,BLACK);

    //see if missile was hit by defense weapon
    if (getpixel(screen,x,y) == GREEN)
    {
        //missile destroyed! score a point
        destroyed++;
        updatescore();
    }
    else
    //no hit, just draw the missile and smoke trail
    {
        //draw the smoke trail
        putpixel(buffer,x,y-3,SMOKE);
        //draw the missile
        circlefill(buffer,x,y,2,BLUE);
```

```
    }

    //show mouse pointer
    unscare_mouse();

    //did the missile hit a city?
    curpoint++;
    if (curpoint >= totalpoints)
    {
        //destroy the missile
        destroyed++;
        //animate explosion directly on screen
        explosion(screen, x, y, BLACK);
        //show the damage on the backbuffer
        circlefill(buffer, x, y, 40, BLACK);
    }
}

void main(void)
{
    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();
    install_mouse();
    install_timer();
    srand(time(NULL));

    //create a secondary screen buffer
    buffer = create_bitmap(640,480);

    //display title
    textout(buffer,font,"Strategic Defense (ESC to quit)",0,1,WHITE);

    //display score
    updatescore();

    //draw border around screen
    rect(buffer, 0, 12, SCREEN_W-2, SCREEN_H-2, RED);

    //load and draw the city images
    city = load_bitmap("city.bmp", NULL);
```

```
    for (n = 0; n < 5; n++)
        masked_blit(city, buffer, 0, 0, 50+n*120,
            SCREEN_H-city->h-2, city->w, city->h);

    //load the mouse cursor
    crosshair = load_bitmap("crosshair.bmp", NULL);
    set_mouse_sprite(crosshair);
    set_mouse_sprite_focus(15,15);
    show_mouse(buffer);

    //main loop
    while (!key[KEY_ESC])
    {
        //grab the current mouse values
        mx = mouse_x;
        my = mouse_y;
        mb = (mouse_b & 1);

        //fire another missile if needed
        if (destroyed)
            firenewmissile();

        //left mouse button, fire the defense weapon
        if (mb)
            explosion(screen,mx,my,GREEN);

        //update enemy missile position
        movemissile();

        //update screen
        blit(buffer,screen,0,0,0,0,640,480);

        //pause
        rest(10);
    }

    set_mouse_sprite(NULL);
    destroy_bitmap(city);
    destroy_bitmap(crosshair);
    allegro_exit();

}
END_OF_MAIN();
```

Setting the Mouse Position

You can set the mouse position to any point on the screen explicitly using the `position_mouse` function.

```
void position_mouse(int x, int y);
```

This could be useful if you have a dialog on the screen and you want to move the mouse there automatically. You could also use `position_mouse` to create a tutorial for your game. (Show the player what to click by sliding the mouse around the screen using an array of coordinates, which could be captured by repeatedly grabbing the mouse position and storing the values.)

The *PositionMouse* program demonstrates how to use this function for an interesting effect. Moving the mouse over one location on the screen transports the mouse to another location. Figure 5.7 shows the program running. There are two wormholes, with a spaceship representing the mouse cursor. The only potentially confusing part of the program is the `mouseinside` function, so I'll give you a quick overview. This function checks to see whether the mouse is within the boundary of a rectangle passed to the function (x1, y1,x2,y2); it returns 1 (true) if the mouse is inside the rectangular area.

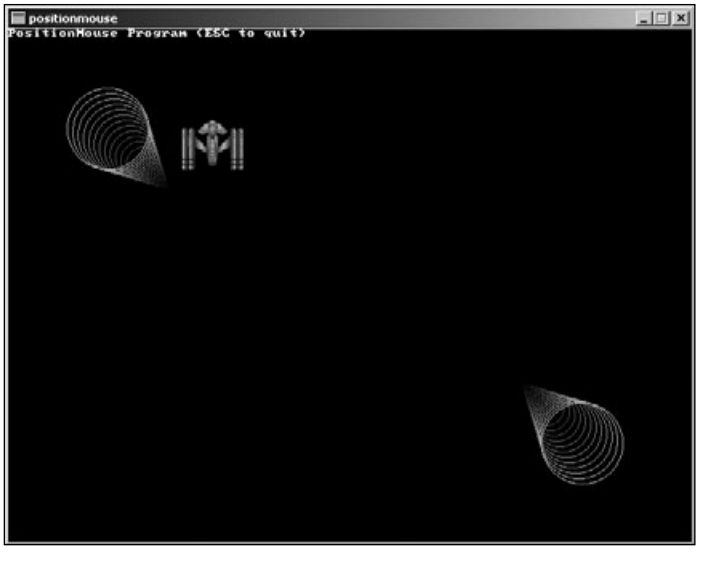


Figure 5.7 The *PositionMouse* program demonstrates the pros and cons of hyperspace travel. Ship image courtesy of Ari Feldman.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

int mouseinside(int x1,int y1,int x2,int y2)
{
    if (mouse_x > x1 && mouse_x < x2 && mouse_y > y1 && mouse_y < y2)
```

```
        return 1;
    else
        return 0;
}

void main(void)
{
    int n, x, y;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();
    install_mouse();
    textout(screen,font,"PositionMouse Program (ESC to quit)",0,0,WHITE);

    //load the custom mouse pointer
    BITMAP *ship = load_bitmap("spaceship.bmp", NULL);
    set_mouse_sprite(ship);
    set_mouse_sprite_focus(ship->w/2,ship->h/2);
    show_mouse(screen);

    //draw the wormholes
    for (n=0;n<20;n++)
    {
        circle(screen,150-3*n,150-3*n,n*2,makecol(10*n,10*n,10*n));
        circle(screen,480+3*n,330+3*n,n*2,makecol(10*n,10*n,10*n));
    }

    while (!key[KEY_ESC])
    {
        if (mouseinside(90,90,150,150))
            position_mouse(550,400);

        if (mouseinside(480,330,540,390))
            position_mouse(80,80);
    }
    set_mouse_sprite(NULL);
    destroy_bitmap(ship);
    allegro_exit();
}
END_OF_MAIN();
```

Limiting Mouse Movement and Speed

There are two helper functions that you will likely never use, but which are available nonetheless. The `set_mouse_range` function limits the mouse pointer to a specified rectangular region on the screen. Obviously, the default range is the entire screen, but you can limit the range if you want.

```
void set_mouse_range(int x1, int y1, int x2, int y2);
```

The second helper function is `set_mouse_speed`, which overrides the default mouse pointer speed set by the operating system. (Note that the mouse speed is not affected outside your program.) Greater values for the `xspeed` and `yspeed` parameters result in slower mouse movement. The default is 2 for each.

```
void set_mouse_speed(int xspeed, int yspeed);
```

Relative Mouse Motion

When it comes to game programming, relative mouse motion can be a very important feature at your disposal. Often, games will need to track the mouse movement without regard to the position of a pointer on the screen. Indeed, many games (especially first-person shooters) don't even have a mouse pointer; rather, they use the mouse to adjust the viewpoint of the player in the game world. This is called *relative mouse motion* because you can continue to move the mouse to the left (lifting the mouse and dragging it to the left again) over and over again, resulting in the game world spinning around the player continuously. Keep this in mind as you design your own games. The mouse need not be limited to the boundaries of the screen; it can return an infinite range of mouse movement.

```
void get_mouse_mickeys(int *mickeyx, int *mickeyy);
```

To use the mickeys returned by this function, you will want to create two integer variables to keep track of the last values, and then compare them to the new values returned by `get_mouse_mickeys`. You can then determine whether the mouse has moved up, down, left, or right, with the result having some effect in the game.

Using a Mouse Wheel

The mouse wheel is another great feature to support in your games. Although I would not assign any critical gameplay controls to the mouse wheel (because it might not be present), it is definitely a nice accessory of which you should take advantage when available. The mouse wheel is abstracted by Allegro into a simple variable that you can check at your leisure.

```
extern volatile int mouse_z;
```

Allegro provides a mouse wheel support function that seems rather odd at first glance, but it allows you to set the mouse wheel variable to a specific starting value, after which successive "reads" will result in values to and from that central value. In effect, `position_mouse_z` sets the current mouse wheel position as the starting position. Technically, the mouse wheel doesn't have starting and ending points because it is freewheeling.

```
void position_mouse_z(int z);
```

I have written a short program that demonstrates how to use the mouse wheel. The *MouseWheel* program doesn't really do anything; it displays a fictional throttle ramp (or any other lever, for that matter) and displays a small image that moves up or down based on the mouse wheel value. The program is shown in Figure 5.8. What was I thinking about when I wrote this program? I have no idea, and it makes no sense at all, does it? I think the idea is that this represents a reactor core temperature gauge, and if it goes critical, the reactor will explode. But you have your mouse wheel handy to prevent that. Your mouse *does* have a wheel, right? Perhaps you can turn this into a real game. I find it convenient to have one of those Microsoft Office keyboards with the big spinning wheel—yeah, you can spin that sucker like a top! Hey, why don't you turn this into an interesting game?

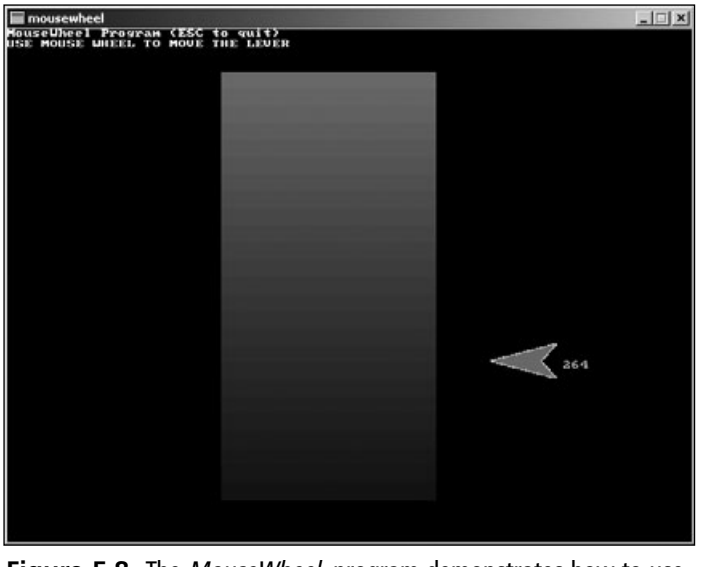


Figure 5.8 The *MouseWheel* program demonstrates how to use the mouse wheel. (Duh!)

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLACK makecol(0,0,0)
#define AQUA makecol(0,200,255)

void main(void)
```

```
{
    int n, color, value;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();
    install_mouse();
    textout(screen,font,"MouseWheel Program (ESC to quit)",0,0,WHITE);
    textout(screen,font,"USE MOUSE WHEEL TO MOVE THE LEVER",0,10,WHITE);

    //load the control lever image
    BITMAP *lever = load_bitmap("lever.bmp", NULL);

    //draw the throttle control
    for (n=0; n<200; n++)
    {
        color = makecol(255-n,10,10);
        rectfill(screen, 200, 40 + n * 2, 400, 42 + n * 2, color);
    }

    value=200;
    position_mouse_z(value);

    while (!key[KEY_ESC])
    {
        //erase the lever
        rectfill(screen, 450, 29 + value, 550, value + 65, BLACK);

        //update lever position
        value = mouse_z;
        if (value < 0)
            value = 0;
        if (value > 390)
            value = 390;

        //draw the lever
        blit(lever, screen, 0, 0, 450, 30 + value, lever->w, lever->h);

        //display value
        textprintf(screen, font, 520, 30 + value + lever->h / 2,
            AQUA,"%d", value);
```

```
        rest(30);
    }

    allegro_exit();
}
END_OF_MAIN();
```

Handling Joystick Input

Joysticks are not as common on the PC as they used to be, and the accessory controller market has fallen significantly since the late 1990s—to such a degree that Microsoft has dropped its Sidewinder line of gamepads and flight sticks (although at least one stick is still available from Microsoft to support its legendary *Flight Simulator* and *Combat Flight Simulator* products). I have personally been a Logitech fan for many years, and I appreciate the high quality of their mouse and joystick peripherals. The Logitech WingMan RumblePad is still my favorite gamepad because it has two analog sticks that make it useful for flight and space sims. In this section, I'll show you how to add joystick support to your bevy of new game development skills made possible with the Allegro library.

The Joystick Handler

At this point, it's becoming redundant, but we still have to initialize the joystick handler like we did for the keyboard and mouse. At least Allegro is consistent, which is not something that can be said about all libraries. The first function you need to learn is `install_joystick`.

```
int install_joystick(int type);
```

What is the `type` parameter, you might wonder? Actually, I have no idea, so I just plug random values into it to see what happens—so far with no result.

Just kidding! The `type` parameter specifies the type of joystick being used, while `JOY_TYPE_AUTODETECT` is currently the only supported value. Because Allegro abstracts the DirectInput library to provide a generic joystick controller interface, it provides functionality for supporting digital and analog buttons and sticks. If you ever need to remove the joystick handler, you can call `remove_joystick`.

```
void remove_joystick();
```

Allegro's joystick handler can handle at most four joysticks, which is more than I have ever seen in a single game. If you have written a game that needs more than four joysticks, let me know because I'd like to help you redesign the game! Seriously, what this really means is that you can use a driving wheel with foot pedals, which are usually treated as two joystick devices. To find out how many joysticks have been detected by Allegro, you can use `num_joysticks`.

```
extern int num_joysticks;
```

As was the case with the two previous hardware handlers, some systems do not support asynchronous interrupt handlers. However, this point is moot when it comes to joysticks, which must be polled. Here is the function:

```
int poll_joystick();
```

Remember, most (if not all) systems require you to poll the joystick because there is no automatic joystick interrupt handler running like there is for the keyboard and mouse handlers. Keep this in mind! If your joystick routine is not responding, it could be that you forgot to poll the joystick during the game loop!

tip

The joystick handler has no interrupt routine, so you must poll the joystick inside your game loop or the joystick values will not be updated. The keyboard and mouse usually do not need to be polled, but the joystick does need it!

This function fills the `JOYSTICK_INFO` struct, which has this definition:

```
typedef struct JOYSTICK_INFO
{
   int flags;
   int num_sticks;
   int num_buttons;
   JOYSTICK_STICK_INFO stick[n];
   JOYSTICK_BUTTON_INFO button[n];
} JOYSTICK_INFO;
```

Allegro defines an array to handle any joysticks plugged into the system based on this struct.

```
extern JOYSTICK_INFO joy[n];
```

The default joystick should therefore be `joy[0]`, which is what you will use most of the time if you are writing a game with joystick support.

Detecting Controller Stick Movement

The `JOYSTICK_INFO` struct contains two sub-structs, as you can see, and these sub-structs contain all of the actual joystick status information (analog/digital values). The `JOYSTICK_STICK_INFO` struct contains information about the sticks, which may be digital (such as an eight-way directional pad) or analog (with a range of values for position). Here is what that struct looks like:

```
typedef struct JOYSTICK_STICK_INFO
{
```

```
   int flags;
   int num_axis;
   JOYSTICK_AXIS_INFO axis[n];
   char *name;
} JOYSTICK_STICK_INFO;
```

I'll explain the `flags` element in a moment. For now, you need to know about `num_axis` and the `axis[n]` elements. `char *name` contains the name of the stick (if supported by your operating system's joystick driver). `num_axis` will tell you how many axes are provided by that stick. (Remember, there could be more than one stick on a joystick.) A normal stick will have two axes: X and Y. Therefore, most of the time `num_axis` will equal 2, and you will be able to read those axis values by looking at `axis[0]` and `axis[1]`. Some sticks are special types (such as a throttle control) that may only have one axis. If you are writing a large and complex game and you want to support as many joystick options as possible, you will want to look at all of these structs and their values to come up with a list of features available. For instance, if there are two sticks, and the first has two axes, while the second has one axis, it's a sure bet that this represents a flight-style joystick with a single stick and a throttle control. Obviously, for a large game it will be worth the time investment to create a joystick configuration option screen.

A single joystick might provide several different stick inputs (such as the two analog sticks on the Logitech WingMan RumblePad), but it is safe to assume that the first element in the stick array will always be the main directional stick. (Most joysticks have a single stick; the duals are the exception most of the time.)

Allegro really doesn't provide many support functions for decoding these structs—something that I found disappointing. However, the structs contain everything you need to read the joystick in real time, so there's no room for complaint as long as all the data is available. Besides, it's a far cry from programming a joystick using assembly language, as I did way back when—during the development of *Starship Battles*, which I talked about in Chapter 1.

Reading the Axes

To read the stick positions, you must take a look at the `JOYSTICK_AXIS_INFO` struct.

```
typedef struct JOYSTICK_AXIS_INFO
{
   int pos;
   int d1, d2;
   char *name;
} JOYSTICK_AXIS_INFO;
```

This struct provides one analog input (`pos`) and two digital inputs (`d1`, `d2`) that describe the same axis. While `pos` may contain a value of –128 to 128 (or 0 to 255, depending on the

type of axis), the d1 and d2 values will be 0 or 1, based on whether the axis was moved left or right. A digital stick will provide just a single yes or no type result using d1 and d2, but the analog values are more common.

Reading the Joystick Flags

I want to digress for a moment to talk about the joystick flags defined as flags in the JOYSTICK_STICK_INFO struct. Table 5.2 shows the possible values stored in flags as a bit mask.

Table 5.2 Joystick Bit Mask Values

Flag	Description
JOYFLAG_DIGITAL	This control is currently providing digital input.
JOYFLAG_ANALOG	This control is currently providing analog input.
JOYFLAG_CALIB_DIGITAL	This control will be capable of providing digital input once it has been calibrated, but it is not doing this at the moment.
JOYFLAG_CALIB_ANALOG	This control will be capable of providing analog input once it has been calibrated, but it is not doing this at the moment.
JOYFLAG_CALIBRATE	This control needs to be calibrated. Many devices require multiple calibration steps, so you should call the calibrate_joystick() function from a loop until this flag is cleared.
JOYFLAG_SIGNED	The analog axis position is in signed format, ranging from –128 to 128. This is the case for all 2D directional controls.
JOYFLAG_UNSIGNED	The analog axis position is in unsigned format, ranging from 0 to 255. This is the case for all 1D throttle controls.

Thus, if you want to know whether the specified stick is analog or digital, you can check the flags member variable.

```
if (flags & JOYFLAG_DIGITAL)
    printf("This is a digital stick");
```

Allegro provides a series of functions for calibrating a joystick; these are useful for older operating systems (such as MS-DOS) where calibration was necessary. Most modern joysticks are calibrated at the driver level. In Windows, go to Start, Settings, Control Panel and look for Gaming Options or Game Controllers to find the joystick dialog. Windows 2000 uses the Gaming Options dialog box, as shown in Figure 5.9.

Clicking on the Properties button opens the calibration and test dialog box, as shown in Figure 5.10.

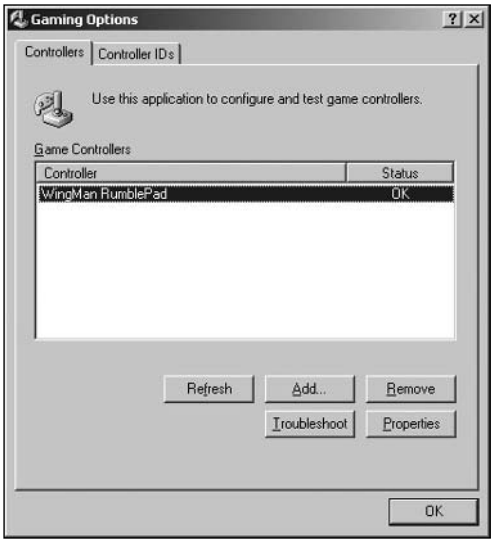


Figure 5.9 The Gaming Options dialog box in Windows 2000

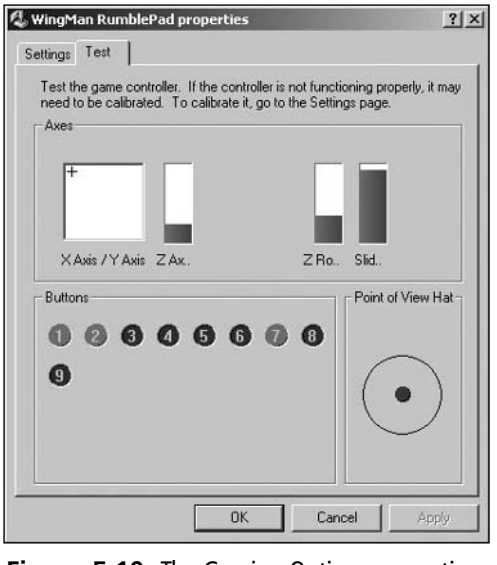


Figure 5.10 The Gaming Options properties dialog box for my WingMan RumblePad joystick

Using the Properties dialog box, you can verify that the joystick is operating (first and foremost) and that all the buttons and sticks are functioning.

Under Windows XP, the control panel applet for configuring your joystick seems to be about 12 levels deep inside the operating system, like an epithermal vein in the earth. For this reason, I recommend switching the Control Panel to Classic View so you can see exactly what you want without wading through Microsoft's patronizing interface. As a follower of the philosophies of Alan Cooper, my personal opinion is that too much interface is condescending. ("Hello sir. I believe you are too stupid to figure this out, so let me bury it for you.") However, I do appreciate and enjoy most of Microsoft's latest products—this company does get it right after eight or nine versions. It's all a matter of personal preference, though. Wouldn't you agree?

I digress again. Windows XP provides a similar applet called Game Controllers, with a similar joystick properties dialog box you can use to test your joystick. (In most cases, calibration is not needed with modern USB joysticks.)

Detecting Controller Buttons

Referring back to the primary joystick struct, `JOYSTICK_INFO`, you'll recall that the second sub-struct is called `JOYSTICK_BUTTON_INFO`.

```
JOYSTICK_BUTTON_INFO button[n];
```

This struct can be read with the help of `num_buttons` to determine the size of the button array.

```
int num_buttons;
```

The final struct you need to see to deal with joystick buttons has this definition:

```
typedef struct JOYSTICK_BUTTON_INFO
{
   int b;
   char *name;
} JOYSTICK_BUTTON_INFO;
```

The `b` element will simply be 0 or 1, based on whether the button is being pushed or not, while `char *name` describes that button.

Testing the Joystick Routines

I could call it a wrap at this point, but what I'd like to do now is provide two sample programs that demonstrate how to use the joystick routines. The first sample program, *ScanJoystick*, iterates through these structs to print out information about the joystick. The second program, *TestJoystick*, is a simple example of how to use the joystick in a real-time program.

The ScanJoystick Program

The *ScanJoystick* program goes through the joystick structs and prints out logistical information, including number of sticks, stick names, number of buttons, and button names. The output from the program is shown in Figure 5.11.

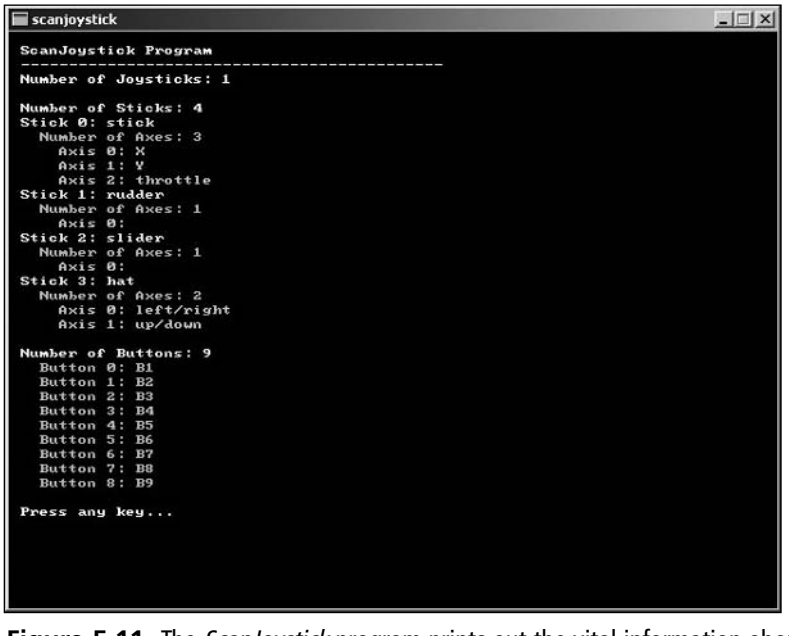


Figure 5.11 The *ScanJoystick* program prints out the vital information about the first joystick device.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define LTGREEN makecol(192,255,192)
#define LTRED makecol(255,192,192)
#define LTBLUE makecol(192,192,255)
int curline = 1;

void print(char *s, int color)
{
    //print text with automatic linefeed
    textout(screen, font, s, 10, (curline++) * 12, color);
}

void printjoyinfo()
{
    char *s;
    int n, ax;

    //display joystick information
    sprintf(s, "Number of Joysticks: %d", num_joysticks);
    print(s, WHITE);
    print("",0);

    //display stick information
    sprintf(s, "Number of Sticks: %d", joy[0].num_sticks);
    print(s, LTGREEN);
    for (n=0; n<joy[0].num_sticks; n++)
    {
        sprintf(s, "Stick %d: %s", n, joy[0].stick[n].name);
        print(s, LTGREEN);
        sprintf(s, "  Number of Axes: %d", joy[0].stick[n].num_axis);
        print(s, LTBLUE);
        for (ax=0; ax<joy[0].stick[n].num_axis; ax++)
        {
            sprintf(s,"    Axis %d: %s", ax,
                joy[0].stick[n].axis[ax].name);
            print(s, LTRED);
        }
    }
```

```
    //display button information
    print("",0);
    sprintf(s, "Number of Buttons: %d", joy[0].num_buttons);
    print(s, LTGREEN);
    for (n=0; n<joy[0].num_buttons; n++)
    {
        sprintf(s,"  Button %d: %s", n, joy[0].button[n].name);
        print(s, LTBLUE);
    }
}

void main(void)
{
    int n, color, value;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();

    //install the joystick handler
    install_joystick(JOY_TYPE_AUTODETECT);
    poll_joystick();

    //display title
    print("ScanJoystick Program", WHITE);
    print("----------------------", WHITE);

    //look for a joystick
    if (num_joysticks > 0)
        printjoyinfo();
    else
        print("No joystick could be found", WHITE);

    //pause and exit
    print("",0);
    print("Press any key...", WHITE);
    while (!keypressed()) { }
    allegro_exit();
}

END_OF_MAIN();
```

The TestJoystick Program

To really see what Allegro's joystick routines can do would require a full-blown game using the sticks for movement and the buttons for perhaps firing weapons. Hey, it sounds like *Tank War* would be a great candidate for just that! But for the time being, it is prudent to focus on a simple joystick demonstration that simply makes clear what you must do to get basic joystick support into your games. Thus, I have written a quick-and-dirty game with a functional name; it's just a ball bouncing around on the screen, with a paddle that is controlled by the joystick. Stop the ball from hitting the floor and gain a point; fail to stop the ball and lose a point. It's a very simple game in that respect. However, this game does use several bitmaps and blitting routines (including `masked_blit` to draw transparently). Unfortunately, these routines have not been explained yet, and I'm loath to do so now, when an entire chapter is dedicated to this subject! (See Chapter 7 for a complete explanation of how to use the bitmap loading and blitting functions.) For now, I would like to leave that discussion for Chapter 7 and just use this functionality to make the game more interesting.

Figure 5.12 shows this very simple and limited *Arkanoid*/*Breakout* style game in action. Again, I am indebted to Ari Feldman for the artwork (http://www.arifeldman.com), which comes from his free SpriteLib collection. The source code is only a few pages long, so I'll leave it to you to read my code comments and see how it works. I hope the game is simple enough that you will find it very easy to read the code and learn some new tricks from it.

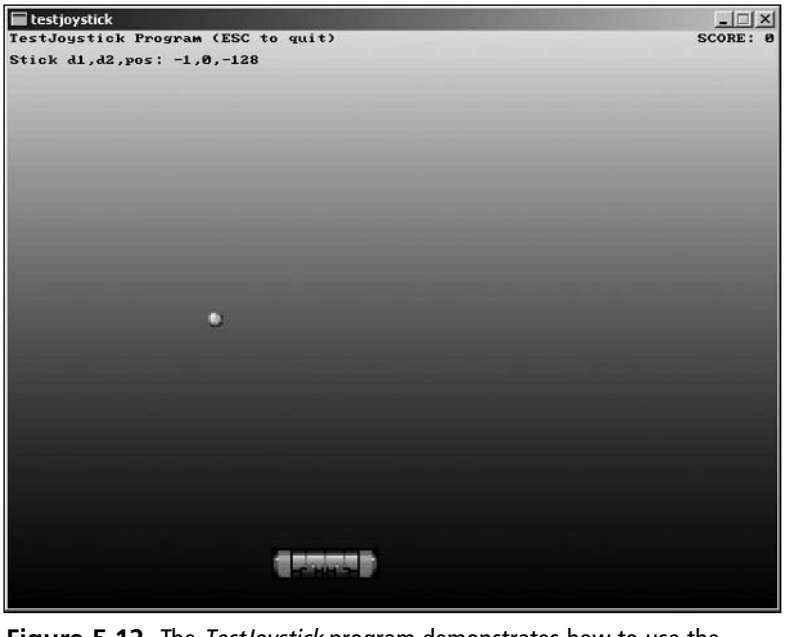


Figure 5.12 The *TestJoystick* program demonstrates how to use the joystick in a simple game.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLACK makecol(0,0,0)

BITMAP *back;
BITMAP *paddle;
BITMAP *ball;

int score = 0, paddlex, paddley = 430;
int ballx = 100, bally = 100;
int dirx = 1, diry = 2;

void updateball()
{
    //update ball x
    ballx += dirx;

    //hit left?
    if (ballx < 0)
    {
        ballx = 1;
        dirx = rand() % 2 + 4;
    }

    //hit right?
    if (ballx > SCREEN_W - ball->w - 1)
    {
        ballx = SCREEN_W - ball->w - 1;
        dirx = rand() % 2 - 6;
    }

    //update ball y
    bally += diry;

    //hit top?
    if (bally < 0)
    {
        bally = 1;
        diry = rand() % 2 + 4;
    }
```

```
    //hit bottom?
    if (bally > SCREEN_H - ball->h - 1)
    {
        score--;
        bally = SCREEN_H - ball->h - 1;
        diry = rand() % 2 - 6;
    }

    //hit the paddle?
    if (ballx > paddlex && ballx < paddlex+paddle->w &&
        bally > paddley && bally < paddley+paddle->h)
    {
        score++;
        bally = paddley - ball->h - 1;
        diry = rand() % 2 - 6;
    }

    //draw ball
    masked_blit(ball, screen, 0, 0, ballx, bally, ball->w, ball->h);
}

void main(void)
{
    int d1, d2, pos, startpos;

    //initialize program
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    srand(time(NULL));
    install_keyboard();

    //install the joystick handler
    install_joystick(JOY_TYPE_AUTODETECT);
    poll_joystick();

    //look for a joystick
    if (num_joysticks == 0)
    {
        textout(screen,font,"No joystick could be found",0,20,WHITE);
        while(!keypressed());
        return;
    }
```

```
//store starting stick position
startpos = joy[0].stick[0].axis[0].pos;

//load the background image
back = load_bitmap("background.bmp", NULL);

//load the paddle image and position it
paddle = load_bitmap("paddle.bmp", NULL);
paddlex = SCREEN_W/2 - paddle->w/2;

//load the ball image
ball = load_bitmap("ball.bmp", NULL);

//set text output to transparent
text_mode(-1);

//main loop
while (!key[KEY_ESC])
{
    //clear screen the slow way (redraw background)
    blit(back, screen, 0, 0, 0, 0, back->w, back->h);

    //update ball position
    updateball();

    //read the joystick
    poll_joystick();
    d1 = joy[0].stick[0].axis[0].d1;
    d2 = joy[0].stick[0].axis[0].d2;
    pos = joy[0].stick[0].axis[0].pos;

    //see if stick moved left
    if (d1 || pos < startpos+10) paddlex -= 4;
    if (paddlex < 2) paddlex = 2;

    //see if stick moved right
    if (d2 || pos > startpos-10) paddlex += 4;
    if (paddlex > SCREEN_W - paddle->w - 2)
        paddlex = SCREEN_W - paddle->w - 2;

    //display text messages
    textout(screen, font, "TestJoystick Program (ESC to quit)",
        2, 2, BLACK);
```

```
        textprintf(screen, font, 2, 20, BLACK,
            "Stick d1,d2,pos: %d,%d,%d", d1, d2, pos);
        textprintf_right(screen, font, SCREEN_W - 2, 2, BLACK,
            "SCORE: %d", score);

        //draw the paddle
        blit(paddle,screen,0,0,paddlex,paddley,paddle->w,paddle->h);

        rest(20);
    }

    destroy_bitmap(back);
    destroy_bitmap(paddle);
    destroy_bitmap(ball);
    return;
}

END_OF_MAIN();
```

Summary

I don't know about you, but I got more from this chapter than I had intended! There were many new functions presented in this chapter, with absolutely no explanation for some of them! I'm talking about `load_bitmap`, `blit`, `masked_blit`, and so on. That is breaking a rule I had intended to follow about only using what I have covered thus far; however, I think it's a helpful learning experience to see some of what is to come.

This chapter presented Allegro's input routines and explained how to read the keyboard, mouse, and joystick—which, it turns out, is not difficult at all thanks to the way in which Allegro abstracted these hardware input devices.

The big question you might have is, why didn't we update *Tank War* to support a joystick? That's a good question. As a matter of fact, I wanted to plug in the joystick support at this point, but I felt that it would make the game too complicated this early along in the book, when the goal is really to demonstrate each chapter's new graphics features in the game. In a nutshell, the game is just too primitive and underdeveloped at this point to warrant joystick support. Therefore, I make this vow: We will add joystick support to *Tank War* in a future chapter. I guarantee it!

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Which function is used to initialize the keyboard handler?
   A. `initialize_keyboard`
   B. `install_keyboard`
   C. `init_keyboard`
   D. `install_keyboard_handler`

2. What does ANSI stand for?
   A. American Negligible Situation Imperative
   B. American Nutritional Studies Institute
   C. American National Standards Institute
   D. American National Scuba Institute

3. What is the name of the array containing keyboard scan codes?
   A. `key`
   B. `keyboard`
   C. `scancodes`
   D. `keys`

4. Where is the real stargate located?
   A. Salt Lake City, Utah
   B. San Antonio, Texas
   C. Colorado Springs, Colorado
   D. Cairo, Egypt

5. Which function provides buffered keyboard input?
   A. `scankey`
   B. `getkey`
   C. `readkey`
   D. `buffered_input`

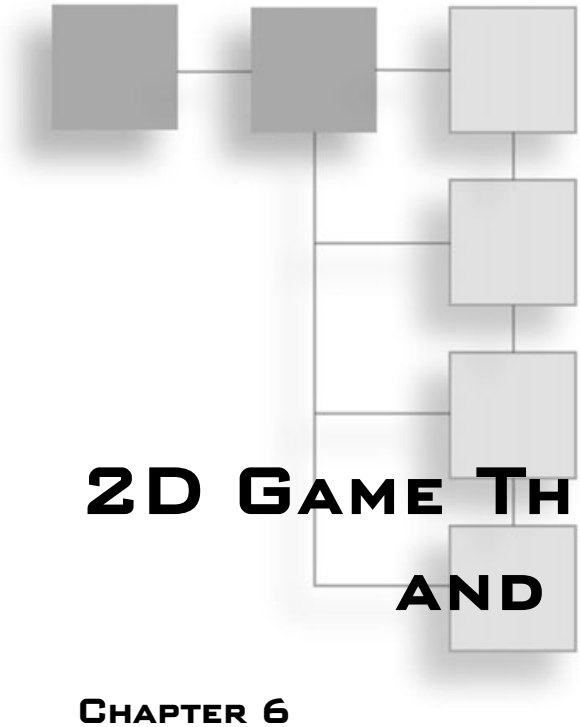
6. Which function is used to initialize the mouse handler?
   A. `install_mouse`
   B. `instantiate_mouse`
   C. `initialize_mouse`
   D. `ingratiate_mouse`

7. Which values or functions are used to read the mouse position?
   A. `mouse_x` and `mouse_y`
   B. `get_mouse_x` and `get_mouse_y`
   C. `mousex` and `mousey`
   D. `mouse_position_x` and `mouse_position_y`

8. Which function is used to read the mouse x and y mickeys for relative motion?
   A. `mickey_mouse`
   B. `read_mouse_mickeys`
   C. `mouse_mickeys`
   D. `get_mouse_mickeys`

9. What is the name of the main `JOYSTICK_INFO` array?
   A. `joysticks`
   B. `joy`
   C. `sticks`
   D. `joystick`

10. Which struct contains joystick button data?
    A. `JOYSTICK_BUTTONS`
    B. `JOYSTICK_BUTTON`
    C. `JOYSTICK_BUTTON_INFO`
    D. `JOYSTICK_BUTTON_DATA`

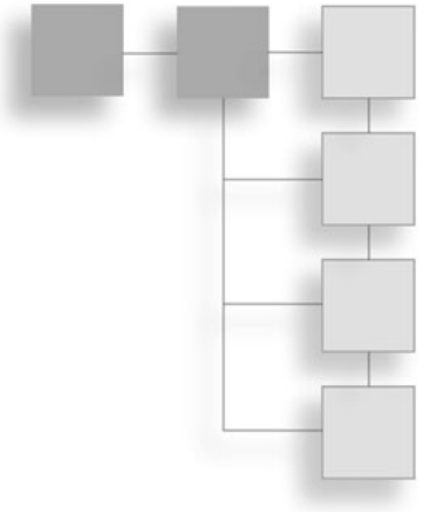


PART II

2D Game Theory, Design, and Programming

Welcome to Part II of *Game Programming All in One, 2nd Edition*. Part II includes nine chapters that form the bulk of the crucial game programming subjects of the book. In this section, you will learn about game design, bitmaps, blitting, basic sprite handling, advanced sprite programming, tile-based backgrounds, background scrolling, timers, interrupt handling, multi-threading, and map editors. This section also includes chapters on vertical and horizontal scrolling games!

CHAPTER 6



Introduction to Game Design

Many years have passed since the days when games were designed in a couple of hours at the family barbeque. Today, games are required to be fun and addictive, but at the same time meaningful and intuitive. The latest games released by the big companies take months to design—and that is with the help of various designers. Contrary to popular belief, a game designer's sole purpose isn't to think of an idea, and then give it to the programmers so they can make a game. A designer must think of the game idea, elaborate it, illustrate it, define it, and describe just about everything from the time the CD is inserted into the CD-ROM drive to the time when the player quits the game. This chapter will help you understand a little more about game design, as well as give you some tips about it, and in the end show you a small game design document for a very popular game.

This chapter goes over the software development process for game development, describing the main steps involved in taking a game from inspiration to completion. This is not about creating a hundred-page document with all the screens, menus, characters, settings, and storyline of the game. Rather, this chapter is geared toward the programmer, with tips for following a design process that keeps a game in development until it is completed. Without a simple plan, most game programmers will become bored with a game that only weeks before had them up all night with earnest fervor.

Here is a breakdown of the major topics in this chapter:

- Understanding game design basics
- Understanding game development phases
- Recognizing post-production woes
- Future-proofing your design
- Understanding the dreaded design document

- Recognizing the importance of good game design
- Recognizing the two types of designs
- Looking at a sample design document
- Looking at a sample game design: *Space Invaders*

Game Design Basics

Creating a computer game without at least a minimal design document or a collection of notes is like building a plastic model of a car or airplane from a kit without the instructions. More than likely, a game that is created without the instructions will end up with missing pieces and loose ends. The tendency is always to jump right in and get some code working, and there is room for that step in the development of a new game! However, the initial coding session should be used to build enthusiasm for yourself and any others in the project, and should be nothing more than a proof of concept or an incomplete prototype.

Like most creative individuals, game programmers must have the discipline to stick to something before moving on, or they will fall into the trap of half-completing a dozen or so games without much to show for the effort. The real difference between a hobbyist and a professional is simply that a professional game programmer will see the game through to completion no matter how long it takes.

Inspiration

Take a look at classic console games for inspiration, and you will have no trouble coming up with an idea for a cool new game. Any time I am bug-eyed and brain-dead from a long coding session, I take a walk outside and then return for a gaming session (preferably with a friend). Consoles are great because they are usually fast paced and they are often based on arcade machines.

PC games, on the other hand, can be quite slow and even boring in comparison because they have so much more depth. If you are in a hurry and just want to have a little fun, go with a console. Video games are full of creativity and interesting technology that PC gamers usually fail to notice.

Game Feasibility

The feasibility of a game is difficult to judge because so much is possible once you get started, and it is easy to underestimate your own capabilities (especially if you have a few people helping you). One thing that you must be careful about when designing a new game is scope. How big will the game be? You also don't want to bite off more than you can chew, to use the familiar expression.

Feasibility is the process of deciding how far you will go with the game and at what point you will stop adding new features. But at the very least, feasibility is a study of whether the game can be created in the first place. Once you are certain that you have the capability to create a certain type of game and you have narrowed the scope of the game to a manageable level, work can begin.

Feature Glut

As a general rule, you should get the game up and running before you work on new features (the bells and whistles). Never spend more than a day working on code without testing it! This is critical. Any time you change a major part of the game, you must completely recompile the game and run it to make sure you haven't broken any part of it in the process.

I can't tell you how many times I have thought up a new way to do something and gone through all the source code for a game, making changes here and there, with the result being that the game won't compile or won't run. Every change you make during the development of the game should be small and easy to undo if it doesn't work.

My personal preference is to keep the game running throughout the day. Every time I make even a minor change, I test the game to make sure it works before I move on to a new section of code. This is really where object-oriented coding pays off. By moving tried-and-tested code into a class, it is relatively safe to assume the code works as expected because you are not modifying it as often.

It really helps to eliminate bugs when you have put the startup and shutdown code inside a class (or within the Allegro library, where the library handles these routines automatically). There is also another tremendous benefit to wrapping code inside a library—information overload.

There is a point at which we humans simply can't handle any more information, and our memory starts to fail. If you try to keep track of too many loose ends in a game, you are bound to make a mistake! By putting common code in classes (or in a separate library file altogether), you reduce the amount of information that you must remember. It is such a relief when you need to do something quickly and you realize that all the code is ready to do your bidding at a moment's notice. The alternative (and old-school) method of copying sections of code and pasting them into your project is error-prone and will introduce bugs into your game.

Back Up Your Work

Follow this simple advice or learn the hard way: Back up your work several times a day! If you don't, you are going to make a significant change to the game code that completely breaks it and you will not be able to figure out how to get the game back up and running.

This is the point at which you return to a backup and start again. Even if the backup is a few hours old, it is better than spending half a day figuring out the problem with the changes you made.

I have an informal method of backing up my work. I use an archive program to zip the entire project directory for a game—including all the graphics, sounds, and source code—into a file with the date and time stamped into the filename, such as Game_070204_1030.zip.

The backup file might be huge, but what is disk space today? You don't always need to back up the entire project directory if you haven't made any changes to the graphics or sound files for the game (which can be quite large). If you are working on source code for days at a time without making any changes to media files, you might just make a complete backup once a day, and then make smaller backups during the day for code changes. As a general rule, I don't use the incremental backup feature available with many compression programs because I prefer to create an entirely new backup file each time.

If you get into the habit of backing up your files every hour or two, you will not be faced with the nightmare of losing a whole day's work if you mess up the source code or if something happens to your hard drive. For this reason, I recommend that you copy all backup files directly to CD (using packet-writing technology, which provides drag-and-drop capability from Windows Explorer). CD-RW drives are very affordable and indispensable when it comes to saving your work, giving you the ability to quickly erase and save your work repeatedly. DVD burners are also very affordable now, and they offer enough storage space to easily back up everything in an entire game project.

In the end, how much is your time worth? Making regular backups is the smart thing to do.

Game Genres

The gaming press seems to differentiate between console and PC games, but the line that separates the two is diminishing as games are ported back and forth. I tend to group console and PC games together in shared genres, although some genres do not work well on both platforms. The following sections detail a number of game genres; they contain a description of each genre and a list of sample games within that genre. It is important to consider the target genre for your game because this affects your target audience. Some gamers are absolutely fanatical about first-person shooters, while others prefer real-time strategy, and so on. It is a good idea to at least identify the type of game you are working on, even if it is a unique game.

Fighting Games

2D and 3D fighting games are almost entirely bound to the console market due to the way these games are played. A PC equipped with a gamepad is a fine platform, but fighting games really shine on console systems with multiple controllers.

One of my favorite Dreamcast fighting games is called *Power Stone 2*. This game is hilarious! Four players can participate on varied levels in hand-to-hand combat, with numerous obstacles and miscellaneous items strewn about on each level, and the action is fast paced.

Here is a list of my favorite fighting game series:

- *Dead or Alive*
- *Mortal Kombat*
- *Power Stone*
- *Ready 2 Rumble*
- *Soul Calibur*
- *Street Fighter*
- *Tekken*
- *Virtua Fighter*

Action/Arcade Games

Action/arcade games turned the fledgling video game industry into a worldwide phenomenon in the 1980s and 1990s, but started to drop off in popularity in arcades in the 2000s. The action/arcade genre encompasses a huge list of games, and here are some of my favorites:

- *Akari Warriors*
- *Blasteroids*
- *Elevator Action*
- *Rolling Thunder*
- *Spy Hunter*
- *Star Control*
- *Super R-Type*
- *Teenage Mutant Ninja Turtles*

Adventure Games

The adventure game genre was once comprised of the largest collection of games in the computer game industry, with blockbuster hits like *King's Quest* and *Space Quest*. Adventure games have fallen out of style in recent years, but there is still an occasional new adventure game that inspires the genre to new heights. For instance, *Starflight III: Mysteries of the Universe* is an official sequel to the original *Starflight* games, with a fantastic galaxy-spanning adventure story with the engaging mystery of space exploration. I am a member of the development team for this game, so I am naturally biased to appreciate

the game. For more information, visit http://www.starflight3.net. My definition of adventure game might differ from someone else's, but most of the following games may be categorized as adventure games:

- *King's Quest*
- *Mean Streets*
- *Myst*
- *Space Quest*
- *Starflight*

First-Person Shooters

The first-person shooter genre is the dominant factor in the gaming industry today, with so many new titles coming out every year that it is easy to overlook some extremely cool games while playing some others. This list is by no means complete, but it includes the most common first-person shooters:

- *Doom*
- *Half-Life*
- *Jedi Knight*
- *Max Payne*
- *Quake*
- *Unreal*
- *Wolfenstein 3-D*

Flight Simulators

Flight simulators (flight sims) are probably the most important type of game in the industry, although they are not always recognized as such. When you think about it, the technology required to render the world is quite a challenge. The best of the best in flight sims usually push the envelope of realism (pun intended) and graphical wizardry. Here is my list of favorites, old and new:

- *Aces of the Pacific*
- *B-17*
- *Battlehawks 1942*
- *Falcon 4.0*
- *Jane's WWII Fighters*
- *Red Baron*

Galactic Conquest Games

Galactic conquest games have seen mixed success at various times, with a popular title about once a year. One early success was a game called *Stellar Crusade*, which focused heavily on the economics of running a galactic empire. This may be debatable, but I believe that *Master of Orion* popularized the genre, while *Master of Orion II* perfected it. Even today, *MOO2* (as it is fondly referred to) still holds its own against modern wonders, such as *Imperium Galactica II*.

- *Imperium Galactica*
- *Master of Orion*
- *Stellar Crusade*

Real-Time Strategy Games

Real-time strategy (RTS) games are second only to first-person shooters in popularity and success, with blockbuster titles selling in the millions. Westwood is generally given kudos for inventing the genre with *Dune II*, although the *Command & Conquer* series gave the genre a lot of mileage. *Warcraft* and *Starcraft* (both by Blizzard) were huge in their time and are still popular today. My personal favorites are *Age of Empires* and the follow-up games in the series. Here are the best RTS games on the market today:

- *Age of Empires*
- *Command & Conquer*
- *Dark Reign*
- *MechCommander*
- *Real War*
- *Starcraft*
- *Total Annihilation*
- *Warcraft*

Role-Playing Games

What would the computer industry be without role-playing games? RPGs go back as far as most gamers can remember, with early games such as *Ultima* and *Might and Magic* appearing on some of the earliest PCs. *Ultima Online* followed in the tradition of *Meridian 59* as a massively multiplayer online role-playing game (MMORPG), along with *EverQuest* and *Asheron's Call*. Here are some classic favorites:

- *Baldur's Gate: Dark Alliance*
- *Darkstone*
- *Diablo*

- *Fallout*
- *Forgotten Realms*
- *Might and Magic*
- *The Bard's Tale*
- *Ultima*

Sports Simulation Games

Sports sims have long held a strong position in the computer game industry as a mainstay group of products covering all the major sports themes—baseball, football, soccer, basketball, and hockey. Here are some of my favorites:

- *Earl Weaver Baseball*
- *Madden 2004*
- *Wayne Gretzky and the NHLPA All-Stars*
- *World Series Baseball 2K3*

Third-Person Shooters

The third-person shooter genre was spawned by first-person shooters, but it sports an "over the shoulder" viewpoint. *Tomb Raider* is largely responsible for the popularity of this genre. Here are some favorite third-person shooters:

- *Delta Force*
- *Tom Clancy's Rainbow Six*
- *Resident Evil*
- *Tomb Raider*

Turn-Based Strategy Games

Turn-based strategy (TBS) games have a huge fan following because this genre allows for highly detailed games based on classic board games, such as *Axis & Allies*. Because TBS games do not run in real time, each player is allowed time to think about his next move, providing for some highly competitive and long-running games. Here is a list of the most popular games in the genre:

- *Axis & Allies*
- *Panzer General*
- *Shogun: Total War*
- *Steel Panthers*
- *The Operational Art of War*

Space Simulation Games

Space sims are usually grand in scope and provide a compelling story to follow. Based loosely on movies such as *Star Wars*, space sims usually feature a first-person perspective inside the cockpit of a spaceship. Gameplay is similar to that of a flight sim, but with science fiction themes. Here is a list of popular space sims:

- *Tachyon: The Fringe*
- *Wing Commander*

Real-Life Games

Real-life sims are affectionately referred to as *God games*, although the analogy is not perfect. How do you categorize a game like *Dungeon Keeper*? Peter Molyneux seems to routinely create his own genres. These games usually involve some sort of realistic theme, although it may be based on fictional characters or incidents. Here are some of the most popular real-life games:

- *Black & White*
- *Dungeon Keeper*
- *Populous*
- *SimCity*
- *The Sims*
- *Tropico*

Massively Multiplayer Online Games

I consider this a genre of its own, although the games herein may be categorized elsewhere. The most popular online games are called MMORPGs—massively multiplayer online role-playing games. This convoluted phrase describes an RPG that you can play online with hundreds or thousands of players—at least in theory.

- *Anarchy Online*
- *Asheron's Call*
- *Conquest: Frontier Wars*
- *EverQuest*
- *Ultima Online*
- *Final Fantasy Online*

Game Development Phases

Although there are entire volumes dedicated to software development life cycles and software design, I am going to cover only the basics that you will need to design a game. You

might want to go into finer detail with your game designs, or you might want to skip a few steps. It is all a matter of preference. But the important thing is that you at least attempt to document your ideas before you get started on a new game.

Initial Design

The initial design for a game is usually a hand-drawn figure showing what the game screen will look like, with the game's user interface or game elements shown in roughly the right places on the sketched screen. You can also use a program such as Visio to create your initial design screens.

The initial design should also include a few pages with an overview of the components needed by the game, such as the DirectX components or any third-party software libraries. You should include a description of how the game will be played and what forms of user input will be supported, and you should describe how the graphics will be rendered (in 2D or 3D).

Game Engine

Once you have an initial design for the game down on paper, you can get started on the game engine. This will usually be the most complicated core component of the game, such as the graphics renderer.

In the case of a 2D sprite-based game, the game engine will be a simple game loop with a double-buffer, a static or rendered background, and a few sprites moving around for good measure. If the game runs in real time, you will want to develop the collision detection routine and start working on the physics for the game.

By the end of this phase in development—before you get started on a real prototype—you should try to anticipate (based on the initial design) some of the possible graphics and miscellaneous routines you will need later. Obviously, you will not know in advance all of the functionality the game will need, but you should at least code the core routines up front.

Alpha Prototype

After you have developed the engine that will power your game, the next natural step in development is to create a prototype of the game. This phase is really a natural result of testing the game engine, so the two phases are often seamless. But if you treat the prototype as a single complete program without the need for modification, then you will have recognized this phase of the game.

Once you have finished the prototype, I recommend you compile and save it as an individual program or demo. At this point, you might want to send it to a few friends to get some feedback on general gameplay. This version of the game will not even remotely look as if it is complete. Bitmaps will be incomplete, and there might not even be any sound or music in the prototype.

However, one thing that the multiplayer prototype must have from the start is network capabilities. If you are developing a multiplayer game, you must code the networking along with the graphics and the game engine early in development. It is a mistake to start adding multiplayer code to the game after it is half finished, because most likely you will have written routines that are not suited for multiple players and you will have to rewrite a lot of code.

Game Development

The game development phase is clearly the longest phase of work done on a game. It consists of taking the prototype code base—along with feedback received by those who ran the demo—and building the game. Since this phase is the most important one, there are many different ways that you can accomplish it. First, you will most likely be building on the prototype that you developed in the previous phase because it usually does not make sense to start over from scratch unless there are some serious design flaws in the prototype.

You might want to stub out all of the functionality needed to complete the game so there is at least some sort of minimal response from the game when certain things happen or when a chain of events occurs. For instance, if you plan to support a high-score server on the Internet, you might code the high-score server with a simple response message so you can send a request to the server and then display the reply. This way, there is at least some sort of response from this part of the game, even if you do not intend to complete it until later.

Another positive note for stubbing out functionality is that you get to see the entire game as it will eventually appear when completed. This allows you to go back to the initial design phase and make some changes before you are half finished with the game. Stubbing out nonessential functionality lets you see an overview of the entire game. You can then freeze the design and complete each piece of the game individually until the game is finished.

Quality Control

Individuals like you who are working on a game alone might be tempted to skip some of the phases of development, since the formality of it might seem humorous. But even if you are working on a game by yourself, it is a good practice to get into the habit of going through the motions of the formal game development life cycle as if you have a team of people working with you on the game. Someday, you might find yourself working on a professional game with others, and the professionalism that you learned early on will pay off later.

Quality control is the formal testing process that is required to correct bugs in a game. Because the lead developers of a game have been staring at the code and the game screens for months or years, a fresh set of eyes is needed to properly test a game. If you are working solo, you need to recruit one or more friends to help you test the game. I guarantee that they will be able to find problems that you have overlooked or missed completely.

Because this is your pet project, you are very likely to develop habits when playing the game, while anyone else might find your machinations rather strange. Goofy keyboard shortcuts or strange user interface decisions might seem like the greatest thing since ketchup to you, but to someone else the game might not even be fun to play.

Consider quality control as an audit of your game. You need an objective person to point out flaws and gameplay issues that might not have been present in the prototype. It is a critical step when you think about it. After all the work you have put into a game, you certainly don't want a simple and easily correctable bug to tarnish the impression you want your game to have on others.

Beta Testing

Beta testing is a phase that follows the completion of the game's development phase, and it should be recognized as significantly separate from the previous quality control phase. The beta version of a game absolutely should not be released if the game has known bugs. Any time you send out a game for beta testing and you know there are bugs, you should recognize that you are really still in the quality control phase. Only when you have expunged every conceivable bug in the game should you release it to a wider audience for beta testing.

At this point in the game's life cycle, the game is complete and 100 percent functional, and you are only looking for a larger group of users to identify bugs that might have slipped past quality control. Before you release a game to beta testers, make absolutely certain that all of the graphics, sound effects, and music are completely ready to go, as if the game is ready to be sent out to stores. If you do not feel confident that the game is ready to sit on a retail shelf, then that is a sure sign that it is not yet out of the quality control phase. When you identify bugs during the beta test phase, you should collect them at regular time intervals and send out new releases—whether your schedule is daily or weekly.

When users stop thinking of the game as a beta version and they actually start to play it to have fun (with general trust in the game's stability), and when no new bugs have been identified for a length of time (such as a couple weeks or a month), then you can consider the game complete.

Post-Production

Post-production work on a game includes creating the install program that installs the game onto a computer system and writing the game manual. If you will be distributing the game via the Internet, you will definitely want to create a Web site for your game, with a bunch of screenshots and a list of the key features of the game.

Official Release

Once you have a complete package ready to go, burn the complete game installer with everything you need to play the game to a CD and give it to a few people who were not involved in the beta testing process. If you feel that the game is ready for prime time, you might send out copies of it to online- and printed-magazine editors for review.

Out the Door or Out the Window?

One thing is for certain: When you work on a game project for an employer who knows nothing about software development, you can count on having marketing run the show, which is not always good. Some of the best studios in the world are run by a small group of individuals who actually work on games but know very little about how to run a business or advertise a game to the general public. Far too often, those award-winning game designers and developers will turn over the reins of their small company to a fulltime manager (or president) because the pressure of running the business becomes too much for developers (who would rather write code than balance the accounts).

Managing the Game

The manager of a game studio might have learned the strategies to make a retail or wholesale company succeed. These strategies include concepts such as just-in-time inventory, employee management, cost control, and customer relationship management—all very good things to know when running a grocery store or sales department. The problem is, many managers fail to realize that software development is not a business, and programmers should not be treated like factory workers; rather, they should be treated like members of a research and development team.

Consider the infamous Bell Laboratories (or Bell Labs), an R&D center that has come up with hundreds of patents and innovations that have directly affected the computer industry (not the least of which was the transistor). A couple of intelligent guys might have invented the microprocessor, but the transistor was a revolutionary step that made the microprocessor possible. Now imagine if someone had treated Bell Labs like a factory, demanding results on a regular basis. Is that how human creativity works, through schedules and deadlines?

The case might be made that true genius is both creative and timely. Along that same train of thought, it might be said that genius is nothing but an extraordinary amount of hard work with a dash of inspiration here and there.

There are some really terrific game publishers that give development teams the leeway to add every last bell and whistle to a game, and those publishers should be applauded!

But—you knew that was coming, didn't you?—far too often, publishers simply want results without regard for the quality of a game. When shareholders become more important than developers in a game company, it's time to find a new job.

A Note about Quality

What is the best way to work with game developers or the best way to work with management? The goal, after all, is to produce a successful game. Learn the meaning behind the buzzwords. If you are a developer, try to explain the technology behind your game throughout the development life cycle and provide options to managers. By offering several technical solutions to any given problem, and then allowing the decision makers to decide which path to follow, you will succeed in completing the game on time and within budget.

The accusations and jibes actually go both ways! Management is often faced with developers who are competing with other developers in the industry. The goal might be a sound one; high-end game engines are often so difficult to develop that many companies would rather license an existing engine than build their own. Quite often a game is nothing more than a technology demo for the engine, because licensing might provide even more income than actual game sales (especially if royalties are involved). When a game is nearing completion and a competitor's game comes out with some fancy new feature, such as a software renderer with full anisotropic filtering (okay, that is impossible, but you get the point), the tendency is to cram a similar new feature into the game at the last minute for bragging rights. However, the new feature will have absolutely no bearing on the playability or fun factor of the game, and it might even reduce game stability.

This tendency is something that managers must deal with on a daily basis in a struggle to keep developers from modifying the game's design (resulting in a game that is never finished). Rather than constantly modify the design, developers should be promised work on a sequel or a new game so they can use all the new things they learned while working on the current game.

Empowering the Engine

Consider the game *Unreal*, by Epic Games. (As an aside, Epic Games was once called Epic Megagames, and they produced some very cool shareware games.) The *Unreal* engine was touted as a *Quake II* killer, with unbelievable graphics all rendered in software. Of course, 3D acceleration made *Unreal* even more impressive. But the problem with *Unreal* was not the technology behind the mesmerizing graphics in the game, but rather the gameplay. Gamers were playing tournament-style games, a trend that was somewhat missed by the developers, publishers, and gaming media at the time. In contrast, *Quake II* had a large and engaging single-player game in addition to multiplayer support that spawned a cult following and put the game at the top of the charts.

Unreal was developed from the start as a multiplayer game, since the game was in development for several years. Epic Games released *Unreal Tournament* about two years later, and it was simply awesome—a perfect example of putting additional efforts into a second game, rather than delaying the first. The only single-player component of *Unreal Tournament* is a game mode in which you can play against computer-controlled bots; it is undeniably a multiplayer game throughout.

Quality versus Trends

Blizzard was once a company that set the industry standard for creating extremely high-quality games, such as *Warcraft II*, *Starcraft*, and *Diablo*. These games alone have outsold the entire lineup from some publishers, with multiple millions of copies sold worldwide. Why was Blizzard so successful with these early games? In a word: quality. From the installer to the end of the game, Blizzard exuded quality in every respect. Then something happened. The company announced a new game, and then cancelled it. A new installment of *Warcraft* was announced (*Warcraft Adventures: Lord of the Clans*, a cartoon-style game that had the potential to supercede the coming "cell shading" trend pioneered by *Jet Set Radio* for the Dreamcast—not to mention that Blizzard missed out on the resurgence of the adventure game genre), and then forgotten for several years. *Diablo II* came out in 2001, and many scratched their heads, wondering why it took three years to develop a sequel that looked so much like the original.

Consider Future Trends

The problem is often not related to the quality of a game as much as it is related to trends. When it takes several years to develop an extremely complicated game, design decisions must be made in advance, and the designers have to do a little guesswork to try to determine where gaming trends are headed, and then take advantage of those trends in a game. A blockbuster game does not necessarily need to follow every new trend; on the contrary, the trends are set *by* the blockbuster games. An otherwise fantastic game that was revolutionary and ambitious at one point might find itself outdated by the time it is released.

Take Out the Guesswork

Age of Empires was released for the holiday season in 1997, at the dawn of the real-time strategy revolution in the gaming industry. This game was in development for perhaps two years before its release. That means work started on *Age of Empires* as early as 1995! Now, imagine the trends of the time and the average hardware on a PC, and it is obvious that the designer of the game had a good grasp of future trends in gaming.

Those RTS games that were developed with complete 3D environments still haven't seemed to catch on. In many ways, *Dark Reign II* is far superior to *Age of Empires II*, with gorgeous graphics and stunning 3D particle effects. Yet *Age of Empires II* has become more

of a LAN party favorite, along with *Quake III Arena*, *Unreal Tournament*, and *Counter-Strike*. Perhaps RTS fans are not interested in complete 3D environments. My personal suspicion is that the 3D element is distracting to a gamer who would prefer to focus on his strategy rather than navigating the 3D terrain.

Innovation versus Inspiration

As an aspiring game designer, what is the solution to the technology/trend problem? My advice is to play every game you can get your hands on (if you are not already an avid gamer). Play games that don't interest you to get a feel for a variety of games. Download and play every demo that comes out, regardless of the type of game. Demos are a great way for marketing departments to promote a game before it is finished, but they are also a great way for competitors to see what you have planned. As with most things in business or leisure, there is a tradeoff. It is great to have some fun while you play games, but try to determine how the game works and what is under the hood. If the game is based on a licensed engine rather than custom code, you might try to identify which engine powers the game.

Half-Life is probably one of the oldest games in the industry that is still being improved upon and packaged for sale on retail shelves. One of the most significant reasons for the success of *Half-Life* (along with the compelling story and gameplay) is the *Half-Life* SDK. This software development kit for the *Half-Life* engine is available for free download. While hundreds of third-party modifications (MODs) have been created for *Half-Life*, by far the most popular is *Counter-Strike* (which was finally packaged for retail sale after more than a year in beta, and then ported to Xbox).

The Infamous Game Patch

Regardless of the good intentions of developers, many games are rushed and sent out to stores before they are 100-percent complete. This is a result of a game that went over budget, a publisher that decided to drop the game but was convinced to complete it, or a publisher that is interested only in a first run of sales, without regard to quality.

A common trap that publishers have fallen into is the belief that they can rush a game, and then release a downloadable patch for it. The reasoning is that customers are already used to downloading new versions and updates to software, so there is nothing wrong with getting a game out the door a week before Christmas to make it for the holiday season. The flaw behind this reasoning is that games are largely advertised by word of mouth, not by marketing schemes. Due to the huge number of newsgroups and discussion lists (such as Yahoo! Groups) that allow millions of members to share information, ideas, and stories, it is impossible for a killer new game to be released without a few hundred thousand gamers knowing about it.

But now you see the trap. The same gamers who swap war stories online about their favorite games will rip apart a shoddy game that was released prematurely. This is a sign of sure death for a game. Only rarely will a downloadable patch be acceptable for a game that is released before it is complete.

Expanding the Game

Most successful games are followed by an expansion pack of some sort, whether it is a map pack or a complete conversion to a new theme. One of my favorite games of all time is *Homeworld*, which was created by Relic and published by Sierra. *Homeworld* is an extraordinary game of epic proportions, and it is possibly the most engaging and realistic game I have ever played. (The same applies to *Homeworld 2*, the excellent sequel.)

When the expansion game *Homeworld: Cataclysm* was released, I found that not only was there a new theme to the game (in fact, it takes place a number of years after the events in the original game), but the developers had actually added some significant new features to the game engine. The new technologies and ships in *Cataclysm* were enough to warrant buying the game, but *Cataclysm* is also a standalone expansion game that does not require the original to run.

Expansion packs and enhanced sequels allow developers to complete a game on schedule while still exercising their creative and technical skill on an additional product based on the same game. This is a great idea from a marketing perspective because the original game has already been completed, so the amount of work required to create an expansion game is significantly less and allows for some fine-tuning of the game.

Future-Proof Design

Developing a game with code reuse is one thing, but what about designing a game to make it future-proof? That is quite a challenge given that computer technology improves at such a rapid pace. The ironic thing about computer games is that developers usually target high-end systems when building the game, even though they can't fully estimate where mainstream computer hardware will be a year in the future. Yet, when a new high-end game is released, many gamers will go out and purchase upgrades for their computers to play the new game. You can see the circular cause-and-effect that results.

Overall, designing a game for the highest end of the hardware spectrum is not a wise decision because there are thousands of gamers in the world who do not have access to the latest hardware innovations—such as striped hard drives attached to RAID (*Redundant Array of Independent Disks*) controllers or a 64-MB DDR (*Double-Data Rate memory*) GeForce 3 video card. While hardware improvements are increasing as rapidly as prices seem to be dropping, the average gaming rig is still light-years beyond the average consumer PC, and that should be taken into account when you are targeting system hardware.

Game Libraries

A solid understanding of game development usually precedes work on a game library for a particular platform, and this usually takes place during the initial design and prototype phases of game development. It is becoming more common for publishers to contract with developers for multiple platforms. Whether the developers build an entirely new game library for each platform or develop a multi-platform game library is usually irrelevant to the publisher, who is only interested in a finished product. You can see now why Allegro is such a powerful ally and why I selected it for this book!

A development studio is likely to reap incredible rewards by developing a multi-platform game library that can be easily recompiled for any of the supported computer platforms. It is not unheard of to develop a library that supports PC, PlayStation 2, GameCube, and Xbox, all with the same code base. In the case of this book, you are able to write games directly for Windows or Linux without much effort, and for Mac and a few other systems with a little work. Allegro takes care of the details within the library.

Game Engines and SDKs

Game engines are far overrated in the media and online discussion groups as complete solutions to a developer's needs. Not true! Game engines are based on game libraries for one or more platforms, and the game engine is likely optimized to an incredible degree for a particular game. Common engines today include the *Half-Life* SDK, the *Unreal* engine, and the *Quake III* engine. These game engines can be used to create a completely new game, but that game is really just a total conversion for the existing engine. Some studios are up to the challenge of modifying the existing engine for their own needs, but far more often, developers will use the existing engine as is and simply customize it for their own game projects.

Examples of games based on an existing engine include *Star Trek Voyager: Elite Force II*, *Counter-Strike*, and even *Quake IV* (which is based on the *Doom III* engine). *Half-Life 2* is promising to be a strong contender in the *engine* business, pushing the envelope of realism to an even higher level than has been seen to date.

What Is Game Design?

Now that you have some background on the theory of game design and a good overview of the various game genres your game might fit into, I'll go over some real-world examples and cover information you might need when you want to take your game into the retail market.

So what exactly is game design? It is the ancient art of creating and defining games. Well, that's at least the short definition. *Game design* is the entire process of creating a game

idea, from research, to the graphical interface, to the unit's capabilities. Having an idea for a game is easy; making a game from that idea is the hard part—and that is just the design part! When creating a game, some of the jobs of a designer are to:

- Define the game idea
- Define all the screens and how they relate to each other and to the menus
- Explain how and why the interaction with the game is done
- Create a story that makes sense
- Define the game goals
- Write dialogues and other specific game texts
- Analyze the balance of the game and modify it accordingly
- And much, much more…

The Dreaded Design Document

Now that you finally have decided what kind of game you are making and you have almost everything planned out, it's time to prepare a design document. For a better understanding of what a design document should be, think of the movie industry.

When a movie is shot, the story isn't in anyone's head; it is completely described in the movie script. Actually, the movie script is usually written long before shooting starts. The author writes the script and then needs to take it to a big Hollywood company to get the necessary means to produce the movie, but this is a long process. After a company picks the movie, each team (actors, camera people, director, and so on) will get the copies of the script to do their job. When the wardrobe is done, the actors know the lines and emotion, and the director is ready, they start shooting the movie.

When dealing with game design, the process is sort of the same, in that the designers do the design document, and then they pitch to the company they work for to see whether the company has any interest in the idea. (No, trying to sell game designs to companies isn't a very nice future.) When the company gives the go for a game—probably after revising the design and for sure messing it up—each team (artists, programmers, musicians, and so on) gets the design document and starts doing its job. When some progress is made by all the teams, the actual production starts (such as testing the code with the art and including the music).

One more thing before I proceed: Just because some feature or menu is written in the design document, it doesn't mean it has to be that way no matter what. This is also similar to the movies, in that the actors follow the script, but sometimes they improvise, which makes the movie even more captivating.

The Importance of Good Game Design

Many young and beginning game programmers defend the idea that the game is in their head, and thus they refuse to do any kind of formal design. This is a bad approach for several reasons. The first one is probably the most important if you are working with a team. If you are working with other people on the game and you have the idea in your head, there are two possibilities: Your team members are psychic or you spend 90 percent of the time you should be developing your game explaining why the heck the player can't use the item picked in the first level to defeat the second boss. The second option is in no way fun.

Another valid reason to keep a formal design document is to keep focus. When you have the idea in your head, you will be working on it and modifying it even when you are finishing the programming part. This is bad because it will eventually force you to change code and lose time. I'm not saying that when you write something down, it is written in stone. All the aspects of the design document can and should change during development. The difference is that when you have a formal design, it's easy to keep focus and progress, whereas if you keep it in your head, it will be hard to progress because you won't settle with something and you will always be thinking of other stuff.

The last reason why you shouldn't keep the designs in your head is because you are human. We tend to forget stuff. Suppose you have the design in your head and you are about 50 percent done programming the game, but for some reason you have to stop developing the game for three weeks (due to vacation, exams month, aliens invading, or whatever the reason). When you get back to developing the game, most of the stuff that was previously so clear will not be as obvious, thus causing you lose to time rethinking it.

The Two Types of Designs

Even if there isn't an official distinction between design types, separating the design process into two types makes it easier to understand which techniques are more advantageous to the games you are developing.

Mini Design

You can do the mini design in about a week or so. It features a complete but general description of the game. A mini design document should be enough that any team member can pick it up, read it, and get the same idea of the game as the designer—but be allowed to include a little bit of his own ideas for the game (such as the artist designing the main character or the programmer adding a couple of features, such as cloud movement or parallax scrolling). Mini designs are useful when you are creating a small game or one that is heavily based on another game or a very well-known genre. Some distinctive aspects of a mini design document are

- General overview of the game
- Game goals

- Interaction of player and game
- Basic menu layout and game options
- Story
- Overview of enemies
- Image theme

Complete Design

The complete design document looks like the script from *Titanic*. It features every possible aspect of the game, from the menu button color to the number of hit points the barbarian can have. It is usually designed by various people, with help from external people, such as lead programmers or lead artists.

The complete design document takes too much time to make to be ignored or misinterpreted. Anyone reading it should see exactly the same game, colors, and backgrounds as the designer(s). This kind of design is reserved for big companies that have much money to spare. Small teams or lone developers should stay away from this type of design because most of the time they don't have the resources to do it. Some of the aspects a complete design should have are

- General overview of the game
- Game goals
- Game story
- Characters' stories and attributes
- NPC (*Non-Player Character*) attributes
- Player/NPC/other rule charts
- All the rules defined
- Interaction of the player and the game
- Menu layout and style and all game options
- Music description
- Sound description
- Description of the levels and their themes and goals

A Sample Design Document Template

The following sections describe a sample design document you can use for your own designs, but remember—these are just guidelines that you don't have to follow exactly. If you don't think a section applies to your game or if you think it is missing something, don't think twice about changing it.

General Overview

This is usually a paragraph or two describing the game very generally. It should briefly describe the game genre and basic theme, as well as the objectives of the player. It is a summary of the game.

Target System and Requirements

This should include the target system—Windows, Macintosh, or any other system, such as consoles—and a list of requirements for the game.

Story

Come on, this isn't any mind breaker—it is the game story. This covers what happened in the past (before the game started), what is happening when the player starts the game, and possibly what will happen while the game progresses.

Theme: Graphics and Sound

This section describes the overall theme of the game, whether it is set in ancient times in a land of fantasy or two thousand years in the future on planet Neptune. It should also contain descriptions or at least hints of the scenery and sound to be used.

Menus

This section should contain a short description and the objectives of the main menus, such as Start Game or the Options menu.

Playing a Game

This is probably the trickiest section. It should describe what happens from the time the user starts the game to when he starts to play—what usually happens, and how it ends. This should be set up as if you were describing what you would see on the screen if you were playing the game yourself.

Characters and NPCs Description

This section should describe the characters and the NPCs as well as possible. This description should include their names, backgrounds, attributes, special attacks, and so on.

Artificial Intelligence Overview

There are two options for this section. You can give an all-around general description of the game AI (*Artificial Intelligence*) and let the programmers pick that and develop their own set of rules, or you can describe almost every possible reaction and action an NPC can have.

Conclusion

The conclusion is usually a short paragraph covering—obviously—a conclusion to the game. It might feature your motivation in creating the game or some explanation of why the game is the way it is. They basically say the same thing, so just pick the one you prefer.

A Sample Game Design: Space Invaders

This section presents a sample mini design document for a *Space Invaders* type of game. *Space Invaders* is a relatively old game that you are probably familiar with. After reading this design document, you should be able to develop it on your own using the Mirus framework you developed earlier. Figure 16.1 shows a sample sketch of the game screen.

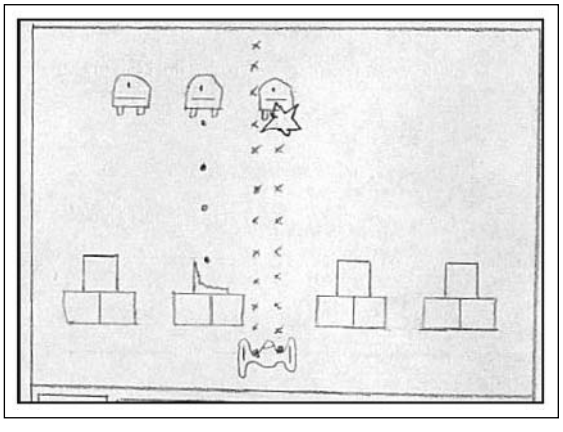


Figure 6.1 *Space Invaders* prototype

General Overview

Space Invaders is a typical arcade shooter game. The objective of the game is to destroy all the enemy ships in each level. The player controls a ship that can move horizontally at the bottom of the screen while it tries to avoid the bullets from the alien ships.

Target System and Requirements

Space Invaders is targeted for Windows 32-bit machines with DirectX 8.0 installed. Being such a low-end game, the basic requirements are minimal:

- Pentium 200 MHz
- 32 MB of memory
- 10 MB of free disk space
- SVGA DirectX-compatible video card

Story

Around 2049 A.D., aliens arrived on our planet, and they were not peaceful. They have destroyed two of the major cities in the world and are now threatening to destroy more.

The United Defense Force has decided to send their special agent, Gui Piskounov (don't ask), to destroy the alien force with the new experimental ship: ZS 3020 Airborne. You play the role of Piskounov. Your mission: To destroy all the alien scum.

Theme: Graphics and Sound

The whole game has a futuristic feeling to it. The main menus are heavily based on metallic walls and wire. The game itself is played in space, and as such, most of the backgrounds are stars or small planets. The ships have a very futuristic look to them. The game is full of heavy trance techno music with a very fast beat. Sounds are generally based on metal beating, explosions, and firing-bullet effects.

Menus

When the game starts, the user is presented with the main menu, in which he has five options.

Start New Game

This option starts a new game. The player is sent to the new game menu, where he can enter his name and chose the game difficulty.

Continue Previously Saved Game

This option starts a game that was previously saved. The player is sent to the load game menu, where he can choose a game from a list of previous saved games.

See Table of High Scores

This option shows the high scores table.

Options

This option shows the options menu. The player is sent to the options menu, where he can change the graphics, sound, and control settings.

Exit

This option exits the game.

Playing a Game

When the game starts, a company splash screen is shown for three seconds. After the three seconds, the screen fades to black and a splash screen starts to fade in. After four seconds, the screen fades to black again, and the player is sent to the main menu. When the player starts a new game, he is presented with a new menu screen, where he can enter his name and choose the game difficulty. After this is done, the user is sent to the game itself.

When each level starts, there is a three-second countdown for the game to start. The player can move his ship to the left or right and shoot using the controls defined in the options menu. When all the enemies are destroyed, the player advances a level. When the player is shot by an alien, he loses a life. If the player loses all the lives, the game ends. If the aliens reach the bottom of the screen, the game is also over.

If the player presses the Esc key while playing, the game is paused and a dialog box appears, asking what the user wants to do. He can choose from the following options:

- **Save game.** This option saves the game.
- **Options.** This option shows the options menu.
- **Quit game.** This option returns the player to the main menu.

Character and NPC Description

In this version of *Space Invaders*, there are two versions of alien ships. The first version consists of the normal ships that are constantly on the screen trying to destroy the player; the second version consists of ships that randomly appear and, if shot, give bonus points to the player.

Normal Ships

Normal ships are the typical enemies of the player. They can have various images, but their functionality is the same. They move left and right and randomly shoot bullets at the player vertically. When the ships reach a vertical margin, they move down a bit. These ships are destroyed with a single shot, and each ship destroyed gives 100 points to the player. As the levels progress, the ships move faster.

Bonus Ships

Bonus ships appear randomly at the top of the screen. They move horizontally and very quickly. These ships exist only to give bonus points to the player; they don't affect the gameplay because they don't shoot at the player and they don't have to be destroyed. When a bonus ship is destroyed, the game awards 500 points to the player.

Artificial Intelligence Overview

This game is very simple and requires almost no artificial intelligence. The ships move horizontally only until they reach one of the vertical margins, where they move down. They also randomly shoot bullets in a vertical-only direction.

Conclusion

The decision to keep this game simple but addictive was made to appeal to younger players, but also to almost any age genre, especially hardcore arcade gamers.

Game Design Mini-FAQ

Q: Why should I care about designing if I want to be a programmer?

A: Tough question. The first reason is because you will probably start developing your small games before you move to a big company and have to follow 200-page design documents in which you don't have any say. Next, being able to at least understand the concept of designing games will make your life a lot easier. If and when you are called for a meeting with the lead designer, you will at least understand what is happening.

Q: What is the best way to get a position as a fulltime game designer in some big game company?

A: First, chances of doing that are very slim, really. But the best way to try would be to start low and eventually climb the ladder. Start by working on the beta testing team, then maybe try to move to quality assurance or programming, and eventually try to give a game design to your boss. Please be aware that there are many steps from beta testing to even being a guest designer for a section of a game; time, patience, and perseverance are very important.

Summary

This chapter covered the subject of game design and discussed the phases of the game development life cycle. You learned how to classify your games by genre, how to manage development and testing, how to release and market your game, how to improve quality while meeting deadlines, and how to recognize some of the pitfalls of releasing an incomplete product. You then learned how to follow trends, how to expand and enhance a game with expansion packs, and how game libraries and game engines work together.

This was a rather short chapter for such an important topic, but this is a book mostly about programming, not design. If you have been paying attention, by now you should have a vague idea why designs are important and you should be able to pick up some of the topics covered here and design your own games. If you are having trouble, just use the fill-in template design document provided in this chapter and start designing.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the best way to get started creating a new game?
   A. Write the source code for a prototype.
   B. Create a game design document.
   C. Hire the cast and crew.
   D. Play other games to engender some inspiration.

2. What types of games are full of creativity and interesting technology that PC gamers often fail to notice?
   A. Console games
   B. Arcade games
   C. PC games
   D. Board games
3. What phrase best describes the additional features and extras in a game?
   A. Bonus levels
   B. Easter eggs
   C. Bells and whistles
   D. Updates and patches
4. What is usually the most complicated core component of a game, also called the graphics renderer?
   A. The DirectX library
   B. The Allegro library
   C. The double-buffer
   D. The game engine
5. What is the name of an initial demonstration of a game that presents the basic gameplay elements before the actual game has been completed?
   A. Beta
   B. Prototype
   C. Demo
   D. Release
6. What is the name of the document that contains the blueprints for a game?
   A. Game document
   B. Blueprint document
   C. Design document
   D. Construction document
7. What are the two types of game designs presented in this chapter?
   A. Mini and complete
   B. Partial and full
   C. Prototype and final
   D. Typical and sarcastic

8. What does NPC stand for?
   A. Non-Pertinent Character
   B. Non-Practical Condition
   C. Non-Perfect Caricature
   D. Non-Player Character
9. What are the chances of a newcomer finding a job as a fulltime game programmer or designer?
   A. Guaranteed
   B. Pretty good
   C. Questionable
   D. Negligible
10. What is the most important aspect of game development?
   A. Design
   B. Artwork
   C. Programming
   D. Implementation

CHAPTER 7

Basic Bitmap Handling and Blitting

The time has come to move into the core of the subject of this book on 2D game programming. Bitmaps are that core, and they are also at the very core of the Allegro game library. This chapter is not only an overview of bitmaps, but also of the core subject of blitting—two subjects that are closely related. In fact, because blitting is the process of displaying a bitmap, it might just be considered the workhorse for working with bitmaps. By the end of this chapter, you will have a solid understanding of how to create, load, draw, erase, and delete bitmaps, and you will use this new information to enhance the *Tank War* game that you started back in Chapter 4 by converting it to a bitmap-based game.

Here is a breakdown of the major topics in this chapter:

- Creating and deleting bitmaps
- Drawing and clipping bitmaps
- Reading a bitmap from disk
- Saving a screenshot to disk
- Enhancing *Tank War*

Introduction

The infamous *sprite* is at the very core of 2D game programming, representing an object that moves around on the screen. That object might be a solid object or it might be transparent, meaning that you can see through some parts of the object, revealing the background. These are called *transparent sprites*. Another special effect called *translucency*, also known as *alpha blending*, causes an object on the screen to blend into the background by a variable degree, from opaque to totally transparent (and various values in between).

Sprite programming is one of the most enjoyable aspects of 2D game programming, and it is essential if you want to master the subject.

Before you can actually draw a sprite on the screen, you must find a way to create that sprite. Sprites can be created in memory at run time, although that is not usually a good way to do it. The usual method is to draw a small graphic figure using a graphic editing tool, such as Paint Shop Pro, and then save the image in a graphic file (such as .bmp, .lbm, .pcx, or .tga). Your program can then load that image and use it as a sprite. Of course, you can create precisely the sort of image your sprite needs and load one file per sprite, but that is a time-consuming task that can get really confusing and difficult when the sprites start to add up! Imagine instead that you have many sprites stored in a single bitmap file, gathered in an arrangement so you can "grab" a sprite out of the image when you need it. This way you have to load only one bitmap image into memory, and that image serves as the "home" for all of your sprites. This is a much faster method, and it works better, too!

But how do you grab the sprites out of the bitmap image? That will be the focus of the next chapter. For now, I'll focus on how to create a bitmap in memory and then draw it to the screen. You can actually use this method to create an entire game (maybe one like *Tank War* from Chapter 4?) by drawing graphics right onto a small bitmap when the program starts, and then displaying that bitmap as often as needed. It makes sense that this would be a lot faster than doing all the drawing at every step along the way. This is how *Tank War* handled the graphics—by drawing every time the tanks needed to be displayed. As you might imagine, it is much faster to render the tanks beforehand and then quickly display that bitmap on the screen. Take a look at this code:

```
BITMAP *tank = create_bitmap(32, 32);
clear_bitmap(tank);
putpixel(tank, 16, 16, 15);
blit(tank, screen, 0, 0, 0, 0, 32, 32);
```

note

Render is a graphical term that can apply to any act of drawing.

There are some new functions here that you haven't seen before, so I'll explain what they do. The first function, called `create_bitmap`, does exactly what it appears to do—it creates a new bitmap of the specified size. The `clear_bitmap` function zeroes out the new bitmap, which is necessary because memory allocation does not imply a clean slate, just the space—sort of like buying a piece of property that contains trees, bushes, and shrubbery that must be cleared before you build a house. Now take notice of the third line, with a call to `putpixel`. Look at the first parameter, `tank`. If you'll recall the `putpixel` function from Chapter 3, you might remember that the first parameter was always `screen`, which caused drawing to go directly to the screen. In this instance, you want the pixel to be drawn on the new bitmap!

The `blit` function is something entirely new and a little bit strange, won't you agree? If you have heard of sprites, you have probably also heard of blitting—but just in case you haven't, I'll go over it. *Blit* is shorthand for the process called "bit-block transfer." This is a fancy way of describing the process of quickly copying memory (a bit block) from one location to another. I have never quite agreed with the phrase because it's not possible to copy individual bits haphazardly; only entire bytes can be copied. To access bits, you can peer into a byte, but there's no way to copy individual bits using the `blit` function. Semantics aside, this is a powerful function that you will use often when you are writing games with Allegro.

Isn't it surprising that you're able to draw a pixel onto the tank bitmap rather than to the screen? Allegro takes care of all the complicated details and provides a nice clean interface to the graphics system. On the Windows platform, this means that Allegro is doing all the DirectX record-keeping behind the scenes, and other platforms are similar with their respective graphics libraries. Now it starts to make sense why all of those graphics functions you learned back in Chapter 3 required the use of `screen` as the first parameter. I don't know about you, but I think it's kind of amazing how just a few short lines of code (such as those shown previously) can have such a huge impact. To validate the point, you'll open the *Tank War* game project at the end of this chapter and tweak it a little, giving it a technological upgrade using bitmaps. In the context of role playing, the game will go up a level.

There is so much information to cover regarding bitmaps and blitting that I'll get into the specifics of sprites and animation in the next chapter.

Dealing with Bitmaps

Now what I'd like to do is introduce you to all of the bitmap-related functions in Allegro so you'll have a complete toolbox before you get into sprites—because sprites depend entirely on bitmaps to work. There are many aspects of Allegro that I don't get into in this book because the library has support for functionality (such as audio recording) that is not directly applicable to a game—unless you want to add voice recognition, perhaps?

You are already familiar with the screen bitmap. Essentially, this is a very advanced and complicated mapping of the video display into a linear buffer—in other words, it's easy to draw pixels on the screen without worrying about the video mode or the type of computer on which it's running. The screen buffer is also called the *frame buffer*, which is a term borrowed from reel-to-reel projectors in theaters. In computing, you don't already have a reel of film waiting to be displayed; instead, you have conditional logic that actually constructs each frame of the reel as it is being displayed. The computer is fast enough to usually do this at a very high frame rate. Although films are only displayed at 24 frames per second (fps) and television is displayed at 30 fps, it is generally agreed that 60 fps is the minimum for computer games. Why do you suppose movies and TV run at such low

frame rates? Actually, the human eye is only capable of discerning about 30 fps. But it's a little different on the computer screen, where refresh rates and contrast ratios play a part, since quality is not always a constant thing as it is on a theater screen. Although a video card is capable of displaying more than 60 fps, if the monitor is only set to 60 Hertz (Hz), then a discernable flicker will be apparent, which is annoying at best and painful at worst. Very low vertical refresh rates can easily give you a headache after only a few minutes.

Although we deal with the screen in two dimensions (X and Y), it is actually just a single-dimensional array. You can figure out how big that array is by using the screen width and height.

```
Array_Size = Screen_Width * Screen_Height
```

A resolution of 800×600 therefore results in:

```
Array_Size = 800 * 600
Array_Size = 480,000
```

That's a pretty large number of pixels, wouldn't you agree? Just imagine that a game running 60 fps is blasting 480,000 pixels onto the screen 60 times per second! That comes to (480,000 * 60 =) 28,800,000 pixels per second. I'm not even talking about bytes here, just pixels. Most video modes use 3 bytes per pixel (bpp) in 24-bit color mode, or 2 bpp in 16-bit color mode. Therefore, what I'm really talking about is on the order of 90 million bytes per second in a typical game. And when was the last time you played a game at the lowly resolution of 800×600? I usually set my games to run at 1280×960. If you were to use 1600×1200, your poor video card would be tasked with pushing 180 million bytes per second. Now you can start to see what all the fuss is about regarding high-speed memory, with all the acronyms such as RDRAM, SDRAM, DDR, and so on. Your PC doesn't need 180 MB of video memory in this case—just very, very fast memory to keep the display going at 60 fps. The latest video cards with 256-MB DDR really use most of that awesome video memory for storing textures used in 3D games. The actual video buffer only requires 32 MB of memory at most.

That's quite a lot of new information (or maybe it's not so new if you are a videophile), and I've only talked about the screen itself. For reference, here is how the screen buffer is declared:

```
extern BITMAP *screen;
```

The real subject here is how to work with bitmaps, so take a look inside that bitmap structure:

```
typedef struct BITMAP           // a bitmap structure
{
   int w, h;                    // width and height in pixels
   int clip;                    // flag if clipping is turned on
   int cl, cr, ct, cb;          // clip left, right, top and bottom values
```

```
   GFX_VTABLE *vtable;          // drawing functions
   void *write_bank;            // C func on some machines, asm on i386
   void *read_bank;             // C func on some machines, asm on i386
   void *dat;                   // the memory we allocated for the bitmap
   unsigned long id             // for identifying sub-bitmaps
   void *extra;                 // points to a structure with more info
   int x_ofs;                   // horizontal offset (for sub-bitmaps)
   int y_ofs;                   // vertical offset (for sub-bitmaps)
   int seg;                     // bitmap segment
   ZERO_SIZE_ARRAY(unsigned char *, line);
} BITMAP;
```

The information in the `BITMAP` structure is not really useful to you as a programmer because it is almost entirely used by Allegro internally. Some of the values are useful, such as `w` and `h` (width and height) and perhaps the clipping variables.

Creating Bitmaps

The first thing you should know when learning about bitmaps is that they are not stored in video memory; they are stored in main system memory. Video memory is primarily reserved for the screen buffer, but it can also store textures. However, video memory is not available for storing run-of-the-mill bitmaps. Allegro supports a special type of bitmap called a *video bitmap*, but it is reserved for page flipping and double-buffering—something I'll get into in the next chapter.

As you have already seen, you use the `create_bitmap` function to create a memory bitmap.

```
BITMAP *create_bitmap(int width, int height);
```

By default, this function creates a bitmap using the current color depth. If you want your game to run at a specific color depth because all of your artwork is at that color depth, it's a good idea to call `set_color_depth` after `set_gfx_mode` when your program starts. The bitmap created with `create_bitmap` has clipping enabled by default, so if you draw outside the boundary of the bitmap, no memory will be corrupted. There is actually a related version of this function you can use if you want to use a specific color depth.

```
BITMAP *create_bitmap_ex(int color_depth, int width, int height);
```

If you do use `create_bitmap_ex` in lieu of `create_bitmap` with the assumed default color depth, you can always retrieve the color depth of a bitmap using this function:

```
int bitmap_color_depth(BITMAP *bmp);
```

After you create a new bitmap, if you plan to draw on it and blit it to the screen or to another bitmap, you must clear it first. The reason is because a new bitmap has random pixels on it based on the contents of memory at the space where the bitmap is now located.

To clear out a bitmap quickly, call this function:

```
void clear_bitmap(BITMAP *bitmap);
```

There is also an alternative version called `clear_to_color` that fills the bitmap with a specified color (while `clear_bitmap` fills in with 0, which equates to black).

```
void clear_to_color(BITMAP *bitmap, int color);
```

Possibly my absolute favorite function in Allegro is `create_sub_bitmap` because there is so much opportunity for mischief with this awesome function! Take a look:

```
BITMAP *create_sub_bitmap(BITMAP *parent, int x, y, width, height);
```

This function creates a sub-bitmap of an existing bitmap that actually shares the memory of the parent bitmap. Any changes you make to the sub-bitmap will be instantly visible on the parent and vice versa (if the sub-bitmap is within the portion of the parent that was drawn to). The sub-bitmap is clipped, so drawing beyond the edges will not cause changes to take place on the parent beyond that border. Now, about that little mention of mischief? You can create a sub-bitmap of the *screen*!

I'll wait a minute for that to sink in.

Do you have an evil grin yet? That's right, you can use sub-bitmaps to update or display portions of the screen, which you can use to create a windowing effect. This is absolutely awesome for building a scrolling background—something I'll spend a lot of time talking about in future chapters. Another point is, you can create a sub-bitmap of a sub-bitmap of a bitmap, but I wouldn't recommend creating a feedback loop by creating a bitmap of a sub-bitmap of a bitmap because that could cause your video card or monitor to explode. (Well, maybe not, but you get the picture.)

Okay, not really, but to be honest, that's the first thing I worry about when the idea of a feedback loop comes to mind. Feedback is generally good when you're talking about movies, books, video games, and so on, but feedback is very, very, very bad in electronics, as well as in software. Have you ever hooked up a video camera to a television and then pointed the camera at the screen? What you end up with is a view into eternity. Well, it *would* be infinite if the camera were centered perfectly, so the lens and TV screen are perfectly parallel, but you get the idea. If you try this, I recommend turning the volume down. Then again, leaving the volume on might help to drive the point home—feedback is dangerous, so naturally, let's try it.

```
BITMAP *hole = create_sub_bitmap(screen, 0, 0, 400, 300);
blit(hole, screen, 0, 0, 0, 0, 400, 300);
```

This snippet of code creates a sub-bitmap of the screen, and then blits that region onto itself. You can get some really weird effects by blitting only a portion of the sub-bitmap

and by moving the sub-bitmap while drawing onto the screen. The point is, this is just the sort of reason you're involved in computer science in the first place—to try new things, to test new hypotheses, and to boldly go where no…let's leave it at that.

Cleaning House

It's important to throw away your hamburger wrapper after you're finished eating, just as it is important to destroy your bitmaps after you're finished using them. To leave a bitmap in memory after you're finished is akin to tossing a wrapper on the ground. You might get away with it without complaint if no one else is around, but you might feel a tinge of guilt later (unless you're completely dissociated from your conscience and society in general). This is a great analogy, which is why I've used it to nail the point home. Leaving a bitmap in memory after your program has ended might not affect anything or anyone right now. After all, it's just one bitmap, and your PC has tons of memory, right? But eventually the trash is going to pile up, and pretty soon the roads, sidewalks, and parks in your once-happy little town will be filled with trash and you'll have to reboot the town…er, the computer. `destroy_bitmap` is your friend.

```
void destroy_bitmap(BITMAP *bitmap);
```

By the way, stop littering. You can't really reboot your town, but that would be convenient, wouldn't it? If Microsoft Windows was the mayor, we wouldn't have to worry about litter.

Bitmap Information

You probably won't need to use the bitmap information functions often, but they can be very useful in some cases. For starters, the most useful function is `bitmap_mask_color`, which returns the transparency color of a bitmap.

```
int bitmap_mask_color(BITMAP *bmp);
```

Allegro defines the transparency for you so there is really no confusion (or choice in the matter). For an 8-bit (256-color) bitmap, the mask/transparent color is 0, the first entry in the palette. All other color depths use pink as the transparent color (255, 0, 255). That's fine by me because I use these colors for transparency anyway, and I'm sure you would too if given the choice. I have occasionally used black (0, 0, 0) for transparency in the past, but I've found pink to be far easier to use. For one thing, the source images are much easier to edit with a pink background because dark-shaded pixels stand out clearly when they are superimposed over pink. Actually, Allegro assumes that transparency is always on. This surprised me at first because I always made use of a transparency flag with my own sprite engines in the past. But this assumption really does make sense when the transparent color is assumed to be the mask color, which implies hardware support. On the Windows platform, Allegro tells DirectDraw that pink (255, 0, 255) is the mask color, and DirectDraw handles the rest. What if you don't want transparency? Don't use pink! For

example, in later chapters I'll get into backgrounds and scrolling using tiles, and you certainly won't need transparency. Although you will use the same `blit` function to draw background tiles and foreground sprites, there is no speed penalty for doing so because drawing background tiles is handled at a lower level (within DirectX, SVGAlib, or whatever library Allegro uses on your platform of choice).

An American president brought the simple word "is" into the forefront of attention a few years back, and that's what you're going to do now—focus on several definitions using the word "is." The first is called `is_same_bitmap`.

```
int is_same_bitmap(BITMAP *bmp1, BITMAP *bmp2);
```

This function returns true if the two bitmaps share the same region of memory, with one being a sub-bitmap of another or both being sub-bitmaps of the same parent.

The `is_linear_bitmap` function returns true if the layout of video memory is natively linear, in which case you would have an opportunity to write optimized graphics code. This is not often the case, but it is available nonetheless.

```
int is_linear_bitmap(BITMAP *bmp);
```

A related function, `is_planar_bitmap`, returns true if the parameter is an extended-mode or mode-x bitmap. Given the cross-platform nature of Allegro, this *might* be true in some cases because the source code for your game *might* run if compiled for MS-DOS or console Linux.

```
int is_planar_bitmap(BITMAP *bmp);
```

The `is_memory_bitmap` function returns true if the parameter points to a bitmap that was created with `create_bitmap`, loaded from a data file or an image file. Memory bitmaps differ from screen and video bitmaps in that they can be manipulated as an array (such as `bitmap[y][x] = color`).

```
int is_memory_bitmap(BITMAP *bmp);
```

The related functions `is_screen_bitmap` and `is_video_bitmap` return true if their respective parameters point to screen or video bitmaps or sub-bitmaps of either.

```
int is_screen_bitmap(BITMAP *bmp);
int is_video_bitmap(BITMAP *bmp);
```

So if you create a sub-bitmap of the screen, such as:

```
BITMAP *scrn = screen;
```

then calling the function like this:

```
if (is_screen_bitmap(scrn))
```

will return true. Along that same line of thinking, `is_sub_bitmap` returns true if the parameter points to a sub-bitmap.

```
int is_sub_bitmap(BITMAP *bmp);
```

Acquiring and Releasing Bitmaps

Most modern operating systems use bitmaps as the basis for their entire GUI (*Graphical User Interface*), and Windows is at the forefront. There is an advanced technique for speeding up your program's drawing and blitting functions called "locking the bitmap." This means that a bitmap (including the screen buffer) can be locked so that only your code is able to modify it at a given moment. Allegro automatically locks and unlocks the screen whenever you draw onto it.

That is the bottleneck! Do you recall how many drawing functions were needed in *Tank War* to draw the tanks on the screen? Well, converting those drawing functions into bitmaps not only sped up the game thanks to blitting, but it also sped it up because each call to `rectfill` caused a lock and unlock of the screen, which was very, very time consuming (as far as clock cycles are concerned). But even a well-designed game with a scrolling background, transparent sprites, and so on will suffer if the screen or destination bitmap is not locked first. This process involves locking the bitmap, performing all drawing, and then unlocking it.

To lock a bitmap, you call the `acquire_bitmap` function.

```
void acquire_bitmap(BITMAP *bmp);
```

A shortcut function called `acquire_screen` is also available and simply calls `acquire_bitmap(screen)` for you.

```
void acquire_screen();
```

There is a danger to this situation, however, if you fail to release a bitmap after you have acquired (or locked) it. So always be sure to release any bitmaps that you have locked! More than likely you'll notice the mistake because your program will likely crash from repeated acquires and no releases (in which case the screen might never get updated). This situation is akin to falling into a black hole—the closer you get, the faster you fall! Note also that there is another function called `lock_bitmap` that is similar but only used by Allegro programs running under MS-DOS (which likely will never be the case—even the lowliest PC is capable of running at least Windows 95 or Linux, so I see no reason to support DOS).

After you update a locked bitmap, you want to release the bitmap with this function:

```
void release_bitmap(BITMAP *bmp);
```

and the related shortcut for the screen:

```
void release_screen();
```

Bitmap Clipping

Clipping is the process of ensuring that drawing to a bitmap or the screen does not occur beyond the boundary of that object. In most cases this is handled by the underlying architecture (DirectDraw, SVGAlib, and so on), but it is also possible to set a portion of the screen or a bitmap with clipping in order to limit drawing to a smaller region using the `set_clip` function.

```
void set_clip(BITMAP *bitmap, int x1, int y1, int x2, int y2);
```

The screen object in Allegro and all bitmaps that are created or loaded will automatically have clipping turned on by default and set to the boundary of the bitmap. However, you might want to change the default clip region using this function. If you want to turn clipping off, then you can pass zeros to the `x1`, `y1`, `x2`, and `y2` parameters, like this:

```
set_clip(bmp, 0, 0, 0, 0);
```

Why would you ever want to turn off clipping? It is a very real possibility. For one thing, if you are very careful how you update the screen in your own code, you might want to turn off automatic clipping of the screen to gain a slight improvement in the drawing speed. If you are very careful with your own created bitmaps, you can also turn off clipping of those objects if you are certain that clipping is not necessary. If you only read from a bitmap and you do not draw onto it, then clipping is irrelevant and not a performance factor at all. Clipping is only an issue with drawing to a bitmap. I highly recommend that you leave clipping alone at the default setting. More than likely, you will not need the slight increase in speed that comes from a lack of clipping, and you are more than likely to crash your program without it.

Loading Bitmaps from Disk

Not too long ago, video memory was scarce and a video palette was needed to allow low-end video cards to support more than a measly 256 colors. Even an 8-bit display is capable of supporting more colors, but they must be palettized, meaning that a custom selection of 256 colors may be active out of a palette of many thousands of available colors. I once had an 8-bit video card, and at one time I used to work with an 8-bit video mode. (If you must know, VGA mode 13h was extremely popular in the DOS days.) Today you can assume that anyone who will play your games will have at least a 16-bit display. Even that is up for discussion, and it can be argued that 24- and 32-bit color will always be available on any computer system likely to run your games.

I think 24-bit color (also called *true color*) is the best mode to settle on, as far as a standard for my own games, and I feel pretty confident about it. If anyone is still stuck with a 16-bit video card, then perhaps it's time for an upgrade. After all, even an old GeForce 2 or Radeon 7500 card can be had for about 30 dollars. Of course, as often happens, someone with a 15-year-old laptop will want to run your game and will complain that it doesn't support 16-bit color. In the world we live in today, it's not always safe to walk the streets, but it is safe to assume that 24-bit color is available. For one thing, 16-bit modes are slower than 24-bit modes, even if they are supported in the GPU. Video drivers get around the problem of packing 24 bits into 16 bits by prepacking them when a game first starts (in other words, when the bitmaps are first loaded), after which time all blitting (or 3D texture drawing) is as fast as any other color depth. If you want to target the widest possible audience for your game, 16-bit is a better choice. The decision is up to you because Allegro doesn't care which mode you choose; it will work no matter what.

You were given a glimpse at how to load a bitmap file way back in Chapter 3, but now I'm going to go over all the intricate details of Allegro's graphics file support. Allegro supports several formats, which is really convenient. If I were discussing only DirectX in this book, I would be limited to just .bmp files (or I could write the code to load other types of files). Windows .bmp files are fine in most cases, but some programmers prefer other formats—not for any real technical reason, but sometimes artwork is delivered in another format.

Allegro natively supports the graphics file formats in Table 7.1.

Table 7.1 Natively Supported Graphics File Formats

Graphics Format	Extension	Color Depths
Windows / OS/2 Bitmap	BMP	8, 24
Truevision Targa	TGA	8, 16, 24, 32
Z-Soft's PC Paintbrush	PCX	8, 24
Deluxe Paint / Amiga	LBM	8

Reading a Bitmap File

The easiest way to load a bitmap file from disk is to call the `load_bitmap` function.

```
BITMAP *load_bitmap(const char *filename, RGB *pal);
```

This function will load the specified file by looking at the file extension (.bmp, .tga, .pcx, or .lbm) and returning a pointer to the bitmap data loaded into memory. If there is an error, such as if the file is not found, then the function returns NULL. The first parameter is the filename, and the second parameter is a pointer to a palette that you have already

defined. In most cases this will simply be NULL because there is no need for a palette unless you are using an 8-bit video mode. Just for the sake of discussion, if you are using an 8-bit video mode and you load a true color image, passing a pointer to the palette parameter will cause an optimized palette to be generated when the image is loaded. If you want to use the current palette in an 8-bit display, simply pass NULL, and the current palette will be used.

As I mentioned, `load_bitmap` will read any of the four supported graphics formats based on the extension. If you want to specifically load only one particular format from a file, there are functions for doing so. First, you have `load_bmp`.

```
BITMAP *load_bmp(const char *filename, RGB *pal);
```

As was the case with `load_bitmap`, you can simply pass NULL to the second parameter unless you are in need of a palette. Note that in addition to these loading functions, Allegro also provides functions for saving to any of the supported formats. This means you can write your own graphics file converter using Allegro if you have any special need (such as doing batch conversions).

To load a Deluxe Paint/Amiga LBM file, you can call `load_lbm`:

```
BITMAP *load_lbm(const char *filename, RGB *pal);
```

which does pretty much the same thing as `load_bmp`, only with a different format. The really nice thing about these loaders is that they provide a common bitmap format in memory that can be used by any Allegro drawing or blitting function. Here are the other two loaders:

```
BITMAP *load_pcx(const char *filename, RGB *pal);
BITMAP *load_tga(const char *filename, RGB *pal);
```

Saving Images to Disk

What if you want to add a feature to your game so that when a certain button is pressed, a screenshot of the game is written to disk? This is a very useful feature you might want to add to any game you work on. Allegro provides the functionality to save to BMP, PCX, and TGA files, but not LBM files. Here's the `save_bitmap` function:

```
int save_bitmap(const char *filename, BITMAP *bmp, const RGB *pal);
```

This couldn't be any easier to use. You just pass the filename, source bitmap, and optional palette to `save_bitmap`, and it creates the image file. Here are the individual versions of the function:

```
int save_bmp(const char *filename, BITMAP *bmp, const RGB *pal);
int save_pcx(const char *filename, BITMAP *bmp, const RGB *pal);
int save_tga (const char *filename, BITMAP *bmp, const RGB *pal);
```

Saving a Screenshot to Disk

Now how about that screen-save feature? Here's a short example of how you might do that (assuming you have already initialized graphics mode and the game is running):

```
BITMAP *bmp;
bmp = create_sub_bitmap(screen, 0, 0, SCREEN_W, SCREEN_H);
save_bitmap("screenshot.pcx", bmp, NULL);
destroy_bitmap(bmp);
```

Whew, that's a lot of functions to remember! But don't worry, I don't expect you to memorize them. Just use this chapter as a flip-to reference whenever you need to use these functions. It's also helpful to see them and get a little experience with the various bitmap functions that you will be using frequently in later chapters.

Blitting Functions

Blitting is the process of copying one bit block to another location in memory, with the goal of doing this as quickly as possible. Most blitters are implemented in assembly language on each specific platform for optimum performance. The inherent low-level libraries (such as DirectDraw) will handle the details, with Allegro passing it on to the blitter in DirectDraw.

Standard Blitting

You have already seen the `blit` function several times, so here's the definition:

```
void blit(BITMAP *source, BITMAP *dest, int source_x, int source_y,
    int dest_x, int dest_y, int width, int height);
```

Table 7.2 provides a rundown of the parameters for the `blit` function.

Table 7.2 Parameters for the blit Function

Parameter	Description
`BITMAP *source`	The source bitmap (copy from)
`BITMAP *dest`	The destination bitmap (copy to)
`int source_x`	The x location on the source bitmap to copy from
`int source_y`	The y location on the source bitmap to copy from
`int dest_x`	The x location on the destination bitmap to copy to
`int dest_y`	The y location on the destination bitmap to copy to
`int width`	The width of the source rectangle to be copied
`int height`	The height of the source rectangle to be copied

Don't be intimidated by this function; `blit` is always this messy on any platform and with every game library I have ever used. But trust me, this is the bare minimum information you need to blit a bitmap (in fact, one of the simplest I have seen), and once you've used it a few times, it'll be old nature to you. The important thing to remember is how the source rectangle is copied into the destination bitmap. The rectangle's upper-left corner starts at (`source_x`, `source_y`) and extends right by `width` pixels and down by `height` pixels. In addition to raw blitting, you can use the `blit` function to convert images from one pixel format to another if the source and destination bitmaps have different color depths.

Scaled Blitting

There are several more blitters provided by Allegro, including the very useful `stretch_blit` function.

```
void stretch_blit(BITMAP *source, BITMAP *dest, int source_x, source_y,
     source_width, source_height, int dest_x, dest_y, dest_width, dest_height);
```

The `stretch_blit` function performs a scaling process to squeeze the source rectangle into the destination bitmap. Table 7.3 presents a rundown of the parameters.

Table 7.3 Parameters for the stretch_blit Function

Parameter	Description
`BITMAP *source`	The source bitmap
`BITMAP *dest`	The destination bitmap
`int source_x`	The x location on the source bitmap to copy from
`int source_y`	The y location on the source bitmap to copy from
`int source_width`	The width of the source rectangle
`int source_height`	The height of the source rectangle
`int dest_x`	The x location on the destination bitmap to copy to
`int dest_y`	The y location on the destination bitmap to copy to
`int dest_width`	The width of the destination rectangle (scaled into)
`int dest_height`	The height of the destination rectangle (scaled into)

The `stretch_blit` function is really useful and can be extremely handy at times for doing special effects, such as scaling the sprites in a game to simulate zooming in and out. However, take care when you use `stretch_blit` because it's not as hardy as `blit`. For one thing, the source and destination bitmaps must have the same color depth, and the source must be a memory bitmap. (In other words, the source can't be the screen.) You should also take care that you don't try to specify a rectangle outside the boundary of either the source or the destination. This means if you are copying the entire screen into a smaller

bitmap, be sure to specify (0,0) for the upper-left corner, `(SCREEN_W - 1)` for the width, and `(SCREEN_H - 1)` for the height. The screen width and height values are counts of pixels, not screen positions. If you specify a source rectangle of (0, 0, 1024, 768), it could crash the program. What you want instead is (0, 0, 1023, 767) and likewise for other resolutions. The same rule applies to memory bitmaps—stay within the boundary or it could cause the program to crash.

Masked Blitting

A masked blit involves copying only the solid pixels and ignoring the transparent pixels, which are defined by the color pink (255, 0, 255) on high color and true color displays or by the color at palette index 0 in 8-bit video modes (which I will not discuss anymore beyond this point). Here is the definition for the `masked_blit` function:

```
void masked_blit(BITMAP *source, BITMAP *dest, int source_x, int source_y,
     int dest_x, int dest_y, int width, int height);
```

This function has the exact same list of parameters as `blit`, so to learn one is to understand both, but `masked_blit` ignores transparent pixels while `blit` draws everything! This function is the basis for sprite-based games, as you will see later in this chapter. Although there are custom sprite-drawing functions provided by Allegro, they essentially call upon `masked_blit` to do the real work of drawing sprites. However, unlike `blit`, the source and destination bitmaps must have the same color depth.

Masked Scaled Blitting

One of the rather odd but potentially very useful alternative blitters in Allegro is `masked_stretch_blit`, which does both masking of transparent pixels and scaling.

```
void masked_stretch_blit(BITMAP *source, BITMAP *dest, int source_x,
     source_y, source_w, source_h, int dest_x, dest_y, dest_w, dest_h);
```

The parameters for this function are identical to those for `stretch_blit`, so I won't go over them again. Just know that this combines the functionality of masking and scaling. However, you should be aware that scaling often mangles the transparent pixels in an image, so this function can't guarantee perfect results, especially you are when dealing with non-aligned rectangles. In other words, for best results, make sure the destination rectangle is a multiple of the source so that scaling is more effective.

Enhancing Tank War—From Graphics Primitives to Bitmaps

Well, are you ready to start making enhancements to *Tank War,* as promised back in Chapter 4? The last two chapters have not been very forthcoming with this sort of information, so now that you have more knowledge, let's put it to good use.

Tank War was developed in Chapter 4 to demonstrate all of the vector graphics support in Allegro, and also to provide a short break from all the theory. If I had my way, each new subject would be followed by a short game to demonstrate how a new feature works, but that would take too much time (and paper). Instead of going the creative route and creating a fun new game in every chapter, I think it's helpful to enhance an existing game with the new technology you learn as you go along. It has a parallel in real life, and it demonstrates the life cycle of game development from early concept through the prototype stage and on to completion. One benefit to enhancing the game with new tricks and techniques you learn as you go along is that changes only affect a few lines of code here and there, while entirely new games take up pages of code. Besides, this is not a "101 games" type of book, like those that were popular years ago; instead, you are learning both high- and low-level game programming techniques that will work across different operating systems.

I have huge plans for *Tank War*, and you will snicker at these early versions later because you will be making all kinds of improvements to the game in the coming chapters—a scrolling background, animated sprites, joystick control, sound effects, and other great things. Who knows, maybe the game will eventually be playable over the Internet!

Now, returning to the new code you just learned (which I will explain completely in the next section), what you need to do is create a bitmap surface for both of the tanks so that blitting will work. Create two tank bitmaps.

```
BITMAP *tank_bmp1 = create_bitmap(32, 32);
BITMAP *tank_bmp2 = create_bitmap(32, 32);
```

That will do nicely in theory, but the tank variables will be put in tankwar.h so the declaration will have to be separated from the initialization. (You can't use `create_bitmap` on the same line—more on that in a moment.) There's also the problem that each tank requires four directions, so each one will actually need four bitmaps. Now you need to clear out the bitmap memory so it's a nice clean slate.

```
clear_bitmap(tank_bmp1);
clear_bitmap(tank_bmp2);
```

Great! Now what? Now all you have to do is modify *Tank War* so it draws the tanks using `blit` instead of calling `drawtank` every time. Here is that blitting code:

```
blit(tank1, screen, 0, 0, X, Y, 32, 32);
blit(tank2, screen, 0, 0, X, Y, 32, 32);
```

Of course, this pseudo-code doesn't take into account the need for a separate bitmap for each direction the tank can travel (north, south, east, and west). But in theory, this is how it will work. On the CD-ROM, there is a project in the chapter07 folder for *Tank War* with the completed changes. But I encourage you to load up the initial Chapter 4 version of *Tank War* and make these minor changes yourself so you can get the full effect of this lesson.

When you open the tankwar project, you'll see the two files that comprise the source code: main.c and tankwar.h. Open the tankwar.h header file and add the following line after the `gameover` variable line:

```
//declare some variables
int gameover = 0;
BITMAP *tank_bmp[2][4];
```

This will take care of four bitmaps for each tank, and it's all wrapped nicely into a single array so it will be easy to use. Based on how the game uses `tanks[0]` and `tanks[1]` structures to keep track of the tanks, it will be easier if the bitmaps are stored in this array. Now open the main.c source code file. The goal here is to make as few changes as possible, keeping to the core of the original game at this point and just making those changes necessary to convert the game from vector-based graphics to bitmap-based graphics.

You can't really create the bitmaps in the header file, so this line just created the bitmap variables; you'll actually create the bitmaps in main.c. Do you remember how the tanks were set up back in Chapter 4? It was actually done by a function called `setuptanks`. All that needs to be done here is to create the two bitmaps, so put that code inside `setuptanks`. Look in main.c for the function and modify it as shown. (The changes are in bold.)

```
void setuptanks()
{
    int n;

    //player 1
    tanks[0].x = 30;
    tanks[0].y = 40;
    tanks[0].speed = 0;
    tanks[0].color = 9;
    tanks[0].score = 0;
    for (n=0; n<4; n++)
    {
        tank_bmp[0][n] = create_bitmap(32, 32);
        clear_bitmap(tank_bmp[0][n]);
        tanks[0].dir = n;
        drawtank(0);
    }
    tanks[0].dir = 1;

    //player 2
    tanks[1].x = SCREEN_W-30;
    tanks[1].y = SCREEN_H-30;
    tanks[1].dir = 3;
```

```
    tanks[1].speed = 0;
    tanks[1].color = 12;
    tanks[1].score = 0;
    for (n=0; n<4; n++)
    {
        tank_bmp[1][n] = create_bitmap(32, 32);
        clear_bitmap(tank_bmp[1][n]);
        tanks[1].dir = n;
        drawtank(1);
    }
}
```

It has required a lot of jumping around in the code, but so far you've only added a few lines of code. Not bad for starters! But now you're going to make some major changes to the `drawtank` function. This is where all those `rectfill` function calls will be pointed to the new tank bitmaps instead of directly to the screen. The actual logic hasn't changed, just the destination bitmap. I realize there are better and easier ways to rewrite this game to use bitmaps, but again, the goal is not to rewrite half the game, it is to make the fewest changes to get the job done. Note the changes in bold and make these changes in the `drawtank` function so it looks like this:

```
void drawtank(int num)
{
    int x = 15; //tanks[num].x;
    int y = 15; //tanks[num].y;
    int dir = tanks[num].dir;

    //draw tank body and turret
    rectfill(tank_bmp[num][dir], x-11, y-11, x+11, y+11, tanks[num].color);
    rectfill(tank_bmp[num][dir], x-6, y-6, x+6, y+6, 7);

    //draw the treads based on orientation
    if (dir == 0 || dir == 2)
    {
        rectfill(tank_bmp[num][dir], x-16, y-16, x-11, y+16, 8);
        rectfill(tank_bmp[num][dir], x+11, y-16, x+16, y+16, 8);
    }
    else
    if (dir == 1 || dir == 3)
    {
        rectfill(tank_bmp[num][dir], x-16, y-16, x+16, y-11, 8);
        rectfill(tank_bmp[num][dir], x-16, y+16, x+16, y+11, 8);
    }
```

```
    //draw the turret based on direction
    switch (dir)
    {
        case 0:
            rectfill(tank_bmp[num][dir], x-1, y, x+1, y-16, 8);
            break;
        case 1:
            rectfill(tank_bmp[num][dir], x, y-1, x+16, y+1, 8);
            break;
        case 2:
            rectfill(tank_bmp[num][dir], x-1, y, x+1, y+16, 8);
            break;
        case 3:
            rectfill(tank_bmp[num][dir], x, y-1, x-16, y+1, 8);
            break;
    }
}
```

Now that wasn't difficult at all, was it? Just a single parameter on all the `rectfill` function calls to point the drawing onto the tank bitmaps instead of onto the screen, and a minor change to the x and y variables. The original *Tank War* would draw the tanks directly on the screen using the x and y values for each tank, so I just modified it here to base the x and y on the center of the tank bitmap instead. So let's summarize what has been done so far.

1. Define the tank bitmap variables.
2. Create the tank bitmaps in memory.
3. Draw the tank images onto the tank bitmaps.

What is left to do? Just one more thing! Instead of calling `drawtank` in the main game loop, this has to be changed to `blit`! Let's do it. Scroll down to the end of the main.c file, look for the two `drawtank` lines of code, and replace them with the `blit` functions as the following listing shows:

```
    //game loop
    while(!gameover)
    {
        //erase the tanks
        erasetank(0);
        erasetank(1);

        //check for collisions
        clearpath(0);
        clearpath(1);
```

```
        //move the tanks
        movetank(0);
        movetank(1);

        //draw the tanks
        blit(tank_bmp[0][tanks[0].dir], screen, 0, 0,
            tanks[0].x-16, tanks[0].y-16, 32, 32);
        blit(tank_bmp[1][tanks[1].dir], screen, 0, 0,
            tanks[1].x-16, tanks[1].y-16, 32, 32);

        //update the bullets
        updatebullet(0);
        updatebullet(1);

        //check for keypresses
        if (keypressed())
            getinput();

        //slow the game down (adjust as necessary)
        rest(30);
    }
```

The `blit` function really is only complicated by the multi-dimensional `tank_bmp` array, but this array results in far fewer lines of code than would otherwise be necessary using a `switch` or an `if` statement to draw the appropriate bitmap.

Summary

This chapter was an essential step in the path to writing great 2D games. Bitmaps are the core of 2D games and of Allegro, and in this chapter you learned to create, draw, erase, load, and delete bitmaps using a variety of Allegro functions. You also learned quite a bit about blitting, the process of drawing a bitmap to the screen really quickly.

Chapter Quiz

1. What does "blit" stand for?
   A. Blitzkrieg
   B. Bit-block transfer
   C. Bit-wise transparency
   D. Basic logarithmic infrared transmitter

2. What is a DHD?
   A. Dynamic hard drive
   B. Destructive hyperactivity disorder
   C. Dial home device
   D. That wasn't in the chapter!

3. How many pixels are there in an 800×600 screen?
   A. 480,000
   B. 28,800,000
   C. 65,538
   D. 47

4. What is the name of the object used to hold a bitmap in memory?
   A. `hold_bitmap`
   B. `create_bitmap`
   C. `OBJECT`
   D. `BITMAP`

5. Allegorically speaking, why is it important to destroy bitmaps after you're done using them?
   A. Because bitmaps are evil and must be destroyed.
   B. Because Microsoft Windows is the mayor.
   C. Because the trash will pile up over time.
   D. Because you can't reboot your hometown.

6. Which Allegro function has the potential to create a black hole if used improperly?
   A. `acquire_bitmap`
   B. `create_supernova`
   C. `do_feedback`
   D. `release_bitmap`

7. What types of graphics files are supported by Allegro?
   A. PCX, LBM, BMP, and GIF
   B. BMP, PCX, LBM, and TGA
   C. GIF, JPG, PNG, and BMP
   D. TGA, TIF, JPG, and BMP

8. What function is used to draw a scaled bitmap?
   A. `draw_scaled_bitmap`
   B. `stretch_blit`
   C. `scaled_blit`
   D. `masked_scaled_blit`

9. Why would you want to lock the screen while drawing on it?
   A. If it's not locked, Allegro will lock and unlock the screen for every draw.
   B. To prevent anyone else from drawing on your screen.
   C. To keep the screen from getting away while you're using it.
   D. To prevent a feedback loop that could destroy your monitor.

10. What is the name of the game you've been developing in this book?
    A. *Super Allegro Bros.*
    B. *Barbie's Motorhome Adventure*
    C. *Teenage Neutered Midget Poodles*
    D. *Tank War*

CHAPTER 8

Basic Sprite Programming: Drawing Scaled, Flipped, Rotated, Pivoted, and Translucent Sprites

It is amazing to me that in the year 2004, we are still talking about, writing about, and developing games with sprites. There are just some ideas that are so great that no amount of new technology truly replaces them entirely. A *sprite* is a small image that is moved around on the screen. Many programmers misuse the term to describe any small graphic image in a game. Static, unmoving objects on the screen are not sprites because by very definition a *sprite* is something that moves around and does something on the screen, usually in direct relation to something the player is doing within the game. The analogy is to a mythical sprite—a tiny, mischievous, flying creature that quickly flits about, looking something like a classical fairy, but smaller. Of course the definition of a sprite has grown to include any onscreen game object, regardless of size or speed.

While the previous chapter provided all the prerequisites for working with sprites, this chapter delves right into the subject at full speed. Technically, a sprite is no different than a bitmap as far as Allegro is concerned. In fact, the sprite-handling functions in this chapter define sprites using the `BITMAP *` pointer. You can also draw a sprite directly using any of the bitmap drawing functions. However, Allegro provided a number of custom sprite functions that help to make your life as a 2D game programmer a little easier, in addition to some special effects that will knock your socks off! What I'm talking about is the ability to add dynamic lighting effects to one or more sprites on the screen! That's right—in this chapter you will learn to not only load, create, and draw sprites, but also how to apply lighting effects to those sprites. Combine this with alpha blending and transparency, and you'll learn to do some really amazing things in this chapter.

This chapter uses the word "basic" in the title because, although this is a complete overview of Allegro's sprite support, the upcoming chapters will feature a lot of the more advanced coverage. At this point, I believe it's more important to provide you with some exposure to all of the sprite routines available so you can see how they'll be used as you go along. If you don't see the big picture yet, that's understandable, but it's very helpful to grasp the key topics in this chapter because they're vital to the rest of the book. To help solidify the new information in your mind, you'll dig into *Tank War* a little more at the end of the chapter and enhance it with some sprites!

Here is a breakdown of the major topics in this chapter:

- Drawing regular and scaled sprites
- Drawing flipped sprites
- Drawing rotated and pivoted sprites
- Enhancing *Tank War*

Basic Sprite Handling

Now that you've had a thorough introduction to bitmaps in the last chapter—how to create them, load them from disk, make copies, and blit them—you have the prerequisite information for working with sprites. A sprite image is simply a bitmap image. What you do with a sprite image (and the sprite functionality built into Allegro) differentiates sprites from mere bitmaps.

Drawing Regular Sprites

The first and most important function to learn is `draw_sprite`.

```
void draw_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

This function is similar to `masked_blit` in that it draws the sprite image using transparency. As you'll recall from the previous chapter, the transparent color in Allegro is defined as pink (255, 0, 255). Therefore, if your source bitmap uses pink to outline the image, then that image will be drawn transparently by `draw_sprite`. Did you notice that there are no `source_x`, `source_y`, `width`, or `height` parameters in this function call? That is one convenience provided by this function. It is assumed that you intend to draw the whole sprite, so those values are provided automatically by `draw_sprite` and you don't need to worry about them. This assumes that the entire bitmap is comprised of a single sprite. Of course, you can use this technique if you want, but a far better method is to store multiple sprites in a single bitmap and then draw the sprites by "grabbing" them out of the bitmap (something I'll cover later in this chapter).

The most important factor to consider up front when you are dealing with sprites is the color depth of your game. Until now, you have used the default color depth and simply called `set_gfx_mode` before drawing to the screen. Allegro does not automatically use a high-color or true-color color depth even if your desktop is running in those modes. By default, Allegro runs in 8-bit color mode (the mode that has been used automatically in all the sample programs thus far). Figure 8.1 shows a sprite drawn to the screen with the default color depth.

Figure 8.1 A high-color sprite drawn to the screen with a default 8-bit color depth. Sprite image courtesy of Ari Feldman.

Drawing that same sprite using a 16-bit high-color mode results in the screen shown in Figure 8.2. Notice how the sprite is now drawn with the correct transparency, whereas the pink transparent color was incorrectly drawn on the 8-bit display shown in Figure 8.1.

The program to produce these sprites is provided in the following listing and included on the CD-ROM under the name drawsprite.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

int main()
```

```
{
    BITMAP *dragon;
    int x, y;

    //initialize the program
    allegro_init();
    install_keyboard();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0);

    //print some status information
    textprintf(screen,font,0,0,WHITE,"Resolution = %ix%i",
        SCREEN_W, SCREEN_H);
    textprintf(screen, font, 0, 10, WHITE, "Color depth = %i",
        bitmap_color_depth(screen));

    //load the bitmap
    dragon = load_bitmap("spacedragon1.bmp", NULL);
    x = SCREEN_W/2 - dragon->w/2;
    y = SCREEN_H/2 - dragon->h/2;

      //main loop
    while (!key[KEY_ESC])
      {
        //erase the sprite
        rectfill(screen, x, y, x+dragon->w, y+dragon->h, 0);

        //move the sprite
        if (x-- < 2)
            x = SCREEN_W - dragon->w;

        //draw the sprite
        draw_sprite(screen, dragon, x, y);

        textprintf(screen,font,0,20,WHITE, "Location = %ix%i", x, y);
        rest(1);
    }

    //delete the bitmap
    destroy_bitmap(dragon);

    return 0;
}
END_OF_MAIN();
```

Figure 8.2 The high-color sprite is drawn to the screen with 16-bit color. Sprite image courtesy of Ari Feldman.

Transparency is an important subject when you are working with sprites, so it is helpful to gain an understanding of it right from the start. Figure 8.3 shows an example of a sprite drawn with and without transparency, as you saw in the sample drawsprite program when an 8-bit color depth was used.

When a sprite is drawn transparently, all but the transparent pixels are copied to the destination bitmap (or screen). This is necessary because the sprite has to be stored in a bitmap image of one type or another (.bmp, .pcx, and so on), and the computer can only deal with rectangular bitmaps in memory. In reality, the computer only deals with chunks of memory anyway, so it cannot draw images in any other shape but rectangular (see Figure 8.4).

In the next chapter, I'll show you a technique you can use to draw only the actual pixels of a sprite and completely ignore the transparent pixels during the drawing process. This is a special feature built into Allegro called *compiled sprites*. Compiled sprites, as well as run-length encoded (compressed) sprites, can be drawn much faster than regular sprites drawn with `draw_sprite`, so the next chapter will be very interesting indeed!

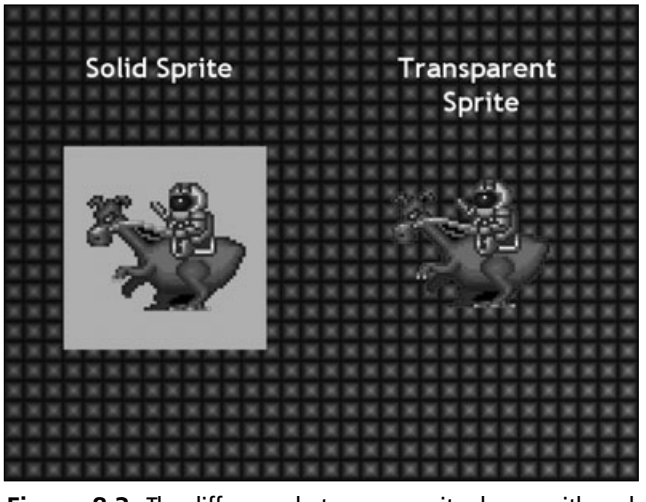


Figure 8.3 The difference between a sprite drawn with and without transparency. Sprite image courtesy of Ari Feldman.

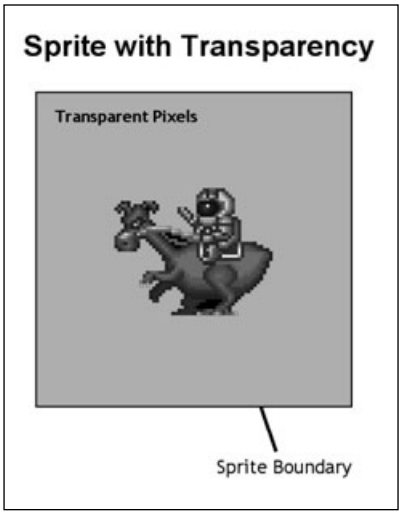


Figure 8.4 The actual sprite is contained inside a rectangular image with transparent pixels. Sprite image courtesy of Ari Feldman.

Drawing Scaled Sprites

Scaling is the process of zooming in or out of an image, or in another context, shrinking or enlarging an image. Allegro provides a function for drawing a sprite within a specified rectangle on the destination bitmap; it is similar to `stretched_blit`. The function is called `stretch_sprite` and it looks like this:

```
void stretch_sprite(BITMAP *bmp, BITMAP *sprite,
  int x, int y, int w, int h);
```

The first parameter is the destination, and the second is the sprite image. The next two parameters specify the location of the sprite on the destination bitmap, while the last two parameters specify the width and height of the resulting sprite. You can only truly appreciate this function by seeing it in action. Figure 8.5 shows the *ScaledSprite* program, which displays a sprite at various resolutions.

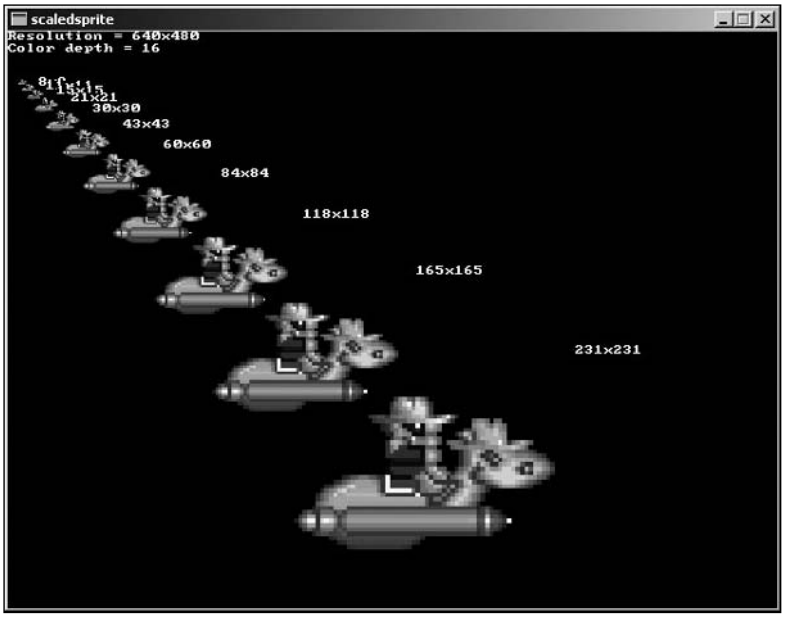


Figure 8.5 A high-resolution sprite image scales quite well. Sprite image courtesy of Ari Feldman.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

int main()
{
    BITMAP *cowboy;
      int x, y, n;
    float size = 8;

    //initialize the program
    allegro_init();
      install_keyboard();
      set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0);

    //print some status information
    textprintf(screen,font,0,0,WHITE,"Resolution = %ix%i",
        SCREEN_W, SCREEN_H);
    textprintf(screen, font, 0, 10, WHITE, "Color depth = %i",
        bitmap_color_depth(screen));

    //load the bitmap
      cowboy = load_bitmap("spacecowboy1.bmp", NULL);

    //draw the sprite
    for (n = 0; n < 11; n++)
    {
        y = 30 + size;
        stretch_sprite(screen, cowboy, size, y, size, size);
        textprintf(screen,font,size+size+10,y,WHITE,"%ix%i",
            (int)size,(int)size);
        size *= 1.4;
    }

    //delete the bitmap
    destroy_bitmap(cowboy);

    while(!key[KEY_ESC]);
    return 0;
}
END_OF_MAIN();
```

Drawing Flipped Sprites

Suppose you are writing a game called *Tank War* that features tanks able to move in four directions (north, south, east, and west), much like the game we have been building in the last few chapters. As you might recall, the last enhancement to the game in the last chapter added the ability to blit each tank image as a bitmap, which sped up the game significantly. Now imagine eliminating the east-, west-, and south-facing bitmaps from the game by simply drawing the north-facing bitmap in one of the four directions using a special version of `draw_sprite` for each one. In addition to the standard `draw_sprite`, you now have the use of three more functions to flip the sprite three ways:

```
void draw_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
void draw_sprite_h_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
void draw_sprite_vh_flip(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

Take a look at Figure 8.6, a shot from the *FlipSprite* program.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

int main()
{
    int x, y;

    //initialize the program
    allegro_init();
    install_keyboard();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);

    //load the bitmap
    BITMAP *panel = load_bitmap("panel.bmp", NULL);

    //draw the sprite
    draw_sprite(screen, panel, 200, 100);
    draw_sprite_h_flip(screen, panel, 200+128, 100);
    draw_sprite_v_flip(screen, panel, 200, 100+128);
    draw_sprite_vh_flip(screen, panel, 200+128, 100+128);

    //delete the bitmap
    destroy_bitmap(panel);

    while(!key[KEY_ESC]);
    return 0;
}
END_OF_MAIN();
```

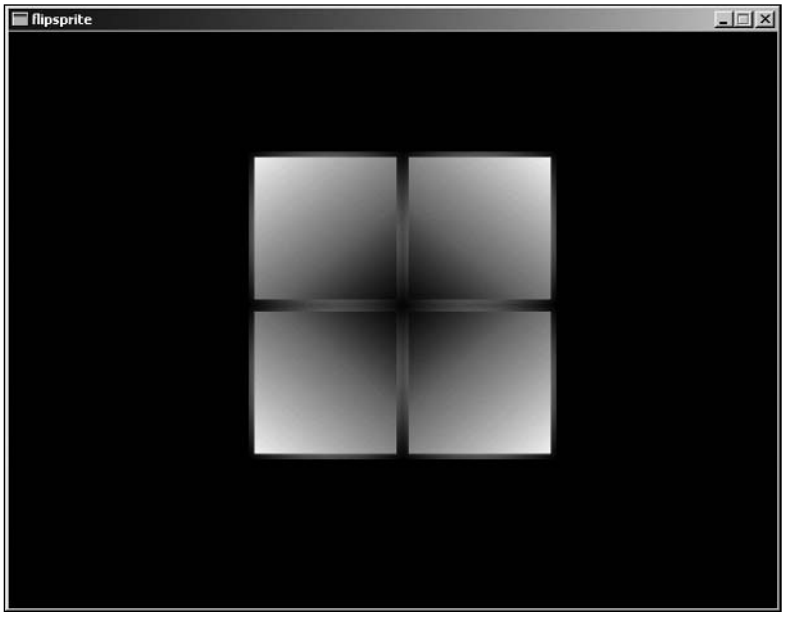


Figure 8.6 A single sprite is flipped both vertically and horizontally.

Drawing Rotated Sprites

Allegro has some very cool sprite manipulation functions that I'm sure you will have fun exploring. I have had to curtail my goofing off with all these functions in order to finish writing this chapter; otherwise, there might have been 90 sample programs to go over here! It really is incredibly fun to see all of the possibilities of these functions, which some might describe as "simple" or "2D."

Perhaps the most impressive (and incredibly useful) sprite manipulation function is `rotate_sprite`.

```
void rotate_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y,
    fixed angle);
```

This function rotates a sprite using an advanced algorithm that retains a high level of quality in the resulting sprite image. Most sprite rotation is done in a graphic editor by an artist because this is a time-consuming procedure in the middle of a high-speed game. The last thing you want slowing your game down is a sprite rotation occurring while you are rendering your sprites.

However, what about rotating and rendering your sprites at game startup, and then using the resulting bitmaps as a sprite array? That way, sprite rotation is provided at run time, and you only need to draw the first image of a sprite (such as a tank) facing north, and then rotate all of the angles you need for the game. For some programmers, this is a wonderful

and welcome feature because many of us are terrible artists. Chances are, if you are a good artist, you aren't a game programmer, and vice versa. Why would an artistically creative person be interested in writing code? Likewise, why would a programmer be interested in fooling with pixels? Naturally, there are exceptions (maybe you?), but in general, this is the way of things.

Who cares? Oh, right. Okay, let's try it out then. But first, here are the details. The `rotate_sprite` function draws the sprite image onto the destination bitmap with the top-left corner at the specified x and y position, rotated by the specified angle around its center. The tricky part is understanding that the angle does not represent a usual 360-degree circle; rather, it represents a set of floating-point angles from 0 to 256. If you would like to rotate a sprite at each of the usual 360 degrees of a circle, you can rotate it by (256 / 360 =) 0.711 for each angle.

Eight-Way Rotations

In reality, you will probably want a rotation scheme that generates eight, 16, or 32 rotation frames for each sprite. I've never seen a game that needed more than 32 frames for a full rotation. A highly spatial 2D shooter such as Atari's classic *Blasteroids* probably used 16 frames at most. Take a look at Figure 8.7 for an example of a tank sprite comprised of eight rotation frames.

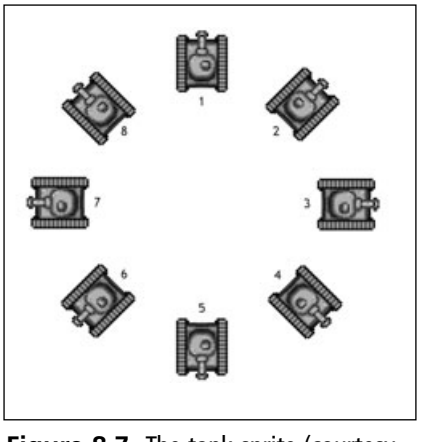


Figure 8.7 The tank sprite (courtesy of Ari Feldman) rotated in eight directions

When you want to generate eight frames, rotate each frame by 45 degrees more than the last one. This presumes that you are talking about a graphic editor, such as Paint Shop Pro, that is able to rotate images by any angle. Table 8.1 provides a rundown of the eight-frame rotation angles and the equivalent Allegro angles (based on 256). In the Allegro system, each frame is incremented by 32 degrees, which is actually easier to use from a programming perspective.

note

Even an eight-way sprite is a lot better than what we have done so far in *Tank War*, with only four pathetic sprite frames! What a travesty! Now that you've seen what is possible, I'm sure you have lost any ounce of respect you had for the game. Just hold on for a little while because you'll give the *Tank War* game a facelift at the end of this chapter with some proper sprites. It's almost time to do away with those ugly vector-based graphics once and for all!

Table 8.1 Eight-Frame Rotation Angles

Frame	Standard Angle (360)	Allegro Angle (256)
1	0	0
2	45	32
3	90	64
4	135	96
5	180	128
6	225	160
7	270	192
8	315	224

Sixteen-Way Rotations

A 16-way sprite is comprised of frames that are each incremented 22.5 degrees from the previous frame. Using this value, you can calculate the angles for an entire 16-way sprite, as shown in Figure 8.8.

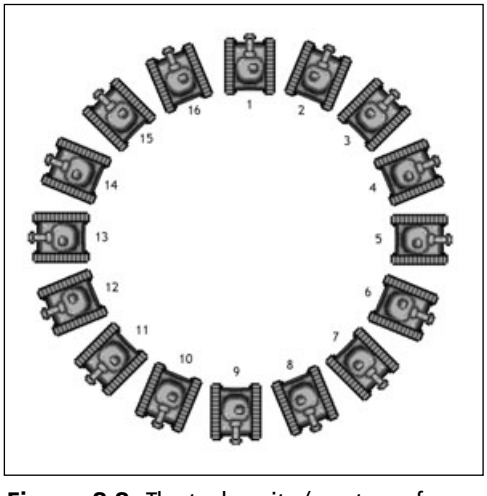


Figure 8.8 The tank sprite (courtesy of Ari Feldman) rotated in 16 directions

One glance at the column of Allegro angles in Table 8.2, and you can see why Allegro uses the 256-degree circle system instead of the 360-degree system; it is far easier to calculate the common angles used in games! Again, to determine what each angle should be, just divide the maximum angle (360 or 256, in either case) by the maximum number of frames to come up with a value for each frame.

Thirty-Two-Way Rotations

Although it's certainly a great goal to try for 24 or 32 frames of rotation in a 2D game, such as *Tank War*, each new set of frames added to the previous dimension of rotation adds a whole new complexity to the game. Remember, you need to calculate how the gun will fire in all of these directions! If your tank (or other sprite) needs to shoot in 32 directions, then you will have to calculate how that projectile will travel for each of those directions, too! To put it mildly, this is not easy to do. Combine that with the fact that the whole point of using higher rotations is simply to improve the quality of the game, and you might want to scale back to 16 if it becomes too difficult. I would suggest working from

Table 8.2 Sixteen-Frame Rotation Angles

Frame	Standard Angle (360)	Allegro Angle (256)
1	0.0	0
2	22.5	16
3	45.0	32
4	67.5	48
5	90.0	64
6	112.5	80
7	135.0	96
8	157.5	112
9	180.0	128
10	202.5	144
11	225.0	160
12	247.5	176
13	270.0	192
14	292.5	208
15	315.0	224
16	337.5	240

that common rotation count and adding more later if you have time, but don't delay the game just to get in all those frames so the game will be even better. My first rule is always to make the game work first, and then add cool factors (the bells and whistles).

Take a look at Figure 8.9 for an example of what a pre-rendered 32-frame sprite looks like. Each rotation frame is 11.25 degrees. In Allegro's 256-degree math, that's just a simple eight degrees per frame. You could write a simple loop to pre-rotate all of the images for *Tank War* using eight degrees, assuming you wanted to use a 32-frame tank.

That's a lot of sprites. In addition, they must all be perfectly situated in the bitmap image so that when it is drawn, the tank doesn't behave erratically with small jumps due to incorrect pixel alignment on each frame. What's a good solution? It probably would be a good idea to simply use a single tank image and rotate it through all 32 frames when the game starts up, and then store the rotation frames in a sprite array. Allegro makes it easy to do this. This is also a terrific solution when you are working on smaller platforms that have limited memory. Don't be surprised by the possibility that if you are serious about game programming, you might end up writing games for cell phones, Nokia N-Gage, and other small platforms where memory is a premium. Of course, Allegro isn't available for those platforms, but speaking in general terms, rotating a sprite based on a single image is very

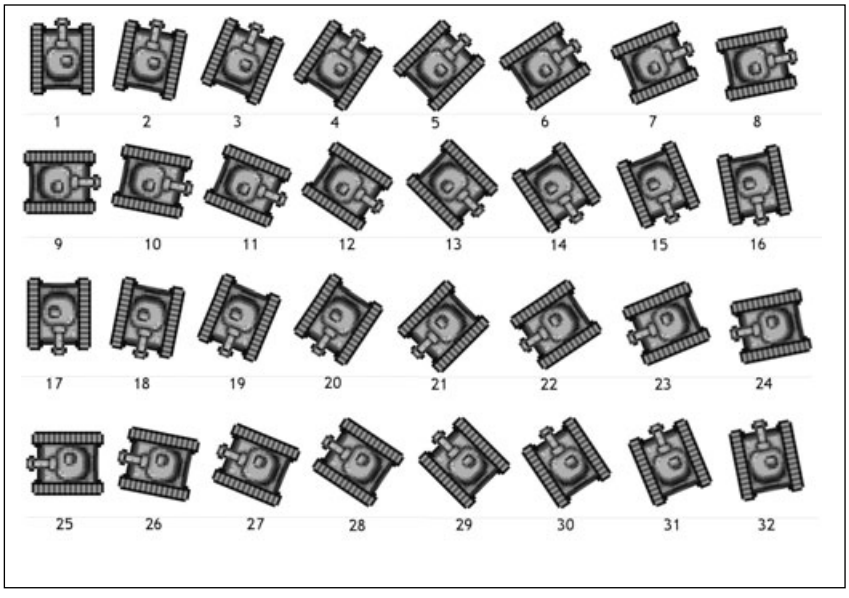


Figure 8.9 The tank sprite (courtesy of Ari Feldman) rotated in 32 directions

efficient and a smart way to develop under limited resources. You can get away with a lot of sloppy code under a large operating system, when it is assumed that the player must have a minimum amount of memory. (512 MB and 1 GB are common on Windows machines nowadays.)

The RotateSprite Program

Now it's time to put some of this newfound knowledge to use in an example program. This program is called *RotateSprite*, and it simply demonstrates the `rotate_sprite` function. You can use the left and right arrow keys to rotate the sprite in either direction. There is no fixed angle used in this sample program, but the angle is adjusted by 0.1 degree in either direction, giving it a nice steady rotation rate that shouldn't be too fast. If you are using a slower PC, you can increase the angle. Note that a whole number angle will go so fast that you'll have to slow down the program the hard way, using the `rest` function. Take a look at Figure 8.10, which shows the *RotateSprite* program running.

The only aspect of the code listing for the *RotateSprite* program that I want you to keep an eye out for is the actual call to `rotate_sprite`. I have set the two lines that use `rotate_sprite` in bold so you will be able to identify them easily. Note the last parameter, `itofix(angle)`. This extremely important function converts the angle to Allegro's fixed 16.16 numeric format used by `rotate_sprite`. You will want to pass your floating-point value (float, double) to `itofix` to convert it to a fixed-point value.

Figure 8.10 The tank sprite (courtesy of Ari Feldman) is rotated with the arrow keys.

tip

Fixed-point is much faster than floating-point—or so says the theory, which I do not subscribe to due to the modern floating-point power of processors. Remember that you must use `itofix` with all of the rotation functions.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

void main(void)
{
    int x, y;
    float angle = 0;

    //initialize program
    allegro_init();
    install_keyboard();
    set_color_depth(32);
```

```
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    textout(screen, font, "Rotate: LEFT / RIGHT arrow keys",
        0, 0, WHITE);

    //load tank sprite
    BITMAP *tank = load_bitmap("tank.bmp", NULL);

    //calculate center of screen
    x = SCREEN_W/2 - tank->w/2;
    y = SCREEN_H/2 - tank->h/2;

    //draw tank at starting location
    rotate_sprite(screen, tank, x, y, 0);

    //main loop
    while(!key[KEY_ESC])
    {
        //wait for keypress
        if (keypressed())
        {
            //left arrow rotates left
            if (key[KEY_LEFT])
            {
                angle -= 0.1;
                if (angle < 0) angle = 256;
                rotate_sprite(screen, tank, x, y, itofix(angle));
            }

            //right arrow rotates right
            if (key[KEY_RIGHT])
            {
                angle += 0.1;
                if (angle > 256) angle = 0;
                rotate_sprite(screen, tank, x, y, itofix(angle));
            }

            //display angle
            textprintf(screen, font, 0, 10, WHITE, "Angle = %f", angle);
        }
    }
}
END_OF_MAIN()
```

Additional Rotation Functions

Allegro is generous with so many great functions, and that includes alternative forms of the `rotate_sprite` function. Here you have a rotation function that includes vertical flip, another rotation function that includes scaling, and a third function that does both scaling and vertical flip while rotating. Whew! You can see from these functions that the creators of Allegro were not artists, so they incorporated all of these wonderful functions to make it easier to conjure artwork for a game! These functions are similar to `rotate_sprite` so I won't bother with a sample program. You already understand how it works, right?

```
void rotate_sprite_v_flip(BITMAP *bmp, BITMAP *sprite,
    int x, int y, fixed angle)
```

The preceding function rotates and also flips the image vertically. To flip horizontally, add `itofix(128)` to the angle. To flip in both directions, use `rotate_sprite()` and add `itofix(128)` to its angle.

```
void rotate_scaled_sprite(BITMAP *bmp, BITMAP *sprite,
    int x, int y, fixed angle, fixed scale)
```

The preceding function rotates an image and scales (stretches to fit) the image at the same time.

```
void rotate_scaled_sprite_v_flip(BITMAP *bmp, BITMAP *sprite,
    int x, int y, fixed angle, fixed scale)
```

The preceding function rotates the image while also scaling and flipping it vertically, simply combining the functionality of the previous two functions.

Drawing Pivoted Sprites

Allegro provides the functionality to pivot sprites and images. What does pivot mean? The *pivot point* is the location on the image where rotation occurs. If a sprite is 64×64 pixels, then the default pivot point is at 31×31 (accounting for zero); a sprite sized at 32×32 would have a default pivot point at 15×15. The `pivot` functions allow you to change the position of the pivot where rotation takes place.

```
void pivot_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y,
     int cx, int cy, fixed angle)
```

The `x` and `y` values specify where the sprite is drawn, while `cx` and `cy` specify the pivot *within* the sprite (not globally to the screen). Therefore, if you have a 32×32 sprite, you can draw it anywhere on the screen, but the pivot points `cx` and `cy` should be values of 0 to 31.

The PivotSprite Program

The *PivotSprite* program demonstrates how to use the `pivot_sprite` function by drawing two blue lines on the screen, showing the pivot point on the sprite. You can use the arrow

keys to adjust the pivot point and see how the sprite reacts while it is rotating in real time (see Figure 8.11).

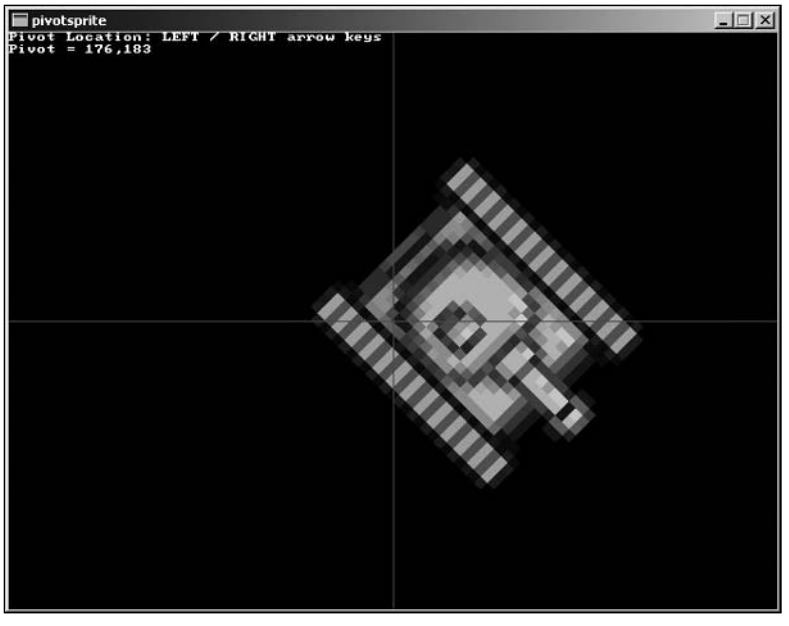


Figure 8.11 The *PivotSprite* program demonstrates how to adjust the pivot point. Image courtesy of Ari Feldman.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLUE makecol(64,64,255)

void main(void)
{
    int x, y;
    int pivotx, pivoty;
    float angle = 0;

    //initialize program
    allegro_init();
    install_keyboard();
    set_color_depth(32);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
```

```
//load tank sprite
BITMAP *tank = load_bitmap("tank.bmp", NULL);

//calculate center of screen
x = SCREEN_W/2;
y = SCREEN_H/2;
pivotx = tank->w/2;
pivoty = tank->h/2;

//main loop
while(!key[KEY_ESC])
{
    //wait for keypress
    if (keypressed())
    {
        //left arrow moves pivot left
        if (key[KEY_LEFT])
        {
            pivotx -= 2;
            if (pivotx < 0)
                pivotx = 0;
        }

        //right arrow moves pivot right
        if (key[KEY_RIGHT])
        {
            pivotx += 2;
            if (pivotx > tank->w-1)
                pivotx = tank->w-1;
        }

        //up arrow moves pivot up
        if (key[KEY_UP])
        {
            pivoty -= 2;
            if (pivoty < 0)
                pivoty = 0;
        }

        //down arrow moves pivot down
        if (key[KEY_DOWN])
        {
            pivoty += 2;
```

```
                if (pivoty > tank->h-1)
                    pivoty = tank->h-1;
            }
        }

        //pivot/rotate the sprite
        angle += 0.5;
        if (angle > 256) angle = 0;
        pivot_sprite(screen, tank, x, y, pivotx, pivoty, itofix(angle));

        //draw the pivot lines
        hline(screen, 0, y, SCREEN_W-1, BLUE);
        vline(screen, x, 0, SCREEN_H-1, BLUE);

        //display information
        textout(screen, font, "Pivot Location: LEFT / RIGHT arrow keys",
            0, 0, WHITE);
        textprintf(screen, font, 0, 10, WHITE, "Pivot = %3d,%3d ",
            pivotx, pivoty);
        rest(1);
    }
}
END_OF_MAIN()
```

Additional Pivot Functions

As usual, Allegro provides everything including the clichéd kitchen sink. Here are the additional pivot functions that you might have already expected to see, given the consistency of Allegro in this matter. Here you have three functions—pivot with vertical flip, pivot with scaling, and pivot with scaling and vertical flip. It's nice to know that Allegro is so consistent, so any time you are in need of a special sprite manipulation within your game, you are certain to be able to accomplish it using a combination of rotation, pivot, scaling, and flipping functions that have been provided.

```
void pivot_sprite_v_flip(BITMAP *bmp, BITMAP *sprite, int x, int y,
        int cx, int cy, fixed angle);


void pivot_scaled_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y,
        int cx, int cy, fixed angle, fixed scale));


void pivot_scaled_sprite_v_flip(BITMAP *bmp, BITMAP *sprite,
    int x, int y, fixed angle, fixed scale)
```

Drawing Translucent Sprites

Allegro provides many special effects that you can apply to sprites, as you saw in the previous sections. The next technique is unusual enough to warrant a separate discussion. This section explains how to draw sprites with translucent alpha blending. Two more special effects (sprite lighting and Gouraud shading) are covered in the next chapter.

Translucency is a degree of "see-through" that differs from *transparency*, which is entirely see-through. Think of the glass in a window as being translucent, while an open window is transparent. There is quite a bit of work involved in making a sprite translucent, and I'm not entirely sure it's necessary for a game to use this feature, which is most definitely a drain on the graphics hardware. Although a late-model video card can handle translucency, or *alpha blending*, with ease, there is still the issue of supporting older computers or those with non-standard video cards. As such, many 2D games have steered clear of using this feature. One of the problems with translucency in a software implementation is that you must prepare both bitmaps before they will render with translucency. Some hardware solutions are likely available, but they are not provided for in Allegro.

Translucency is provided by the `draw_trans_sprite` function.

```
void draw_trans_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);
```

Unfortunately, it's not quite as cut-and-dried as this simple function makes it appear. To use translucency, you have to use an alpha channel blender, and even the Allegro documentation is elusive in describing how this works. Suffice it to say, translucency is not something you would probably want to use in a game because it was really designed to work between just two bitmaps. You could use the same background image with multiple foreground sprites that are blended with the background using the alpha channel, but each sprite must be adjusted pixel by pixel when the program starts. This is a special effect that you might find a use for, but I would advise against using it in the main loop of a game.

Here is the source code for the *TransSprite* program, shown in Figure 8.12. I will explain how it works after the listing.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

int main()
{
    int x, y, c, a;

    //initialize
    allegro_init();
    install_keyboard();
```

Figure 8.12 The *TransSprite* program demonstrates how to draw a translucent sprite.

```
install_mouse();
set_color_depth(32);
set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);

//load the background bitmap
BITMAP *background = load_bitmap("mustang.bmp", NULL);

//load the translucent foreground image
BITMAP *alpha = load_bitmap("alpha.bmp", NULL);
BITMAP *sprite = create_bitmap(alpha->w, alpha->h);

//set the alpha channel blend values
drawing_mode(DRAW_MODE_TRANS, NULL, 0, 0);
set_write_alpha_blender();
//blend the two bitmap alpha channels
for (y=0; y<alpha->h; y++) {
    for (x=0; x<alpha->w; x++) {
        //grab the pixel color
        c = getpixel(alpha, x, y);
        a = getr(c) + getg(c) + getb(c);
        //find the middle alpha value
        a = MID(0, a/2-128, 255);
        //copy the alpha-enabled pixel to the sprite
        putpixel(sprite, x, y, a);
    }
```

```
    }

    //create a double buffer bitmap
    BITMAP *buffer = create_bitmap(SCREEN_W, SCREEN_H);

    //draw the background image
    blit(background, buffer, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

    while (!key[KEY_ESC])
    {
        //get the mouse coordinates
        x = mouse_x - sprite->w/2;
        y = mouse_y - sprite->h/2;

        //draw the translucent image
        set_alpha_blender();
        draw_trans_sprite(buffer, sprite, x, y);

        //draw memory buffer to the screen
        blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

        //restore the background
        blit(background, buffer, x, y, x, y, sprite->w, sprite->h);
    }

    destroy_bitmap(background);
    destroy_bitmap(sprite);
    destroy_bitmap(buffer);
    destroy_bitmap(alpha);

    return 0;
}
END_OF_MAIN();
```

Now for some explanation. First, the program loads the background image (called "background"), followed by the foreground sprite (called "alpha"). A new image called "sprite" is created with the same resolution as the background; it receives the alpha-channel information. The drawing mode is set to `DRAW_MODE_TRANS` to enable translucent drawing with the graphics functions (`putpixel`, `line`, and so on). The pixels are then copied from the alpha image into the sprite image.

After that, another new image called "buffer" is created and the background is blitted to it. At this point, the main loop starts. Within the loop, the mouse is polled to move the

sprite around on the screen, demonstrating the alpha blending. The actual translucency is accomplished by two functions.

```
set_alpha_blender();
draw_trans_sprite(buffer, sprite, x, y);
```

The alpha blender is enabled before `draw_trans_sprite` is called, copying the "sprite" image onto the buffer. The memory buffer is blitted to the screen, and then the background is restored for the next iteration through the loop.

```
blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
```

Enhancing Tank War

Now it's time to use the new knowledge you have gained in this chapter to enhance *Tank War*. First, how about a quick recap on the state of the game? Take a look at Figure 8.13, showing *Tank War* as it appeared in the last chapter.

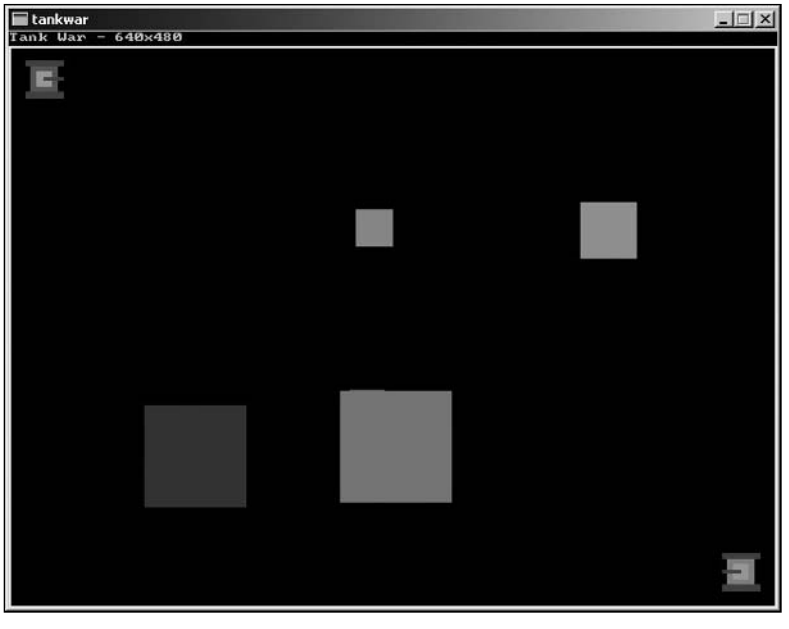


Figure 8.13 The last version of *Tank War*

Not very attractive, is it? It looks like something that would run on an Atari 2600. I have been skirting the issue of using true bitmaps and sprites in *Tank War* since it was first conceived several chapters ago. Now it's time to give this pathetic game a serious upgrade!

What's New?

First, to upgrade the game, I made a design decision to strip out the pixel collision code and leave the battlefield blank for this enhancement. The game will look better overall with the eight-way tank sprites, but the obstacles will no longer be present. Take a look at Figure 8.14, showing a tank engulfed in an explosion.

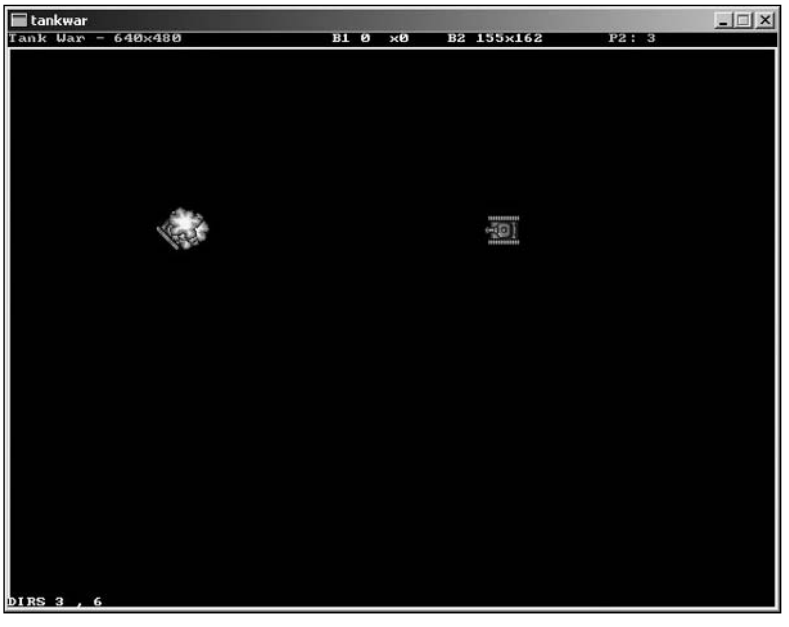


Figure 8.14 *Tank War* now features bitmap-based sprites.

It's really time to move out of the vector theme entirely. Because I haven't covered sprite-based collision detection yet to determine when a tank or bullet hits an actual sprite (rather than just checking the color of the pixel at the bullet's location), I'll leave that for the next chapter, in which I'll get into sprite collision as well as animation and other essential sprite behaviors. What that means right now is that *Tank War* is getting smaller and less complicated, at least for the time being! By stripping the pixel collision code, the source code is shortened considerably. You will lose `checkpath`, `clearpath`, and `setupdebris`, three key functions from the first version of the game. (Although they are useful as designed, they are not very practical.) In fact, that first version had a lot of promise and could have been improved with just the vector graphics upon which it was based. If you are still intrigued by the old-school game technology that used vector graphics, I encourage you to enhance the game and see what can be done with vectors alone. I am forging ahead because the topics of each chapter demand it, but we have not fully explored all the possibilities by any means.

New Tanks

Now what about the new changes for *Tank War*? This will be the third enhancement to the game, but it is somewhat of a backward step in gameplay because there are no longer any obstacles on the battlefield. However, the tanks are no longer rendered with vector graphics functions; rather, they are loaded from a bitmap file. This enhancement also includes a new bitmap for the bullets and explosions. The source code for the game is much shorter than it was before, but due to all the changes, I will provide the entire listing here, rather than just highlighting the changes (as was the case with the previous two enhancements). Much of the original source code is the same, but many seemingly untouched functions have had minor changes to parameters and lines of code that are too numerous to point out. Figure 8.15 shows both tanks firing their newly upgraded weapons.

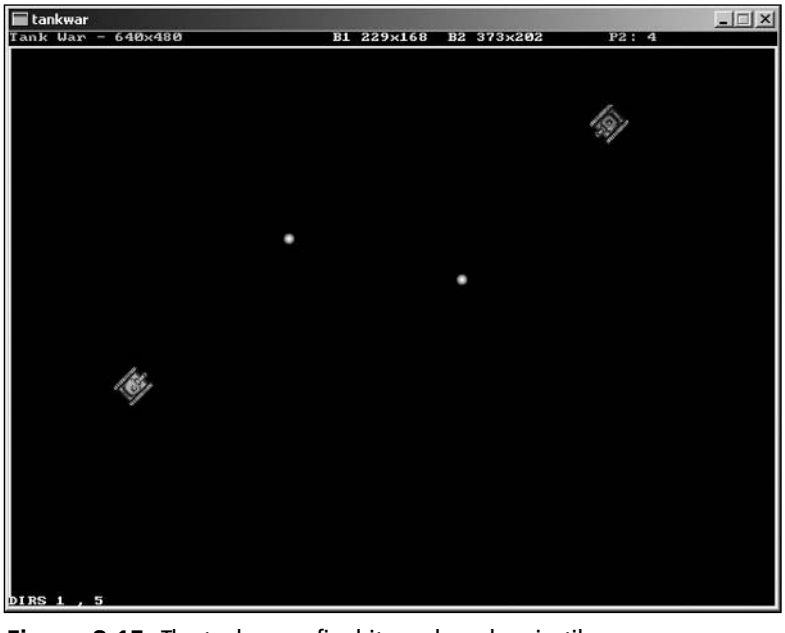


Figure 8.15 The tanks now fire bitmap-based projectiles.

If you'll take a closer look at Figure 8.15, you might notice that the same information is displayed at the top of the screen (name, resolution, bullet locations, and score). I have added a small debug message to the bottom-left corner of the game screen, showing the direction each tank is facing. Since the game now features eight-way directional movement rather than just four-way, I found it useful to display the direction each tank is facing because the new directions required modifications to the `movetank` and `updatebullet` functions.

New Sprites

Figure 8.16 shows the new projectile sprite, and Figure 8.17 shows the new explosion sprite. These might not look like much zoomed in close like this, but they look great in the game.

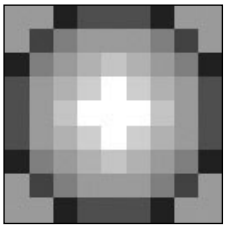


Figure 8.16
The new projectile (bullet) sprite

Figure 8.17
The new explosion sprite

Modifying the Source Code

Here is the new version of tankwar.h, the header file used by the game. You should be able to simply modify your last version to make it look like this. You might notice the new bullet and explosion bitmaps in addition to the changes to `tank_bmp`, which now supports eight bitmaps, one for each direction. Now that color no longer plays a part in drawing the tanks, the color variable has been removed from the tank structure, `tagTank`. The three function prototypes for collision detection are included: `clearpath`, `checkpath`, and `setupdebris`. Since the game loop has been sped up, I have also modified `BULLETSPEED` so that it is now six instead of 10 (which was too jumpy).

The Tank War Header File

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Chapter 8 - Tank War Header (Enhancement 3)
////////////////////////////////////////////////////////////////////////

#ifndef _TANKWAR_H
#define _TANKWAR_H

#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

//define some game constants
#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define MAXSPEED 2
#define BULLETSPEED 6
```

```
//define some colors
#define TAN makecol(255,242,169)
#define BURST makecol(255,189,73)
#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

//define tank structure
struct tagTank
{
    int x,y;
    int dir,speed;
    int score;

} tanks[2];

//define bullet structure
struct tagBullet
{
    int x,y;
    int alive;
    int xspd,yspd;

} bullets[2];

//declare some variables
int gameover = 0;

//sprite bitmaps
BITMAP *tank_bmp[2][8];
BITMAP *bullet_bmp;
BITMAP *explode_bmp;

//function prototypes
void drawtank(int num);
void erasetank(int num);
void movetank(int num);
void explode(int num, int x, int y);
void updatebullet(int num);
void fireweapon(int num);
void forward(int num);
void backward(int num);
void turnleft(int num);
```

```
void turnright(int num);
void getinput();
void setuptanks();
void score(int);
void setupscreen();

#endif
```

The Tank War Source Code File

Now I want to focus on the new source code for *Tank War*. As I mentioned previously, nearly every function has been modified for this new version, so if you have any problems running it after you modify your last copy of the game, you have likely missed some change in the following listing. As a last resort, you can load the project off the CD-ROM, located in \chapter08\tankwar for your favorite compiler (devcpp, kdevelop, or msvc).

I'll walk you through each major change in the game, starting with the first part. Here you have a new `drawtank`, `erasetank`, and `movetank` that support sprites and eight directions.

```
//////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Chapter 8 - Tank War Game (Enhancement 3)
//////////////////////////////////////////////////////////////////////

#include "tankwar.h"

//////////////////////////////////////////////////////////////////////
// drawtank function
// display the tank bitmap in the current direction
//////////////////////////////////////////////////////////////////////
void drawtank(int num)
{
    int dir = tanks[num].dir;
    int x = tanks[num].x-15;
    int y = tanks[num].y-15;
    draw_sprite(screen, tank_bmp[num][dir], x, y);
}

//////////////////////////////////////////////////////////////////////
// erasetank function
// erase the tank using rectfill
//////////////////////////////////////////////////////////////////////
void erasetank(int num)
```

```
{
    int x = tanks[num].x-17;
    int y = tanks[num].y-17;
    rectfill(screen, x, y, x+33, y+33, BLACK);
}

//////////////////////////////////////////////////////////////////////
// movetank function
// move the tank in the current direction
//////////////////////////////////////////////////////////////////////
void movetank(int num){
    int dir = tanks[num].dir;
    int speed = tanks[num].speed;

    //update tank position based on direction
    switch(dir)
    {
        case 0:
            tanks[num].y -= speed;
            break;
        case 1:
            tanks[num].x += speed;
            tanks[num].y -= speed;
            break;
        case 2:
            tanks[num].x += speed;
            break;
        case 3:
            tanks[num].x += speed;
            tanks[num].y += speed;
            break;
        case 4:
            tanks[num].y += speed;
            break;
        case 5:
            tanks[num].x -= speed;
            tanks[num].y += speed;
            break;
        case 6:
            tanks[num].x -= speed;
            break;
        case 7:
            tanks[num].x -= speed;
```

```
            tanks[num].y -= speed;
            break;
    }

    //keep tank inside the screen
    if (tanks[num].x > SCREEN_W-22)
    {
        tanks[num].x = SCREEN_W-22;
        tanks[num].speed = 0;
    }
    if (tanks[num].x < 22)
    {
        tanks[num].x = 22;
        tanks[num].speed = 0;
    }
    if (tanks[num].y > SCREEN_H-22)
    {
        tanks[num].y = SCREEN_H-22;
        tanks[num].speed = 0;
    }
    if (tanks[num].y < 22)
    {
        tanks[num].y = 22;
        tanks[num].speed = 0;
    }
}
```

The next section of code includes highly modified versions of `explode`, `updatebullet`, and `fireweapon`, which, again, must support all eight directions. One significant change is that `explode` no longer includes the code that checks for a tank hit—that code has been moved to `updatebullet`. You might also notice in `explode` that the explosion is now a bitmap rather than a random-colored rectangle. This small effect alone dramatically improves the game.

```
//////////////////////////////////////////////////////////////////////
// explode function
// display an explosion image
//////////////////////////////////////////////////////////////////////
void explode(int num, int x, int y)
{
    int n;

    //load explode image
    if (explode_bmp == NULL)
    {
```

```
        explode_bmp = load_bitmap("explode.bmp", NULL);
    }

    //draw the explosion bitmap several times
    for (n = 0; n < 5; n++)
    {
        rotate_sprite(screen, explode_bmp,
            x + rand()%10 - 20, y + rand()%10 - 20,
            itofix(rand()%255));

        rest(30);
    }

    //clear the explosion
    circlefill(screen, x, y, 50, BLACK);

}

////////////////////////////////////////////////////////////////////
// updatebullet function
// update the position of a bullet
////////////////////////////////////////////////////////////////////
void updatebullet(int num)
{
    int x = bullets[num].x;
    int y = bullets[num].y;

    //is the bullet active?
    if (!bullets[num].alive) return;

    //erase bullet
    rectfill(screen, x, y, x+10, y+10, BLACK);

    //move bullet
    bullets[num].x += bullets[num].xspd;
    bullets[num].y += bullets[num].yspd;
    x = bullets[num].x;
    y = bullets[num].y;

    //stay within the screen
    if (x < 6 || x > SCREEN_W-6 || y < 20 || y > SCREEN_H-6)
    {
        bullets[num].alive = 0;
```

```
            return;
        }

        //look for a direct hit using basic collision
        //tank is either 0 or 1, so negative num = other tank
        int tx = tanks[!num].x;
        int ty = tanks[!num].y;
        if (x > tx-16 && x < tx+16 && y > ty-16 && y < ty+16)
        {
            //kill the bullet
            bullets[num].alive = 0;

            //blow up the tank
            explode(num, x, y);
            score(num);
        }
        else
        //if no hit then draw the bullet
        {
            //draw bullet sprite
            draw_sprite(screen, bullet_bmp, x, y);

            //update the bullet positions (for debugging)
            textprintf(screen, font, SCREEN_W/2-50, 1, TAN,
                "B1 %-3dx%-3d  B2 %-3dx%-3d",
                bullets[0].x, bullets[0].y,
                bullets[1].x, bullets[1].y);
        }
    }


    ////////////////////////////////////////////////////////////////////
    // fireweapon function
    // set bullet direction and speed and activate it
    ////////////////////////////////////////////////////////////////////
    void fireweapon(int num)
    {
        int x = tanks[num].x;
        int y = tanks[num].y;

        //load bullet image if necessary
        if (bullet_bmp == NULL)
        {
```

```
    bullet_bmp = load_bitmap("bullet.bmp", NULL);
}

//ready to fire again?
if (!bullets[num].alive)
{
    bullets[num].alive = 1;

    //fire bullet in direction tank is facing
    switch (tanks[num].dir)
    {
        //north
        case 0:
            bullets[num].x = x-2;
            bullets[num].y = y-22;
            bullets[num].xspd = 0;
            bullets[num].yspd = -BULLETSPEED;
            break;
        //NE
        case 1:
            bullets[num].x = x+18;
            bullets[num].y = y-18;
            bullets[num].xspd = BULLETSPEED;
            bullets[num].yspd = -BULLETSPEED;
            break;
        //east
        case 2:
            bullets[num].x = x+22;
            bullets[num].y = y-2;
            bullets[num].xspd = BULLETSPEED;
            bullets[num].yspd = 0;
            break;
        //SE
        case 3:
            bullets[num].x = x+18;
            bullets[num].y = y+18;
            bullets[num].xspd = BULLETSPEED;
            bullets[num].yspd = BULLETSPEED;
            break;
        //south
        case 4:
            bullets[num].x = x-2;
            bullets[num].y = y+22;
```

```
                bullets[num].xspd = 0;
                bullets[num].yspd = BULLETSPEED;
                break;
            //SW
            case 5:
                bullets[num].x = x-18;
                bullets[num].y = y+18;
                bullets[num].xspd = -BULLETSPEED;
                bullets[num].yspd = BULLETSPEED;
                break;
            //west
            case 6:
                bullets[num].x = x-22;
                bullets[num].y = y-2;
                bullets[num].xspd = -BULLETSPEED;
                bullets[num].yspd = 0;
                break;
            //NW
            case 7:
                bullets[num].x = x-18;
                bullets[num].y = y-18;
                bullets[num].xspd = -BULLETSPEED;
                bullets[num].yspd = -BULLETSPEED;
                break;
        }
    }
}
```

The next section of code covers the keyboard input code, including `forward`, `backward`, `turnleft`, `turnright`, and `getinput`. These functions are largely the same as before, but they now must support eight directions (evident in the `if` statement within `turnleft` and `turnright`).

```
//////////////////////////////////////////////////////////////////////
// forward function
// increase the tank's speed
//////////////////////////////////////////////////////////////////////
void forward(int num)
{
    tanks[num].speed++;
    if (tanks[num].speed > MAXSPEED)
        tanks[num].speed = MAXSPEED;
}
```

```
////////////////////////////////////////////////////////////////////////
// backward function
// decrease the tank's speed
////////////////////////////////////////////////////////////////////////
void backward(int num)
{
    tanks[num].speed--;
    if (tanks[num].speed < -MAXSPEED)
        tanks[num].speed = -MAXSPEED;
}

////////////////////////////////////////////////////////////////////////
// turnleft function
// rotate the tank counter-clockwise
////////////////////////////////////////////////////////////////////////
void turnleft(int num)
{
//***
    tanks[num].dir--;
    if (tanks[num].dir < 0)
        tanks[num].dir = 7;
}

////////////////////////////////////////////////////////////////////////
// turnright function
// rotate the tank clockwise
////////////////////////////////////////////////////////////////////////
void turnright(int num)
{
    tanks[num].dir++;
    if (tanks[num].dir > 7)
        tanks[num].dir = 0;
}

////////////////////////////////////////////////////////////////////////
// getinput function
// check for player input keys (2 player support)
////////////////////////////////////////////////////////////////////////
void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])   gameover = 1;
```

```
    //WASD - SPACE keys control tank 1
    if (key[KEY_W])     forward(0);
    if (key[KEY_D])     turnright(0);
    if (key[KEY_A])     turnleft(0);
    if (key[KEY_S])     backward(0);
    if (key[KEY_SPACE]) fireweapon(0);

    //arrow - ENTER keys control tank 2
    if (key[KEY_UP])    forward(1);
    if (key[KEY_RIGHT]) turnright(1);
    if (key[KEY_DOWN])  backward(1);
    if (key[KEY_LEFT])  turnleft(1);
    if (key[KEY_ENTER]) fireweapon(1);

    //short delay after keypress
    rest(20);
}
```

The next short code section includes the `score` function that is used to update the score for each player.

```
////////////////////////////////////////////////////////////////////
// score function
// add a point to a player's score
////////////////////////////////////////////////////////////////////
void score(int player)
{
    //update score
    int points = ++tanks[player].score;

    //display score
    textprintf(screen, font, SCREEN_W-70*(player+1), 1,
        BURST, "P%d: %d", player+1, points);
}
```

The `setuptanks` function has changed dramatically from the last version because that is where the new tank bitmaps are loaded. Since this game uses the `rotate_sprite` function to generate the sprite images for all eight directions, this function takes care of that by first creating each image and then blitting the source tank image into each new image with a specified rotation angle. The end result is two tanks fully rotated in eight directions.

```
////////////////////////////////////////////////////////////////////
// setuptanks function
// load tank bitmaps and position the tank
////////////////////////////////////////////////////////////////////
```

```
void setuptanks()
{
    int n;

    //configure player 1's tank
    tanks[0].x = 30;
    tanks[0].y = 40;
    tanks[0].speed = 0;
    tanks[0].score = 0;
    tanks[0].dir = 3;

    //load first tank bitmap
    tank_bmp[0][0] = load_bitmap("tank1.bmp", NULL);

    //rotate image to generate all 8 directions
    for (n=1; n<8; n++)
    {
        tank_bmp[0][n] = create_bitmap(32, 32);
        clear_bitmap(tank_bmp[0][n]);
        rotate_sprite(tank_bmp[0][n], tank_bmp[0][0],
            0, 0, itofix(n*32));
    }

    //configure player 2's tank
    tanks[1].x = SCREEN_W-30;
    tanks[1].y = SCREEN_H-30;
    tanks[1].speed = 0;
    tanks[1].score = 0;
    tanks[1].dir = 7;

    //load second tank bitmap
    tank_bmp[1][0] = load_bitmap("tank2.bmp", NULL);

    //rotate image to generate all 8 directions
    for (n=1; n<8; n++)
    {
        tank_bmp[1][n] = create_bitmap(32, 32);
        clear_bitmap(tank_bmp[1][n]);
        rotate_sprite(tank_bmp[1][n], tank_bmp[1][0],
            0, 0, itofix(n*32));
    }
}
```

The next section of the code includes the `setupscreen` function. The most important change to this function is the inclusion of a single line calling `set_color_depth(32)`, which causes the game to run in 32-bit color mode. Note that if you don't have a 32-bit video card, you might want to change this to 16 (which will still work).

```
////////////////////////////////////////////////////////////////////
// setupscreen function
// set up the graphics mode and draw the game screen
////////////////////////////////////////////////////////////////////
void setupscreen()
{
    //set video mode
    set_color_depth(32);
    int ret = set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //print title
    textprintf(screen, font, 1, 1, BURST,
        "Tank War - %dx%d", SCREEN_W, SCREEN_H);

    //draw screen border
    rect(screen, 0, 12, SCREEN_W-1, SCREEN_H-1, TAN);
    rect(screen, 1, 13, SCREEN_W-2, SCREEN_H-2, TAN);
}
```

Finally, the last section of code in the third enhancement to *Tank War* includes the all-important `main` function. Several changes have been made in `main`, notably the removal of the calls to `clearpath` (which checked for bullet hits by looking directly at pixel color). The call to `rest` now has a value of 10 to speed up the game a bit in order to have smoother bullet trajectories. There is also a line of code that displays the direction of each tank, as I explained previously.

```
////////////////////////////////////////////////////////////////////
// main function
// start point of the program
////////////////////////////////////////////////////////////////////
void main(void)
{
    //initialize the game
```

```
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    setupscreen();
    setuptanks();

    //game loop
    while(!gameover)
    {
        textprintf(screen, font, 0, SCREEN_H-10, WHITE,
            "DIRS %d , %d", tanks[0].dir, tanks[1].dir);
        //erase the tanks
        erasetank(0);
        erasetank(1);

        //move the tanks
        movetank(0);
        movetank(1);

        //draw the tanks
        drawtank(0);
        drawtank(1);

        //update the bullets
        updatebullet(0);
        updatebullet(1);

        //check for keypresses
        if (keypressed())
            getinput();

        //slow the game down
        rest(10);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Summary

This marks the end of perhaps the most interesting chapter so far, at least in my opinion. The introduction to sprites that you have received in this chapter provided the basics without delving too deeply into sprite programming theory. The next chapter covers some advanced sprite programming topics, including the sorely needed collision detection. I will also get into sprite animation in the next chapter. There are many more changes on the way for *Tank War* as well. The next several chapters will provide a huge amount of new functionality that you can use to greatly enhance *Tank War*, making it into a truly top-notch game with a scrolling background, animated tanks, a radar screen, and many more new features!

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the term given to a small image that is moved around on the screen?
   A. Bitmap
   B. Sprite
   C. Fairy
   D. Mouse cursor

2. Which function draws a sprite?
   A. `draw_sprite`
   B. `show_sprite`
   C. `display_sprite`
   D. `blit_sprite`

3. What is the term for drawing all but a certain color of pixel from one bitmap to another?
   A. Alpha blending
   B. Translucency
   C. Transparency
   D. Telekinesis

4. Which function draws a scaled sprite?
   A. `stretch_sprite`
   B. `draw_scaled_sprite`
   C. `draw_stretched_sprite`
   D. `scale_sprite`

5. Which function draws a vertically-flipped sprite?
   A. `draw_vertical_flip`
   B. `draw_sprite_v_flip`
   C. `flip_v_sprite`
   D. `draw_flipped_sprite`

6. Which function draws a rotated sprite?
   A. `rotate_angle`
   B. `draw_rotated_sprite`
   C. `draw_rotation`
   D. `rotate_sprite`

7. Which function draws a sprite with both rotation and scaling?
   A. `draw_sprite_rotation_scaled`
   B. `rotate_scaled_sprite`
   C. `draw_rotated_scaled_sprite`
   D. `scale_rotate_sprite`

8. What function draws a pivoted sprite?
   A. `draw_pivoted_sprite`
   B. `draw_pivot_sprite`
   C. `pivot_sprite`
   D. `draw_sprite_pivot`

9. Which function draws a pivoted sprite with scaling and vertical flip?
   A. `pivot_scaled_sprite_v_flip`
   B. `pivot_stretch_v_flip_sprite`
   C. `draw_scaled_pivoted_flipped_sprite`
   D. `scale_pivot_v_flip_sprite`

10. Which function draws a sprite with translucency (alpha blending)?
    A. `alpha_blend_sprite`
    B. `draw_trans_sprite`
    C. `draw_alpha`
    D. `trans_sprite`

This page intentionally left blank

CHAPTER 9

Advanced Sprite Programming: Animation, Compiled Sprites, and Collision Detection

If Chapter 7 provided the foundation for developing bitmap-based games, then Chapter 8 provided the frame, walls, plumbing, and wiring. (House analogies are frequently used to describe software development, so they may be used to describe game programming as well.) Therefore, what you need from this chapter are the sheetrock, finishing, painting, stucco, roof tiles, appliances, and all the cosmetic accessories that complete a new house—yes, including the kitchen sink.

The other sections of this chapter (on RLE sprites, compiled sprites, and collision detection) are helpful, but might be considered the Italian tile of floors, whereas linoleum will work fine for most people. But the segment on animated sprites is absolutely crucial in your quest to master the subject of 2D game programming. So what is an animated sprite? You already learned a great deal about sprites in the last chapter, and you have at your disposal a good tool set for loading and blitting sprites (which are just based on common bitmaps). An *animated sprite*, then, is an array of sprites drawn using new properties, such as timing, direction, and velocity.

Here is a breakdown of the major topics in this chapter:

- Working with animated sprites
- Using run-length encoded sprites
- Working with compiled sprites
- Understanding collision detection

Animated Sprites

The sprites you have seen thus far were handled somewhat haphazardly, in that no real structure was available for keeping track of these sprites. They have simply been loaded using `load_bitmap` and then drawn using `draw_sprite`, with little else in the way of control or handling. To really be able to work with animated sprites in a highly complex game (such as a high-speed scrolling shooter like *R-Type* or *Mars Matrix*), you need a framework for drawing, erasing, and moving these sprites, and for detecting collisions. For all of its abstraction, Allegro leaves this entirely up to you—and for good reason. No single person can foresee the needs of another game programmer because every game has a unique set of requirements (more or less). Limiting another programmer (who may be far more talented than you) to using your concept of a sprite handler only encourages that person to ignore your handler and write his own. That is exactly why Allegro has no sprite handler; rather, it simply has a great set of low-level sprite routines, the likes of which you have already seen.

What should you do next, then? The real challenge is not designing a handler for working with animated sprites; rather, it is designing a game that will need these animated sprites, and then writing the code to fulfill the needs of the game. In this case, the game I am targeting for the sprite handler is *Tank War*, which you have improved in each new chapter—and this one will be no exception. In Chapter 8, you modified *Tank War* extensively to convert it from a vector- and bitmap-based game into a sprite-based game, losing some gameplay along the way. (The battlefield obstacles were removed.) At the end of this chapter, you'll add the sprite handler and collision detection—finally!

Drawing an Animated Sprite

To get started, you need a simple example followed by an explanation of how it works. I have written a quick little program that loads six images (of an animated cat) and draws them on the screen. The cat runs across the screen from left to right, using the sprite frames shown in Figure 9.1.

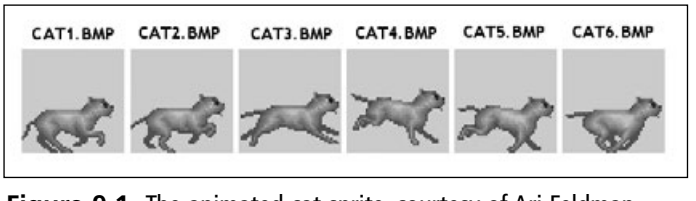


Figure 9.1 The animated cat sprite, courtesy of Ari Feldman

The *AnimSprite* program loads these six image files, each containing a single frame of the animated cat, and draws them in sequence, one frame after another, as the sprite is moved across the screen (see Figure 9.2).

Figure 9.2 The *AnimSprite* program shows how you can do basic sprite animation.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLACK makecol(0,0,0)

BITMAP *kitty[7];
char s[20];
int curframe=0, framedelay=5, framecount=0;
int x=100, y=200, n;

int main(void)
{
    //initialize the program
    allegro_init();
    install_keyboard();
    install_timer();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
```

```
    textout(screen, font, "AnimSprite Program (ESC to quit)",
        0, 0, WHITE);

    //load the animated sprite
    for (n=0; n<6; n++)
    {
        sprintf(s,"cat%d.bmp",n+1);
        kitty[n] = load_bitmap(s, NULL);
    }

    //main loop
    while(!key[KEY_ESC])
    {
        //erase the sprite
        rectfill(screen, x, y, x+kitty[0]->w, y+kitty[0]->h, BLACK);

        //update the position
        x += 5;
        if (x > SCREEN_W - kitty[0]->w)
            x = 0;

        //update the frame
        if (framecount++ > framedelay)
        {
            framecount = 0;
            curframe++;
            if (curframe > 5)
                curframe = 0;
        }

        acquire_screen();

        //draw the sprite
        draw_sprite(screen, kitty[curframe], x, y);

        //display logistics
        textprintf(screen, font, 0, 20, WHITE,
            "Sprite X,Y: %3d,%3d", x, y);
        textprintf(screen, font, 0, 40, WHITE,
            "Frame,Count,Delay: %2d,%2d,%2d",
            curframe, framecount, framedelay);

        release_screen();
```

```
        rest(10);
    }

    return 0;
}
END_OF_MAIN();
```

Now for that explanation, as promised. The difference between *AnimSprite* and *DrawSprite* (from the previous chapter) is multifaceted. The key variables, `curframe`, `framecount`, and `framedelay`, make realistic animation possible. You don't want to simply change the frame every time through the loop, or the animation will be too fast. The frame delay is a static value that really needs to be adjusted depending on the speed of your computer (at least until I cover timers in Chapter 11, "Timers, Interrupt Handlers, and Multi-Threading"). The frame counter, then, works with the frame delay to increment the current frame of the sprite. The actual movement of the sprite is a simple horizontal motion using the `x` variable.

```
//update the frame
if (framecount++ > framedelay)
{
    framecount = 0;
    curframe++;
    if (curframe > 5)
        curframe = 0;
}
```

A really well thought-out sprite handler will have variables for both the velocity (x, y) and velocity (x speed, y speed), along with a velocity delay to allow some sprites to move quite slowly compared to others. If there is no velocity delay, each sprite will move at least one pixel during each iteration of the game loop (unless velocity is zero, which means that sprite is motionless).

```
//update the position
x += 5;
if (x > SCREEN_W - kitty[0]->w)
    x = 0;
```

This concept is something I'll explain shortly.

Creating a Sprite Handler

Now that you have a basic—if a bit rushed—concept of sprite animation, I'd like to walk you through the creation of a sprite handler and a sample program with which to test it. Now you'll take the animation code from the last few pages and encapsulate it into a struct. If you were using the object-oriented C++ language instead of C, you'd no doubt

"class it." That's all well and good, but I don't care what C++ programmers claim—it's more difficult to understand, which is the key reason why this book focuses on C. That Allegro itself is written in C only supports this decision. The actual bitmap images for the sprite are stored separately from the sprite struct because it is more flexible that way.

In addition to those few animation variables seen in *AnimSprite*, a full-blown animated sprite handler needs to track several more variables. Here is the struct:

```
typedef struct SPRITE
{
      int x,y;
      int width,height;
      int xspeed,yspeed;
      int xdelay,ydelay;
      int xcount,ycount;
      int curframe,maxframe,animdir;
      int framecount,framedelay;
}SPRITE;
```

The variables inside a struct are called *struct elements*, so I will refer to them as such (see Figure 9.3).

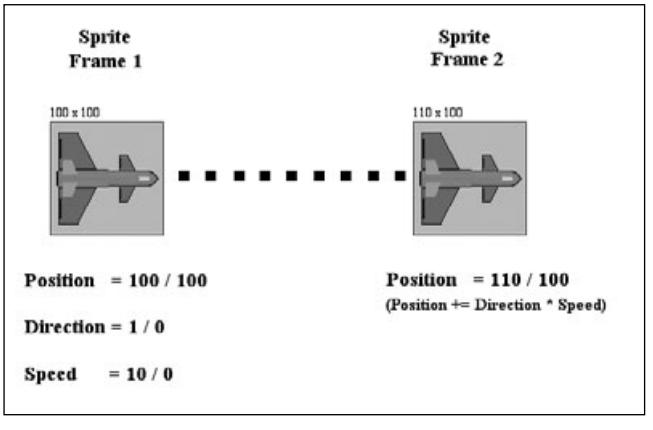


Figure 9.3 The `SPRITE` struct and its elements help abstract sprite movement into reusable code.

The first two elements (`x`, `y`) track the sprite's position. The next two (`width`, `height`) are set to the size of the sprite image (stored outside the struct). The velocity elements (`xspeed`, `yspeed`) determine how many pixels the sprite will move in conjunction with the velocity delay (`xdelay`, `xcount` and `ydelay`, `ycount`). The velocity delay allows some sprites to move much slower than other sprites on the screen—even more slowly than one pixel per frame. This gives you a far greater degree of control over how a sprite behaves. The animation elements (`curframe`, `maxframe`, `animdir`) help the sprite animation, and the animation delay elements (`framecount`, `framedelay`) help slow down the animation rate. The `animdir` element is of particular interest because it allows you to reverse the direction that the sprite frames are drawn (from 0 to `maxframe` or from `maxframe` to 0, with looping in either direction). The main reason why the `BITMAP` array containing the sprite images is not stored inside the struct is because that is wasteful—there might be many sprites sharing the same animation images.

Now that we have a sprite struct, the actual handler is contained in a function that I will call `updatesprite`:

```
void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}
```

As you can see, `updatesprite` accepts a pointer to a `SPRITE` variable. A pointer is necessary because elements of the struct are updated inside this function. This function would be called at every iteration through the game loop because the sprite elements should be closely tied to the game loop and timing of the game. The delay elements in particular should rely upon regular updates using a timed game loop. The animation section checks `animdir` to increment or decrement the `framecount` element.

However, `updatesprite` was not designed to affect sprite behavior, only to manage the logistics of sprite movement. After `updatesprite` has been called, you want to deal with that sprite's behavior within the game. For instance, if you are writing a space-based shooter featuring a spaceship and objects (such as asteroids) that the ship must shoot, then you might assign a simple warping behavior to the asteroids so that when they exit one side of the screen, they will appear at the opposite side. Or, in a more realistic game featuring a larger scrolling background, the asteroids might warp or bounce at the edges of the universe rather than just the screen. In that case, you would call `updatesprite` followed by another function that affects the behavior of all asteroids in the game and rely on custom or random values for each asteroid's struct elements to cause it to behave slightly differently than the other asteroids, but basically follow the same behavioral rules. Too many programmers ignore the concept of behavior and simply hard-code behaviors into a game.

I love the idea of constructing many behavior functions, and then using them in a game at random times to keep the player guessing what will happen next. For instance, a simple behavior that I often use in example programs is to have a sprite bounce off the edges of the screen. This could be abstracted into a bounce behavior if you go that one extra step in thinking and design it as a reusable function.

One thing that must be obvious when you are working with a real sprite handler is that it seems to have a lot of overhead, in that the struct elements must all be set at startup. There's no getting around that unless you want total chaos instead of a working game! You have to give all your sprites their starting values to make the game function as planned. Stuffing those variables into a struct helps to keep the game manageable when the source code starts to grow out of control (which frequently happens when you have a truly great game idea and you follow through with building it).

The SpriteHandler Program

I have written a program called *SpriteHandler* that demonstrates how to put all this together into a workable program that you can study. This program uses a ball sprite with 16 frames of animation, each stored in a file (ball1.bmp, ball2.bmp, and so on to ball16.bmp). One thing that you must do is learn how to store an animation sequence inside a single bitmap image and grab the frames out of it at run time. I'll show you how to do that shortly. Figure 9.4 shows the *SpriteHandler* program running. Each time the ball hits the edge, it changes direction and speed.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)
```

Figure 9.4 The *SpriteHandler* program demonstrates a full-featured animated sprite handler.

```
//define the sprite structure
typedef struct SPRITE
{
       int x,y;
       int width,height;
       int xspeed,yspeed;
       int xdelay,ydelay;
       int xcount,ycount;
       int curframe,maxframe,animdir;
       int framecount,framedelay;

}SPRITE;

//sprite variables
BITMAP *ballimg[16];
SPRITE theball;
SPRITE *ball = &theball;

//support variables
char s[20];
int n;
```

```
void erasesprite(BITMAP *dest, SPRITE *spr)
{
    //erase the sprite using BLACK color fill
    rectfill(dest, spr->x, spr->y, spr->x + spr->width,
        spr->y + spr->height, BLACK);
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void bouncesprite(SPRITE *spr)
{
```

```
    //simple screen bouncing behavior
    if (spr->x < 0)
    {
        spr->x = 0;
        spr->xspeed = rand() % 2 + 4;
        spr->animdir *= -1;
    }

    else if (spr->x > SCREEN_W - spr->width)
    {
        spr->x = SCREEN_W - spr->width;
        spr->xspeed = rand() % 2 - 6;
        spr->animdir *= -1;
    }

    if (spr->y < 40)
    {
        spr->y = 40;
        spr->yspeed = rand() % 2 + 4;
        spr->animdir *= -1;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
        spr->y = SCREEN_H - spr->height;
        spr->yspeed = rand() % 2 - 6;
        spr->animdir *= -1;
    }
}

void main(void)
{
    //initialize
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_keyboard();
    install_timer();
    srand(time(NULL));
    textout(screen, font, "SpriteHandler Program (ESC to quit)",
        0, 0, WHITE);
```

```
//load sprite images
for (n=0; n<16; n++)
{
    sprintf(s,"ball%d.bmp",n+1);
    ballimg[n] = load_bitmap(s, NULL);
}

//initialize the sprite with lots of randomness
ball->x = rand() % (SCREEN_W - ballimg[0]->w);
ball->y = rand() % (SCREEN_H - ballimg[0]->h);
ball->width = ballimg[0]->w;
ball->height = ballimg[0]->h;
ball->xdelay = rand() % 2 + 1;
ball->ydelay = rand() % 2 + 1;
ball->xcount = 0;
ball->ycount = 0;
ball->xspeed = rand() % 2 + 4;
ball->yspeed = rand() % 2 + 4;
ball->curframe = 0;
ball->maxframe = 15;
ball->framecount = 0;
ball->framedelay = rand() % 3 + 1;
ball->animdir = 1;

//game loop
while (!key[KEY_ESC])
{
    erasesprite(screen, ball);

    //perform standard position/frame update
    updatesprite(ball);

    //now do something with the sprite--a basic screen bouncer
    bouncesprite(ball);

    //lock the screen
    acquire_screen();

    //draw the ball sprite
    draw_sprite(screen, ballimg[ball->curframe], ball->x, ball->y);

    //display some logistics
    textprintf(screen, font, 0, 20, WHITE,
        "x,y,xspeed,yspeed: %2d,%2d,%2d,%2d",
```

```
            ball->x, ball->y, ball->xspeed, ball->yspeed);
        textprintf(screen, font, 0, 30, WHITE,
            "xcount,ycount,framecount,animdir: %2d,%2d,%2d,%2d",
            ball->xcount, ball->ycount, ball->framecount,
            ball->animdir);

        //unlock the screen
        release_screen();
        rest(10);
    }
    for (n=0; n<15; n++)
        destroy_bitmap(ballimg[n]);
    return;
}
END_OF_MAIN();
```

Grabbing Sprite Frames from an Image

In case you haven't yet noticed, the idea behind the sprite handler that you're building in this chapter is not to encapsulate Allegro's already excellent sprite functions (which were covered in the previous chapter). The temptation of nearly every C++ programmer would be to drool in anticipation over encapsulating Allegro into a series of classes. What a shame and what a waste of time! I can understand classing up an operating system service, which is vague and obscure, to make it easier to use. In my opinion, a class should be used to simplify very complex code, not to make simple code more complex just to satisfy an obsessive-compulsive need to do so.

On the contrary, you want to use the existing functionality of Allegro, not replace it with something else. By "something else" I mean not necessarily better, just different. The wrapping of one thing and turning it into another thing should arise out of use, not compulsion. Add new functions (or in the case of C++, new classes, properties, and methods) as you need them, not from some grandiose scheme of designing a library before using it.

Thus, you have a basic sprite handler and now you need a function to grab an animation sequence out of a tiled image. So you can get an idea of what I'm talking about, Figure 9.5 shows a 32-frame tiled animation sequence in a file called sphere.bmp.

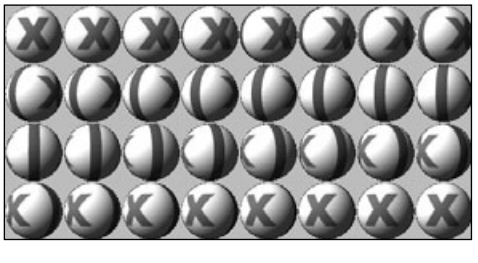


Figure 9.5 This bitmap image contains 32 frames of an animated sphere used as a sprite. Courtesy of Edgar Ibarra.

The frames would be easy to capture if they were lined up in a single row, so how would you grab them out of this file with eight columns and four rows? It's easy if you have the sprite tile algorithm. I'm sure someone described this in some mathematics or computer graphics book at one time or another in the past; I derived it on my own years ago. I suggest you print this simple algorithm in a giant font and paste it on the wall above your computer—or better yet, have a T-shirt made with it pasted across the front.

```
int x = startx + (frame % columns) * width;
int y = starty + (frame / columns) * height;
```

Using this algorithm, you can grab an animation sequence that is stored in a bitmap file, even if it contains more than one animation. (For instance, some simpler games might store all the images in a single bitmap file and grab each sprite at run time.) Now that you have the basic algorithm, here's a full function for grabbing a single frame out of an image by passing the width, height, column, and frame number:

```
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}
```

Note that `grabframe` doesn't destroy the `temp` bitmap after blitting the frame image to it. That is because the smaller `temp` bitmap is the return value for the function. It is up to the caller (usually `main`) to destroy the bitmap after it is no longer needed—or just before the game ends.

note

The `grabframe` function really should have some error detection code built in, such as a check for whether the bitmap is NULL after blitting it. As a matter of fact, all the code in this book is intentionally simplistic—with no error detection code—to make it easier to study. In an actual game, you would absolutely want to add checks in your code.

The SpriteGrabber Program

The *SpriteGrabber* program demonstrates how to use `grabframe` by modifying the *SpriteHandler* program and using a more impressive animated sprite that was rendered (courtesy of Edgar Ibarra). See Figure 9.6 for a glimpse of the program.

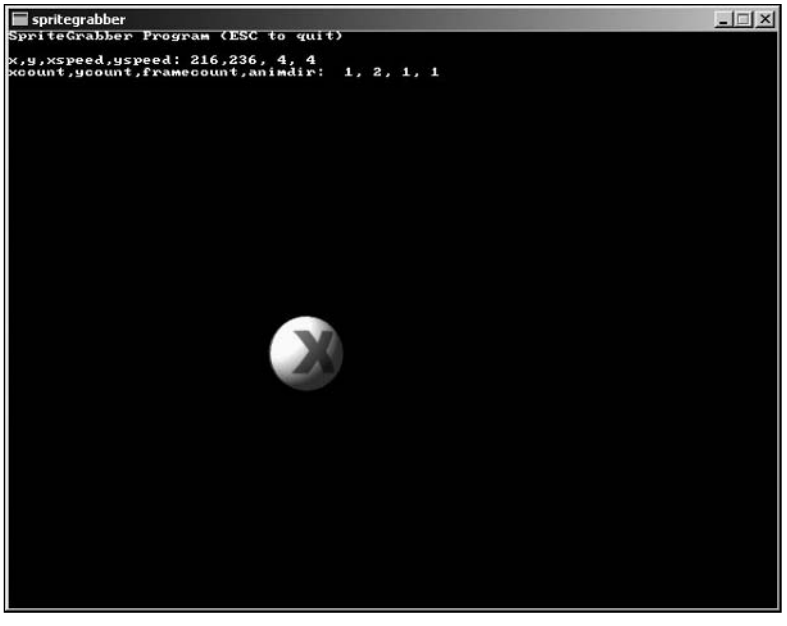


Figure 9.6 The *SpriteGrabber* program demonstrates how to grab sprite images (or animation frames) from a tiled source image.

I'm going to list the entire source code for *SpriteGrabber* and set in boldface the lines that have changed (or been added) so you can note the differences. I believe it would be too confusing to list only the changes to the program. There is a significant learning experience to be had by observing the changes or improvements to a program from one revision to the next.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)
```

```
//define the sprite structure
typedef struct SPRITE
{
       int x,y;
       int width,height;
       int xspeed,yspeed;
       int xdelay,ydelay;
       int xcount,ycount;
       int curframe,maxframe,animdir;
       int framecount,framedelay;

}SPRITE;

//sprite variables
BITMAP *ballimg[32];
SPRITE theball;
SPRITE *ball = &theball;

int n;

void erasesprite(BITMAP *dest, SPRITE *spr)
{
    //erase the sprite using BLACK color fill
    rectfill(dest, spr->x, spr->y, spr->x + spr->width,
        spr->y + spr->height, BLACK);
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }
```

```
    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void bouncesprite(SPRITE *spr)
{
    //simple screen bouncing behavior
    if (spr->x < 0)
    {
        spr->x = 0;
        spr->xspeed = rand() % 2 + 4;
        spr->animdir *= -1;
    }

    else if (spr->x > SCREEN_W - spr->width)
    {
        spr->x = SCREEN_W - spr->width;
        spr->xspeed = rand() % 2 - 6;
        spr->animdir *= -1;
    }

    if (spr->y < 40)
    {
        spr->y = 40;
        spr->yspeed = rand() % 2 + 4;
        spr->animdir *= -1;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
```

```
            spr->y = SCREEN_H - spr->height;
            spr->yspeed = rand() % 2 - 6;
            spr->animdir *= -1;
        }

    }

    BITMAP *grabframe(BITMAP *source,
                      int width, int height,
                      int startx, int starty,
                      int columns, int frame)
    {
        BITMAP *temp = create_bitmap(width,height);

        int x = startx + (frame % columns) * width;
        int y = starty + (frame / columns) * height;

        blit(source,temp,x,y,0,0,width,height);

        return temp;
    }

    void main(void)
    {
        BITMAP *temp;

        //initialize
        allegro_init();
        set_color_depth(16);
        set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
        install_keyboard();
        install_timer();
        srand(time(NULL));
        textout(screen, font, "SpriteGrabber Program (ESC to quit)",
            0, 0, WHITE);

        //load 32-frame tiled sprite image
        temp = load_bitmap("sphere.bmp", NULL);
        for (n=0; n<32; n++)
        {
            ballimg[n] = grabframe(temp,64,64,0,0,8,n);
        }
        destroy_bitmap(temp);
```

```
//initialize the sprite with lots of randomness
ball->x = rand() % (SCREEN_W - ballimg[0]->w);
ball->y = rand() % (SCREEN_H - ballimg[0]->h);
ball->width = ballimg[0]->w;
ball->height = ballimg[0]->h;
ball->xdelay = rand() % 2 + 1;
ball->ydelay = rand() % 2 + 1;
ball->xcount = 0;
ball->ycount = 0;
ball->xspeed = rand() % 2 + 4;
ball->yspeed = rand() % 2 + 4;
ball->curframe = 0;
ball->maxframe = 31;
ball->framecount = 0;
ball->framedelay = 1;
ball->animdir = 1;

//game loop
while (!key[KEY_ESC])
{
    erasesprite(screen, ball);

    //perform standard position/frame update
    updatesprite(ball);

    //now do something with the sprite--a basic screen bouncer
    bouncesprite(ball);

    //lock the screen
    acquire_screen();

    //draw the ball sprite
    draw_sprite(screen, ballimg[ball->curframe], ball->x, ball->y);

    //display some logistics
    textprintf(screen, font, 0, 20, WHITE,
        "x,y,xspeed,yspeed: %2d,%2d,%2d,%2d",
        ball->x, ball->y, ball->xspeed, ball->yspeed);
    textprintf(screen, font, 0, 30, WHITE,
        "xcount,ycount,framecount,animdir: %2d,%2d,%2d,%2d",
        ball->xcount, ball->ycount, ball->framecount, ball->animdir);

    //unlock the screen
```

```
        release_screen();

        rest(10);
    }

    for (n=0; n<31; n++)
        destroy_bitmap(ballimg[n]);

    return;
}
END_OF_MAIN();
```

The Next Step: Multiple Animated Sprites

You might think of a single sprite as a single-dimensional point in space (thinking in terms of geometry). An animated sprite containing multiple images for a single sprite is a two-dimensional entity. The next step, creating multiple copies of the sprite, might be compared to the third dimension. So far you have only dealt with and explored the concepts around a single sprite being drawn on the screen either with a static image or with an animation sequence. But how many games feature only a single sprite? It is really a test of the sprite handler to see how well it performs when it must contend with many sprites at the same time.

Because performance will be a huge issue with multiple sprites, I will use a double-buffer in the upcoming program for a nice, clean screen without flicker. I will add another level of complexity to make this even more interesting—dealing with a bitmapped background image instead of a blank background. `rectfill` will no longer suffice to erase the sprites during each refresh; instead, the background will have to be restored under the sprites as they move around.

Instead of a single sprite struct there is an array of sprite structs, and the code throughout the program has been modified to use the array. To initialize all of these sprites, you need to use a loop and make sure each pointer is pointing to each of the sprite structs.

```
//initialize the sprite
for (n=0; n<MAX; n++)
{
    sprites[n] = &thesprites[n];
    sprites[n]->x = rand() % (SCREEN_W - spriteimg[0]->w);
    sprites[n]->y = rand() % (SCREEN_H - spriteimg[0]->h);
    sprites[n]->width = spriteimg[0]->w;
    sprites[n]->height = spriteimg[0]->h;
    sprites[n]->xdelay = rand() % 3 + 1;
```

```
    sprites[n]->ydelay = rand() % 3 + 1;
    sprites[n]->xcount = 0;
    sprites[n]->ycount = 0;
    sprites[n]->xspeed = rand() % 8 - 5;
    sprites[n]->yspeed = rand() % 8 - 5;
    sprites[n]->curframe = rand() % 64;
    sprites[n]->maxframe = 63;
    sprites[n]->framecount = 0;
    sprites[n]->framedelay = rand() % 5 + 1;
    sprites[n]->animdir = rand() % 3 - 1;
}
```

This time I'm using a much larger animation sequence containing 64 frames, as shown in Figure 9.7. The source frames are laid out in an 8×8 grid of tiles.

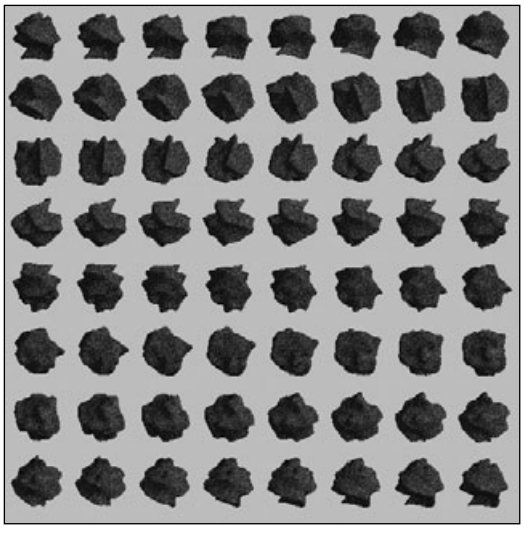


Figure 9.7 The source image for the animated asteroid contains 64 frames.

To load these frames into the sprite handler, a loop is used to grab each frame individually.

```
//load 64-frame tiled sprite image
temp = load_bitmap("asteroid.bmp", NULL);
for (n=0; n<64; n++)
{
    spriteimg[n] = grabframe(temp,64,64,0,0,8,n);
}
destroy_bitmap(temp);
```

The MultipleSprites Program

The *MultipleSprites* program animates 100 sprites on the screen, each of which has 64 frames of animation! Had this program tried to store the actual images with every single sprite instead of sharing the sprite images, it would have taken a huge amount of system memory to run—so now you see the wisdom in storing the images separately from the structs. Figure 9.8 shows the *MultipleSprites* program running at 1024×768. This program differs from *SpriteGrabber* because it uses a screen warp rather than a screen bounce behavior.

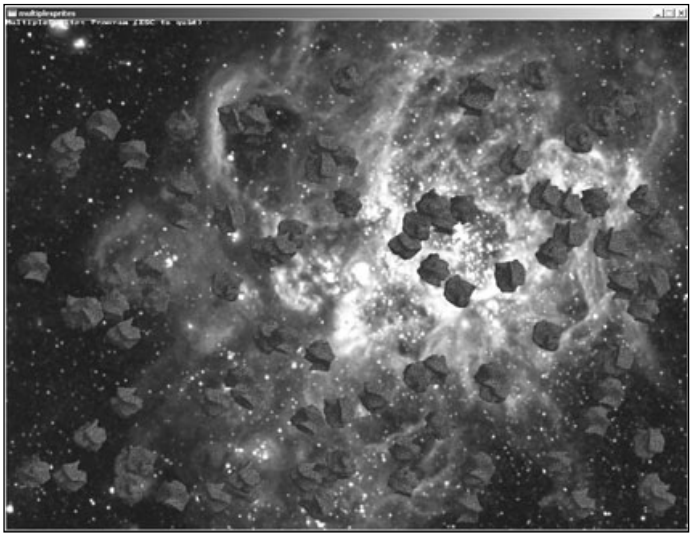


Figure 9.8 The *MultipleSprites* program animates 100 sprites on the screen.

This program uses a second buffer to improve performance. Could you imagine the speed hit after erasing and drawing 100 sprites directly on the screen? Even locking and unlocking the screen wouldn't help much with so many writes taking place on the screen. That is why this program uses double-buffering—so all blitting is done on the second buffer, which is then quickly blitted to the screen with a single function call.

```
//update the screen
acquire_screen();
blit(buffer,screen,0,0,0,0,buffer->w,buffer->h);
release_screen();
```

The game loop in *MultipleSprites* might look inefficient at first glance because there are four identical `for` loops for each operation—erasing, updating, warping, and drawing each of the sprites.

```
//erase the sprites
for (n=0; n<MAX; n++)
    erasesprite(buffer, sprites[n]);

//perform standard position/frame update
for (n=0; n<MAX; n++)
```

```
    updatesprite(sprites[n]);

//apply screen warping behavior
for (n=0; n<MAX; n++)
    warpsprite(sprites[n]);

//draw the sprites
for (n=0; n<MAX; n++)
    draw_sprite(buffer, spriteimg[sprites[n]->curframe],
        sprites[n]->x, sprites[n]->y);
```

It might seem more logical to use a single `for` loop with these functions inside that loop instead, right? Unfortunately, that is not the best way to handle sprites. First, all of the sprites must be erased before anything else happens. Second, all of the sprites must be moved before any are drawn or erased. Finally, all of the sprites must be drawn at the same time, or else artifacts will be left on the screen. Had I simply blasted the entire background onto the buffer to erase the sprites, this would have been a moot point. The program might even run faster than erasing 100 sprites individually. However, this is a learning experience. It's not always practical to clear the entire background, and this is just a demonstration—you won't likely have 100 sprites on the screen at once unless you are building a very complex scrolling arcade shooter or strategy game.

Following is the complete listing for the *MultipleSprites* program. If you are typing in the code directly from the book, you will want to grab the asteroids.bmp and ngc604.bmp files from the CD-ROM. (They are located in \chapter09\multiplesprites.)

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

#define MAX 100
#define WIDTH 640
#define HEIGHT 480
#define MODE GFX_AUTODETECT_WINDOWED

//define the sprite structure
typedef struct SPRITE
{
      int x,y;
      int width,height;
```

```
      int xspeed,yspeed;
      int xdelay,ydelay;
      int xcount,ycount;
      int curframe,maxframe,animdir;
      int framecount,framedelay;

}SPRITE;

//variables
BITMAP *spriteimg[64];
SPRITE thesprites[MAX];
SPRITE *sprites[MAX];
BITMAP *back;

void erasesprite(BITMAP *dest, SPRITE *spr)
{
    //erase the sprite
    blit(back, dest, spr->x, spr->y, spr->x, spr->y,
        spr->width, spr->height);
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
```

```
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void warpsprite(SPRITE *spr)
{
    //simple screen warping behavior
    if (spr->x < 0)
    {
        spr->x = SCREEN_W - spr->width;
    }

    else if (spr->x > SCREEN_W - spr->width)
    {
        spr->x = 0;
    }

    if (spr->y < 40)
    {
        spr->y = SCREEN_H - spr->height;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
        spr->y = 40;
    }

}

BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);
```

```
    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}

void main(void)
{
    BITMAP *temp, *buffer;
    int n;

    //initialize
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    install_keyboard();
    install_timer();
    srand(time(NULL));

    //create second buffer
    buffer = create_bitmap(SCREEN_W, SCREEN_H);

    //load & draw the background
    back = load_bitmap("ngc604.bmp", NULL);
    stretch_blit(back, buffer, 0, 0, back->w, back->h, 0, 0,
        SCREEN_W, SCREEN_H);

    //resize background to fit the variable-size screen
    destroy_bitmap(back);
    back = create_bitmap(SCREEN_W,SCREEN_H);
    blit(buffer,back,0,0,0,0,buffer->w,buffer->h);

    text_mode(-1);
    textout(buffer, font, "MultipleSprites Program (ESC to quit)",
        0, 0, WHITE);

    //load 64-frame tiled sprite image
    temp = load_bitmap("asteroid.bmp", NULL);
    for (n=0; n<64; n++)
    {
        spriteimg[n] = grabframe(temp,64,64,0,0,8,n);
```

```
}
destroy_bitmap(temp);


//initialize the sprite
for (n=0; n<MAX; n++)
{
     sprites[n] = &thesprites[n];
     sprites[n]->x = rand() % (SCREEN_W - spriteimg[0]->w);
     sprites[n]->y = rand() % (SCREEN_H - spriteimg[0]->h);
     sprites[n]->width = spriteimg[0]->w;
     sprites[n]->height = spriteimg[0]->h;
     sprites[n]->xdelay = rand() % 3 + 1;
     sprites[n]->ydelay = rand() % 3 + 1;
     sprites[n]->xcount = 0;
     sprites[n]->ycount = 0;
     sprites[n]->xspeed = rand() % 8 - 5;
     sprites[n]->yspeed = rand() % 8 - 5;
     sprites[n]->curframe = rand() % 64;
     sprites[n]->maxframe = 63;
     sprites[n]->framecount = 0;
     sprites[n]->framedelay = rand() % 5 + 1;
     sprites[n]->animdir = rand() % 3 - 1;
}

//game loop
while (!key[KEY_ESC])
{
     //erase the sprites
     for (n=0; n<MAX; n++)
          erasesprite(buffer, sprites[n]);

     //perform standard position/frame update
     for (n=0; n<MAX; n++)
          updatesprite(sprites[n]);

     //apply screen warping behavior
     for (n=0; n<MAX; n++)
          warpsprite(sprites[n]);

     //draw the sprites
     for (n=0; n<MAX; n++)
          draw_sprite(buffer, spriteimg[sprites[n]->curframe],
```

```
                sprites[n]->x, sprites[n]->y);

        //update the screen
        acquire_screen();
        blit(buffer,screen,0,0,0,0,buffer->w,buffer->h);
        release_screen();

        rest(10);
    }

    for (n=0; n<63; n++)
        destroy_bitmap(spriteimg[n]);

    return;
}
END_OF_MAIN();
```

I think that wraps up the material for animated sprites. You have more than enough information to completely enhance *Tank War* at this point. But hang on for a few more pages so I can go over some more important topics related to sprites.

Run-Length Encoded Sprites

Allegro provides a custom type of sprite that is compressed to save memory. Run-length encoded (RLE) sprite images can have significantly smaller memory footprints than standard bitmaps. In addition, there is some overhead in the header for each bitmap that also consumes memory. If you have an image that is not modified but only copied *from*, then it is a good candidate for RLE compression. (In this case they should be called RLE bitmaps instead of sprites because the image doesn't necessarily need to be small to be RLE compressed.)

There are several drawbacks to using RLE sprites, so some flexibility is sacrificed to save memory (and perhaps increase speed at the same time). RLE sprites can't be flipped, rotated, stretched, or copied *into*. All you can do is copy an RLE sprite to a destination bitmap using one of the custom RLE sprite-drawing functions.

RLE sprite images are stored in a simple run-length encoded format, where repeated zero pixels are replaced by a single length value and strings of normal pixels start with a counter that gives the length of the solid run. RLE sprites are usually much smaller than normal bitmaps because of the compression and because they avoid most of the overhead of the standard bitmap structure (which must support flipping, scaling, and so on). RLE sprites are often faster than normal bitmaps because rather than having to compare every single pixel with zero to determine whether it should be drawn, you can skip over a whole series of transparent pixels with a single instruction.

Creating and Destroying RLE Sprites

You can convert a normal memory bitmap (loaded with `load_bitmap` or created at run time) into an RLE sprite using the `get_rle_sprite` function.

```
RLE_SPRITE *get_rle_sprite(BITMAP *bitmap);
```

When you are using RLE sprites, you must be sure to destroy the sprites just as you destroy regular bitmaps. To destroy an RLE sprite, you will use a custom function created just for this purpose, called `destroy_rle_sprite`.

```
void destroy_rle_sprite(RLE_SPRITE *sprite);
```

Drawing RLE Sprites

Drawing an RLE sprite is very similar to drawing a normal sprite, and the parameters are similar in `draw_rle_sprite`.

```
void draw_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite, int x, int y);
```

Note that the only difference between `draw_rle_sprite` and `draw_sprite` is the second parameter, which refers directly to an `RLE_SPRITE` instead of a `BITMAP`. Otherwise, it is quite easy to convert an existing game to support RLE sprites.

Allegro provides two additional blitting functions for RLE sprites. The first one, `draw_trans_rle_sprite`, draws a sprite using translucent alpha-channel information and is comparable to `draw_trans_sprite` (only for RLE sprites, of course). This involves the use of blenders, as described in the previous chapter.

```
void draw_trans_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite,
    int x, int y);
```

Another variation of the RLE sprite blitter is `draw_lit_rle_sprite`, which uses lighting information to adjust a sprite's brightness when it is blitted to a destination bitmap. These functions are next to useless for any real game, so I am not planning to cover them here in any more detail. However, you can adapt the *TransSprite* program from the previous chapter with a little effort to use `draw_trans_rle_sprite`.

```
void draw_lit_rle_sprite(BITMAP *bmp, const RLE_SPRITE *sprite,
    int x, y, color);
```

The RLESprites Program

To assist with loading an image file into an RLE sprite, I have modified the `grabframe` function to return an `RLE_SPRITE` directly so conversion from a normal `BITMAP` is not necessary. As you can see from the short listing for this function, it creates a temporary `BITMAP` as a scratch buffer for the sprite frame, which is then converted to an RLE sprite, after which the scratch bitmap is destroyed and the `RLE_SPRITE` is returned by the function.

```
RLE_SPRITE *rle_grabframe(BITMAP *source,
                    int width, int height,
                    int startx, int starty,
                    int columns, int frame)
{
    RLE_SPRITE *sprite;
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);
    sprite = get_rle_sprite(temp);
    destroy_bitmap(temp);

    return sprite;
}
```

The *RLESprites* program is unique in that it is the first program to really start using background tiling—something that is covered in the next chapter. As you can see in Figure 9.9, a grass and stone tile is used to fill the bottom portion of the screen, while the dragon sprite flies over the ground. This is a little more realistic and interesting than the sprite being drawn to an otherwise barren, black background (although background scenery and a sky would help a lot).

Figure 9.9 The *RLESprites* program demonstrates how to use run-length encoded sprites to save memory and speed up sprite blitting.

The actual dragon sprite is comprised of six frames of animation, as shown in Figure 9.10. This sprite was created by Ari Feldman, as were the ground tiles.

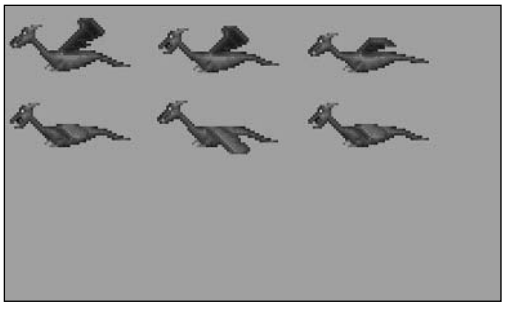


Figure 9.10 The animated dragon sprite used by the *RLESprites* program. Images courtesy of Ari Feldman.

Using the previous *SpriteGrabber* program as a basis, you should be able to adapt the code for the RLE sprite demo. I will highlight the differences in bold.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"

#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

//define the sprite structure
typedef struct SPRITE
{
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

//sprite variables
RLE_SPRITE *dragonimg[6];
SPRITE thedragon;
SPRITE *dragon = &thedragon;

void erasesprite(BITMAP *dest, SPRITE *spr)
{
```

```
    //erase the sprite using BLACK color fill
    rectfill(dest, spr->x, spr->y, spr->x + spr->width,
        spr->y + spr->height, BLACK);
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void warpsprite(SPRITE *spr)
{
    //simple screen warping behavior
    if (spr->x < 0)
    {
```

```
        spr->x = SCREEN_W - spr->width;
    }

    else if (spr->x > SCREEN_W - spr->width)
    {
        spr->x = 0;
    }

    if (spr->y < 40)
    {
        spr->y = SCREEN_H - spr->height;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
        spr->y = 40;
    }

}

RLE_SPRITE *rle_grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    RLE_SPRITE *sprite;
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);
    sprite = get_rle_sprite(temp);
    destroy_bitmap(temp);

    return sprite;
}

void main(void)
{
    BITMAP *temp;
    int n, x, y;
```

```
//initialize
allegro_init();
set_color_depth(16);
set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
install_keyboard();
install_timer();
srand(time(NULL));
textout(screen, font, "RLE Sprites Program (ESC to quit)",
    0, 0, WHITE);

//load and draw the blocks
temp = load_bitmap("block1.bmp", NULL);
for (y=0; y < SCREEN_H/2/temp->h+temp->h; y++)
    for (x=0; x < SCREEN_W/temp->w; x++)
        draw_sprite(screen, temp, x*temp->w, SCREEN_H/2+y*temp->h);
destroy_bitmap(temp);

temp = load_bitmap("block2.bmp", NULL);
for (x=0; x < SCREEN_W/temp->w; x++)
    draw_sprite(screen, temp, x*temp->w, SCREEN_H/2);
destroy_bitmap(temp);

//load rle sprite
temp = load_bitmap("dragon.bmp", NULL);
for (n=0; n<6; n++)
    dragonimg[n] = rle_grabframe(temp,128,64,0,0,3,n);
destroy_bitmap(temp);

//initialize the sprite
dragon->x = 500;
dragon->y = 150;
dragon->width = dragonimg[0]->w;
dragon->height = dragonimg[0]->h;
dragon->xdelay = 1;
dragon->ydelay = 0;
dragon->xcount = 0;
dragon->ycount = 0;
dragon->xspeed = -4;
dragon->yspeed = 0;
dragon->curframe = 0;
dragon->maxframe = 5;
dragon->framecount = 0;
dragon->framedelay = 10;
```

```
    dragon->animdir = 1;

    //game loop
    while (!key[KEY_ESC])
    {
        //erase the dragon
        erasesprite(screen, dragon);

        //move/animate the dragon
        updatesprite(dragon);
        warpsprite(dragon);

        //draw the dragon
        acquire_screen();
        draw_rle_sprite(screen, dragonimg[dragon->curframe],
            dragon->x, dragon->y);
        release_screen();

        rest(10);
    }

    for (n=0; n<6; n++)
        destroy_rle_sprite(dragonimg[n]);

    return;
}
END_OF_MAIN();
```

Compiled Sprites

RLE sprites are interesting because they are rendered with a custom function called `draw_rle_sprite` that actually decompresses the bitmap as it is being drawn to the destination bitmap (or screen). To truly speed up the blitting of these sprites, they would need to contain many repeated pixels. Therefore, a complex sprite with many colors and different pixels will not benefit at all from run-length encoding—don't always assume that just because an RLE sprite sounds cool, it is necessarily better than a regular sprite. Sometimes the good old-fashioned brute-force method works best.

However, if you are using sprites with many pixel runs of the same color in a row, then RLE sprites will draw faster. But isn't there yet another method that would draw them even faster? Given that you will choose the method to use for certain sprites while writing the code, it is up to you to decide whether a sprite contains long runs of pixels (good for packed blitting) or a diversity of pixels (good for brute-force blitting). Why not take RLE

sprites to the next step and actually pre-compile the sprite itself? After all, a blitter is nothing more than a function that copies a source bitmap to a destination bitmap one line at a time (often using fast assembly language copy instructions). How about coding those assembly language instructions directly into the sprites instead of storing pixels?

Intriguing idea? Personally, I love it, for no other reason than that it sounds cool! But what about performance? I'll leave that up to you to decide. Each game is different and each sprite is different, so it's largely up to you. Will standard, RLE, or compiled sprites work best with certain images but not with others? Suppose you are developing a role-playing game. These games typically have beautiful game worlds filled with plants, animals, houses, rivers, forests, and so on. An RLE or compiled sprite would just slow down this type of game compared to a standard sprite. But take a game like *Breakout* or *Tetris* that uses solid blocks for game pieces…now these blocks are absolutely perfect candidates for compressed or compiled sprites!

Using Compiled Sprites

What's the scoop with compiled sprites? They store the actual machine code instructions that draw a specific image onto a bitmap, using assembly language copy instructions with the colors of each pixel directly plugged into these instructions. Depending on the source image, a compiled sprite can render up to five times faster than a regular sprite using `draw_sprite`!

However, one of the drawbacks is that compiled sprites take up much more memory than either standard or RLE sprites, so you might not be able to use them for all the sprites in a game without causing the game to use up a lot of memory. By their very nature, compiled sprites are also quite constricted. Obviously, if you're talking about assembly instructions, a compiled sprite isn't really a bitmap any longer, but a miniature program that knows how to draw the image. Knowing this, one point is fairly evident, but I will enunciate it anyway: If you draw a compiled sprite outside the boundary of a bitmap (or the screen), bad things will happen because parts of program memory will be overwritten! The memory could contain anything—instructions, images, even the Allegro library itself. You must be very careful to keep track of compiled sprites so they are never drawn outside the edge of a bitmap or the screen, or the program will probably crash.

Now, how about another positive point? You can convert regular bitmaps into compiled sprites at run time, just like you could with RLE sprites. There is no need to convert your game artwork to any special format before use—you can do that when the program starts.

From this point, compiled sprites are functionally similar to RLE sprites. The first function you might recognize from the previous section—`get_compiled_sprite`. That's right, this function is almost exactly the same as `get_rle_sprite`, but it returns a pointer to a `COMPILED_SPRITE`.

```
COMPILED_SPRITE *get_compiled_sprite(BITMAP *bitmap, int planar);
```

The bitmap in the first parameter must be a memory bitmap (not a video bitmap or the screen). The second parameter is obsolete and should always be set to `FALSE`, specifying that the bitmap is a memory bitmap and not a multi-plane video mode (a holdover from a time when mode-x was popular with MS-DOS games).

In similar fashion, Allegro provides a custom function for destroying a compiled sprite in the `destroy_compiled_sprite` function.

```
void destroy_compiled_sprite(COMPILED_SPRITE *sprite);
```

What remains to be seen? Ah yes, the blitter. There is a single function for drawing a compiled sprite, and that concludes the discussion. (See, I told you compiled sprites were limited, if powerful.)

```
void draw_compiled_sprite(BITMAP *bmp, const COMPILED_SPRITE *sprite,
    int x, int y);
```

The first parameter is the destination bitmap, then comes the actual `COMPILED_SPRITE` to be blitted, followed by the `x` and `y` location for the sprite. Remember that `draw_compiled_sprite` does not do any clipping at the edges of the screen, so you could hose your program (and perhaps the entire operating system) if you aren't careful!

What if you are used to allowing sprites to go just beyond the boundaries of the screen so that they will warp to the other side more realistically? It certainly looks better than simply popping them to the other side when they near the edge (something that the *SpriteHandler* program did). There is a trick you can try if this will be a problem in your games. Create a memory bitmap (such as the second buffer) that is slightly larger than the actual screen, taking care to adjust the blitter when you are drawing it to the screen. Then you have some room with which to work when you are drawing sprites, so you won't have to be afraid that they will blow up your program.

Testing Compiled Sprites

To save some paper I've simply modified the *RLESprites* program for this section on compiled sprites; I will point out the differences between the programs. You can open the *RLESprites* program and make the few changes needed to test compiled sprites. Also, on the CD-ROM there is a complete *CompiledSprites* program that is already finished; you can load it up if you want. I liked the dragon so much that I've used it again in this program (giving credit again where it is due—thanks to Ari Feldman for the sprites).

Up near the top of the program where the variables are declared, there is a single line that changed from `RLE_SPRITE` to `COMPILED_SPRITE`.

```
//sprite variables
COMPILED_SPRITE *dragonimg[6];
```

Then skip down past `erasesprite`, `updatesprite`, and `warpsprite`, and you'll see the `rle_grabframe` function. I have converted it to `compiled_grabframe`, and it looks like the following. (The changes are in bold.)

```
COMPILED_SPRITE *compiled_grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    COMPILED_SPRITE *sprite;
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    //remember FALSE is always used in second parameter
    sprite = get_compiled_sprite(temp, FALSE);

    destroy_bitmap(temp);
    return sprite;
}
```

Moving along to the `main` function, you change the title.

```
textout(screen, font, "CompiledSprites Program (ESC to quit)",
    0, 0, WHITE);
```

Cruising further into the `main` function, make a change to the code that loads the dragon sprite images.

```
//load compiled sprite of dragon
temp = load_bitmap("dragon.bmp", NULL);
for (n=0; n<6; n++)
    dragonimg[n] = compiled_grabframe(temp,128,64,0,0,3,n);
destroy_bitmap(temp);
```

Down in the game loop where the dragon sprite is drawn to the screen, you need to change the code to use `draw_compiled_sprite`.

```
//draw the dragon
acquire_screen();
draw_compiled_sprite(screen, dragonimg[dragon->curframe],
    dragon->x, dragon->y);
release_screen();
```

There is one last change where the compiled sprite images are destroyed.

```
for (n=0; n<6; n++)
      destroy_compiled_sprite(dragonimg[n]);
```

That's it! Now try out the program and gain some experience with compiled sprites. You might not notice any speed improvement; then again, you might notice a huge improvement. It really depends on the source image, so experiment a little and make use of this new type of sprite whenever the need arises.

Collision Detection

Collision detection is the process of detecting when one sprite intersects (or collides with) another sprite. Although this is treated as a one-to-one interaction, the truth is that any one sprite can collide with many other sprites on the screen while a game is running. Collision detection is much easier once you have a basic sprite handler (which I have already gone over) because it is necessary to abstract the animation and movement so that any sprite can be accommodated (whether it's a space ship or an asteroid or an enemy ship—in other words, controlled versus behavioral sprites).

The easiest (and most efficient) way to detect when two sprites have collided is to compare their bounding rectangles. Figure 9.11 shows the bounding rectangles for two sprites, a jet airplane and a missile. As you can see in the figure, the missile will strike the plane when it contacts the wings, but it has not collided with the plane yet. However, a simple bounding-rectangle collision detection routine would mark this as a true collision because the bounding rectangle of the missile intersects with the bounding rectangle of the plane.

One way to increase the accuracy of bounding-rectangle collision detection is to make the source rectangle closely follow the boundaries of the actual sprite so there is less empty space around the sprite. For instance, suppose you were using a 64×64 image containing a bullet sprite that only uses 8×8 pixels in the center of the image—that's 56 pixels of empty space in each direction around the sprite that will foul up collision detection. There's no reason why you can't use different sizes for each sprite—make each one as small as possible to contain the image. The `load_bitmap` function certainly doesn't care how big the sprite is, and the blitting and collision routines don't care either. But you will speed up the game and make collision detection far more accurate by eliminating any unneeded empty space around a sprite. Keep them lean!

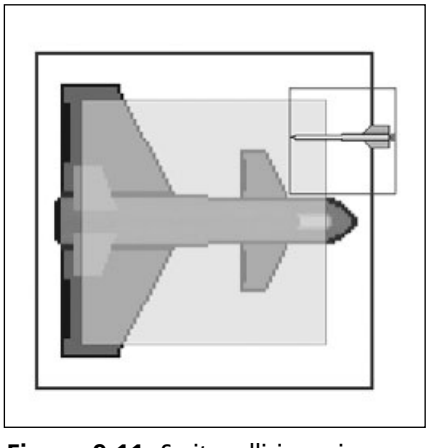


Figure 9.11 Sprite collision using bounding rectangles

Another way to increase collision detection accuracy is by reducing the virtual bounding rectangle used to determine collisions; that is, not by reducing the size of the image, but just the rectangular area used for collision detection. By reducing the bounding rectangle by some value, you can make the collisions behave in a manner that is more believable for the player. In the case of Figure 9.11 again, do you see the shaded rectangle inside the plane image? That represents a virtual bounding rectangle that is slightly smaller than the actual image. It might fail in some cases (look at the rear wings, which are outside the virtual rectangle), but in general this will improve the game. When sprites are quickly moving around the screen, small errors are not noticeable anyway.

Take a look at Figure 9.12, which shows three missiles (A, B, and C) intersecting with the jet airplane sprite. Right away you might notice a problem—the missiles have a lot of empty space. It would improve matters if the missile images were reduced to a smaller image containing only the missile's pixels, without all the blank space above and below the missile. Why? Missile A is clearly missing the plane sprite, but a "dumb" collision routine would see this is a collision using simple intersection. A smarter collision routine using a virtual rectangle would improve the situation, but the bounding rectangles for these missiles are so large that clear misses are likely to be treated as collisions more often than not.

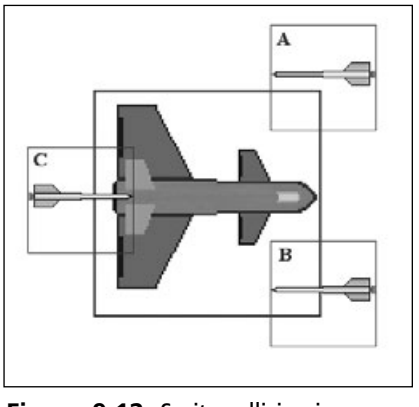


Figure 9.12 Sprite collision is more accurate using a virtual bounding rectangle with very little blank space.

Now take a look at Missile B in Figure 9.12. In this situation, the missile is intersecting the plane sprite's bounding rectangle, resulting in a true collision in most cases, but you can clearly see that it is not a collision. However, a virtual bounding rectangle would have compensated for the position of Missile B and recognized this as a miss. Missile C is clearly intersecting the plane's bounding rectangle, and the actual pixels in each image are intersecting so this is a definite collision. Any collision routine would have no problem with C, but it might have a problem with B and it would definitely have a problem with A. So you should design your collision routine to accommodate each situation and make sure your game's art resources are efficient and use the least amount of blank space necessary.

Following is a generic collision routine. This function accepts two `SPRITE` pointers as parameters, comparing their bounding rectangles for an intersection. A more useful collision routine might have included parameters for the virtual bounding rectangle compensators, but this function uses a hard-coded value of five pixels (`bx` and `by`), which you can modify as needed.

I have included the first glob of code only to simplify the collision code because so many variables are in use. This function works by comparing the four corners of the first sprite with the bounding rectangle of the second sprite, using a virtual rectangle that is five pixels smaller than the actual size of each sprite (which really should be passed as a parameter or calculated as a percentage of the size for the best results).

```
int collided(SPRITE *a, SPRITE *b)
{
    int wa = a->x + a->width;
    int ha =a->y + a->height;
    int wb = b->x + b->width;
    int hb = b->y + b->height;
    int bx = 5;
    int by = 5;

    if (inside(a->x, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(a->x, ha, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, ha, b->x+bx, b->y+by, wb-bx, hb-by))
        return 1;
    else
        return 0;
}
```

The second part of the function uses the shortcut variables to perform the collision detection based on the four corners of the first sprite. If any one of the points at each corner is inside the virtual bounding rectangle of the second sprite, then a collision has occurred and the result is returned to the calling routine.

The CollisionTest Program

I've made some changes to the *SpriteGrabber* program to demonstrate collision detection (rather than writing an entirely new program from scratch). Figure 9.13 shows the *CollisionTest* program in action. By changing a few lines and adding the collision routines, you can adapt *SpriteGrabber* and turn it into the *CollisionTest* program.

The first thing you need to add are some defines for the graphics mode and a define to specify the number of sprites used in the program. Note the additions in bold.

```
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include "allegro.h"
```

```
#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

#define NUM 10
#define WIDTH 640
#define HEIGHT 480
#define MODE GFX_AUTODETECT_FULLSCREEN
```

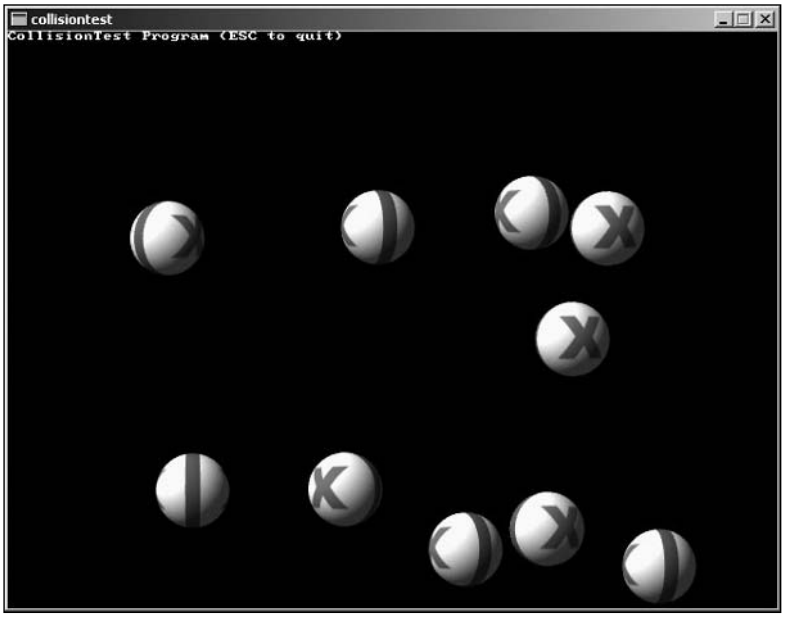


Figure 9.13 The *CollisionTest* program demonstrates how sprites can interact. Sprite image courtesy of Edgar Ibarra.

The next section of code declares the sprite variables below the `SPRITE` struct. All you need to do here is make these variables plural because this program uses many sprites instead of just the one sprite in the original *SpriteGrabber* program. The array of pointers will point to the struct array inside `main` because it is not possible to set the pointers in the declaration. (Each element of the array must be set individually.)

```
//sprite variables
BITMAP *ballimg[32];
SPRITE theballs[NUM];
SPRITE *balls[NUM];
```

After these minor changes, skip down a couple pages in the source code listing (ignoring the functions `erasesprite`, `updatesprite`, `bouncesprite`, and `grabframe`) and add the following functions after `grabframe`:

```
int inside(int x,int y,int left,int top,int right,int bottom)
{
    if (x > left && x < right && y > top && y < bottom)
        return 1;
    else
        return 0;
}

int collided(SPRITE *a, SPRITE *b)
{
    int wa = a->x + a->width;
    int ha =a->y + a->height;
    int wb = b->x + b->width;
    int hb = b->y + b->height;
    int bx = 5;
    int by = 5;

    if (inside(a->x, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(a->x, ha, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, ha, b->x+bx, b->y+by, wb-bx, hb-by))
        return 1;
    else
        return 0;
}

void checkcollisions(int num)
{
    int n,cx1,cy1,cx2,cy2;

    for (n=0; n<NUM; n++)
    {
        if (n != num && collided(balls[n], balls[num]))
        {
            //calculate center of primary sprite
            cx1 = balls[n]->x + balls[n]->width / 2;
            cy1 = balls[n]->y + balls[n]->height / 2;

            //calculate center of secondary sprite
            cx2 = balls[num]->x + balls[num]->width / 2;
            cy2 = balls[num]->y + balls[num]->height / 2;
```

```
                //figure out which way the sprites collided
                if (cx1 <= cx2)
                {
                    balls[n]->xspeed = -1 * rand() % 6 + 1;
                    balls[num]->xspeed = rand() % 6 + 1;
                    if (cy1 <= cy2)
                    {
                        balls[n]->yspeed = -1 * rand() % 6 + 1;
                        balls[num]->yspeed = rand() % 6 + 1;
                    }
                    else
                    {
                        balls[n]->yspeed = rand() % 6 + 1;
                        balls[num]->yspeed = -1 * rand() % 6 + 1;
                    }
                }
                else
                {
                    //cx1 is > cx2
                    balls[n]->xspeed = rand() % 6 + 1;
                    balls[num]->xspeed = -1 * rand() % 6 + 1;
                    if (cy1 <= cy2)
                    {
                        balls[n]->yspeed = rand() % 6 + 1;
                        balls[num]->yspeed = -1 * rand() % 6 + 1;
                    }
                    else
                    {
                        balls[n]->yspeed = -1 * rand() % 6 + 1;
                        balls[num]->yspeed = rand() % 6 + 1;
                    }
                }
            }
        }
    }
```

The `main` function has been modified extensively from the original version in *SpriteGrabber* to accommodate multiple sprites and calls to the collision functions, so I'll provide the complete `main` function here. This is similar to the previous version but now includes `for` loops to handle the multiple sprites on the screen, in addition to calling the collision routine.

```
void main(void)
{
    BITMAP *temp;
    BITMAP *buffer;
    int n;

    //initialize
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    install_keyboard();
    install_timer();
    srand(time(NULL));

    //create second buffer
    buffer = create_bitmap(SCREEN_W, SCREEN_H);

    text_mode(-1);
    textout(buffer, font, "CollisionTest Program (ESC to quit)",
        0, 0, WHITE);

    //load sprite images
    temp = load_bitmap("sphere.bmp", NULL);
    for (n=0; n<32; n++)
        ballimg[n] = grabframe(temp,64,64,0,0,8,n);
    destroy_bitmap(temp);

    //initialize the sprite
    for (n=0; n<NUM; n++)
    {
        balls[n] = &theballs[n];
        balls[n]->x = rand() % (SCREEN_W - ballimg[0]->w);
        balls[n]->y = rand() % (SCREEN_H - ballimg[0]->h);
        balls[n]->width = ballimg[0]->w;
        balls[n]->height = ballimg[0]->h;
        balls[n]->xdelay = 0;
        balls[n]->ydelay = 0;
        balls[n]->xcount = 0;
        balls[n]->ycount = 0;
        balls[n]->xspeed = rand() % 5 + 1;
        balls[n]->yspeed = rand() % 5 + 1;
        balls[n]->curframe = rand() % 32;
        balls[n]->maxframe = 31;
        balls[n]->framecount = 0;
        balls[n]->framedelay = 0;
```

```
        balls[n]->animdir = 1;
    }


    //game loop
    while (!key[KEY_ESC])
    {
        //erase the sprites
        for (n=0; n<NUM; n++)
            erasesprite(buffer, balls[n]);

        for (n=0; n<NUM; n++)
        {
            updatesprite(balls[n]);
            bouncesprite(balls[n]);
            checkcollisions(n);
        }

        //draw the sprites
        for (n=0; n<NUM; n++)
            draw_sprite(buffer, ballimg[balls[n]->curframe],
                balls[n]->x, balls[n]->y);

        //update the screen
        acquire_screen();
        blit(buffer,screen,0,0,0,0,buffer->w,buffer->h);
        release_screen();

        rest(10);
    }

    for (n=0; n<32; n++)
        destroy_bitmap(ballimg[n]);

    return;
}
END_OF_MAIN();
```

Enhancing Tank War

The next enhancement to *Tank War* will incorporate the new features you learned in this chapter, such as the use of a sprite handler and collision detection. For this modification,

you'll follow the same strategy used in previous chapters and only modify the latest version of the game, adding new features.

You need to add the `SPRITE` struct to the tankwar.h header file. But the struct needs two more variables before it will accommodate *Tank War* because the tanks and bullets included variables that are not yet part of the sprite handler. The `SPRITE` struct must also contain an int called `dir` and another called `alive`. Open the tankwar.h file and add the struct to this file just below the color definitions. After declaring the struct, you should also add the sprite arrays. At the same time, you no longer need the `tagTank` or `tagBullet` structs, so delete them! Also, you need to fill in a replacement for the "score" variables for each tank, so declare this as a new standalone `int` array.

```
//define the sprite structure
typedef struct SPRITE
{
    //new elements
    int dir, alive;

    //current elements
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

SPRITE mytanks[2];
SPRITE *tanks[2];
SPRITE mybullets[2];
SPRITE *bullets[2];

//replacement for the "score" variable in tank struct
int scores[2];
```

Replacing the two structs with the new `SPRITE` struct will have repercussions throughout the entire game source code because the new code uses pointers rather than struct variables directly. Therefore, you will need to modify most of the functions to use the `->` symbol in place of the period (`.`) to access elements of the struct when it is referenced with a pointer. The impact of converting the game to use sprite pointers won't be truly apparent until the next chapter, when you add a background to the game (finally!).

Now I want to go over the changes to the main source code file for *Tank War* with the changes in place.

```
//////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Chapter 9 - Tank War Game (Enhancement 4)
//////////////////////////////////////////////////////////////////////

#include "tankwar.h"

int inside(int x,int y,int left,int top,int right,int bottom)
{
    if (x > left && x < right && y > top && y < bottom)
        return 1;
    else
        return 0;
}

int collided(SPRITE *a, SPRITE *b)
{
    int wa = a->x + a->width;
    int ha =a->y + a->height;
    int wb = b->x + b->width;
    int hb = b->y + b->height;
    int bx = 5;
    int by = 5;

    if (inside(a->x, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(a->x, ha, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, a->y, b->x+bx, b->y+by, wb-bx, hb-by) ||
        inside(wa, ha, b->x+bx, b->y+by, wb-bx, hb-by))
        return 1;
    else
        return 0;
}

void drawtank(int num)
{
    int dir = tanks[num]->dir;
    int x = tanks[num]->x-15;
    int y = tanks[num]->y-15;
    draw_sprite(screen, tank_bmp[num][dir], x, y);
```

```
}

void erasetank(int num)
{
    int x = tanks[num]->x-17;
    int y = tanks[num]->y-17;
    rectfill(screen, x, y, x+33, y+33, BLACK);
}

void movetank(int num){
    int dir = tanks[num]->dir;
    int speed = tanks[num]->xspeed;

    //update tank position based on direction
    switch(dir)
    {
        case 0:
            tanks[num]->y -= speed;
            break;
        case 1:
            tanks[num]->x += speed;
            tanks[num]->y -= speed;
            break;
        case 2:
            tanks[num]->x += speed;
            break;
        case 3:
            tanks[num]->x += speed;
            tanks[num]->y += speed;
            break;
        case 4:
            tanks[num]->y += speed;
            break;
        case 5:
            tanks[num]->x -= speed;
            tanks[num]->y += speed;
            break;
        case 6:
            tanks[num]->x -= speed;
            break;
        case 7:
            tanks[num]->x -= speed;
            tanks[num]->y -= speed;
```

```
            break;
    }

    //keep tank inside the screen
    //use xspeed as a generic "speed" variable
    if (tanks[num]->x > SCREEN_W-22)
    {
        tanks[num]->x = SCREEN_W-22;
        tanks[num]->xspeed = 0;
    }
    if (tanks[num]->x < 22)
    {
        tanks[num]->x = 22;
        tanks[num]->xspeed = 0;
    }
    if (tanks[num]->y > SCREEN_H-22)
    {
        tanks[num]->y = SCREEN_H-22;
        tanks[num]->xspeed = 0;
    }
    if (tanks[num]->y < 22)
    {
        tanks[num]->y = 22;
        tanks[num]->xspeed = 0;
    }

    //see if tanks collided
/*    if (collided(tanks[0], tanks[1]))
    {
        textout(screen,font,"HIT",tanks[0]->x, tanks[0]->y,WHITE);
        tanks[0]->xspeed = 0;
        tanks[1]->xspeed = 0;
    }
    */
}

void explode(int num, int x, int y)
{
    int n;

    //load explode image
    if (explode_bmp == NULL)
    {
```

```
        explode_bmp = load_bitmap("explode.bmp", NULL);
    }

    //draw the explosion bitmap several times
    for (n = 0; n < 5; n++)
    {
        rotate_sprite(screen, explode_bmp,
            x + rand()%10 - 20, y + rand()%10 - 20,
            itofix(rand()%255));

        rest(30);
    }

    //clear the explosion
    circlefill(screen, x, y, 50, BLACK);

}

void updatebullet(int num)
{
    int x, y, tx, ty;
    int othertank;

    x = bullets[num]->x;
    y = bullets[num]->y;

    if (num == 1)
        othertank = 0;
    else
        othertank = 1;

    //is the bullet active?
    if (!bullets[num]->alive) return;

    //erase bullet
    rectfill(screen, x, y, x+10, y+10, BLACK);

    //move bullet
    bullets[num]->x += bullets[num]->xspeed;
    bullets[num]->y += bullets[num]->yspeed;
    x = bullets[num]->x;
    y = bullets[num]->y;
```

```
    //stay within the screen
    if (x < 6 || x > SCREEN_W-6 || y < 20 || y > SCREEN_H-6)
    {
        bullets[num]->alive = 0;
        return;
    }

    //look for a direct hit using basic collision
    tx = tanks[!num]->x;
    ty = tanks[!num]->y;
    //if (collided(bullets[num], tanks[!num]))
    if (inside(x,y,tx,ty,tx+16,ty+16))
    {
        //kill the bullet
        bullets[num]->alive = 0;

        //blow up the tank
        explode(num, x, y);
        score(num);
    }
    else
    //if no hit then draw the bullet
    {
        //draw bullet sprite
        draw_sprite(screen, bullet_bmp, x, y);

        //update the bullet positions (for debugging)
        textprintf(screen, font, SCREEN_W/2-50, 1, TAN,
            "B1 %-3dx%-3d  B2 %-3dx%-3d",
            bullets[0]->x, bullets[0]->y,
            bullets[1]->x, bullets[1]->y);
    }
}

void fireweapon(int num)
{
    int x = tanks[num]->x;
    int y = tanks[num]->y;

    //ready to fire again?
    if (!bullets[num]->alive)
    {
        bullets[num]->alive = 1;
```

```
//fire bullet in direction tank is facing
switch (tanks[num]->dir)
{
    //north
    case 0:
        bullets[num]->x = x-2;
        bullets[num]->y = y-22;
        bullets[num]->xspeed = 0;
        bullets[num]->yspeed = -BULLETSPEED;
        break;
    //NE
    case 1:
        bullets[num]->x = x+18;
        bullets[num]->y = y-18;
        bullets[num]->xspeed = BULLETSPEED;
        bullets[num]->yspeed = -BULLETSPEED;
        break;
    //east
    case 2:
        bullets[num]->x = x+22;
        bullets[num]->y = y-2;
        bullets[num]->xspeed = BULLETSPEED;
        bullets[num]->yspeed = 0;
        break;
    //SE
    case 3:
        bullets[num]->x = x+18;
        bullets[num]->y = y+18;
        bullets[num]->xspeed = BULLETSPEED;
        bullets[num]->yspeed = BULLETSPEED;
        break;
    //south
    case 4:
        bullets[num]->x = x-2;
        bullets[num]->y = y+22;
        bullets[num]->xspeed = 0;
        bullets[num]->yspeed = BULLETSPEED;
        break;
    //SW
    case 5:
        bullets[num]->x = x-18;
        bullets[num]->y = y+18;
        bullets[num]->xspeed = -BULLETSPEED;
```

```
                bullets[num]->yspeed = BULLETSPEED;
                break;
            //west
            case 6:
                bullets[num]->x = x-22;
                bullets[num]->y = y-2;
                bullets[num]->xspeed = -BULLETSPEED;
                bullets[num]->yspeed = 0;
                break;
            //NW
            case 7:
                bullets[num]->x = x-18;
                bullets[num]->y = y-18;
                bullets[num]->xspeed = -BULLETSPEED;
                bullets[num]->yspeed = -BULLETSPEED;
                break;
        }
    }
}

void forward(int num)
{
    //use xspeed as a generic "speed" variable
    tanks[num]->xspeed++;
    if (tanks[num]->xspeed > MAXSPEED)
        tanks[num]->xspeed = MAXSPEED;
}

void backward(int num)
{
    tanks[num]->xspeed--;
    if (tanks[num]->xspeed < -MAXSPEED)
        tanks[num]->xspeed = -MAXSPEED;
}

void turnleft(int num)
{
    tanks[num]->dir--;
    if (tanks[num]->dir < 0)
        tanks[num]->dir = 7;
}
```

```
void turnright(int num)
{
    tanks[num]->dir++;
    if (tanks[num]->dir > 7)
        tanks[num]->dir = 0;
}

void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])   gameover = 1;

    //WASD - SPACE keys control tank 1
    if (key[KEY_W])     forward(0);
    if (key[KEY_D])     turnright(0);
    if (key[KEY_A])     turnleft(0);
    if (key[KEY_S])     backward(0);
    if (key[KEY_SPACE]) fireweapon(0);

    //arrow - ENTER keys control tank 2
    if (key[KEY_UP])    forward(1);
    if (key[KEY_RIGHT]) turnright(1);
    if (key[KEY_DOWN])  backward(1);
    if (key[KEY_LEFT])  turnleft(1);
    if (key[KEY_ENTER]) fireweapon(1);

    //short delay after keypress
    rest(20);
}

void score(int player)
{
    //update score
    int points = ++scores[player];

    //display score
    textprintf(screen, font, SCREEN_W-70*(player+1), 1,
        BURST, "P%d: %d", player+1, points);
}

void setuptanks()
{
```

```
int n;

//configure player 1's tank
tanks[0] = &mytanks[0];
tanks[0]->x = 30;
tanks[0]->y = 40;
tanks[0]->xspeed = 0;
scores[0] = 0;
tanks[0]->dir = 3;

//load first tank bitmap
tank_bmp[0][0] = load_bitmap("tank1.bmp", NULL);

//rotate image to generate all 8 directions
for (n=1; n<8; n++)
{
    tank_bmp[0][n] = create_bitmap(32, 32);
    clear_bitmap(tank_bmp[0][n]);
    rotate_sprite(tank_bmp[0][n], tank_bmp[0][0],
        0, 0, itofix(n*32));
}

//configure player 2's tank
tanks[1] = &mytanks[1];
tanks[1]->x = SCREEN_W-30;
tanks[1]->y = SCREEN_H-30;
tanks[1]->xspeed = 0;
scores[1] = 0;
tanks[1]->dir = 7;

//load second tank bitmap
tank_bmp[1][0] = load_bitmap("tank2.bmp", NULL);

//rotate image to generate all 8 directions
for (n=1; n<8; n++)
{
    tank_bmp[1][n] = create_bitmap(32, 32);
    clear_bitmap(tank_bmp[1][n]);
    rotate_sprite(tank_bmp[1][n], tank_bmp[1][0],
        0, 0, itofix(n*32));
}
```

```
    //load bullet image
    if (bullet_bmp == NULL)
        bullet_bmp = load_bitmap("bullet.bmp", NULL);

    //initialize bullets
    for (n=0; n<2; n++)
    {
       bullets[n] = &mybullets[n];
       bullets[n]->x = 0;
       bullets[n]->y = 0;
       bullets[n]->width = bullet_bmp->w;
       bullets[n]->height = bullet_bmp->h;
    }
}

void setupscreen()
{
    int ret;

    //set video mode
    set_color_depth(32);
    ret = set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    //print title
    textprintf(screen, font, 1, 1, BURST,
        "Tank War - %dx%d", SCREEN_W, SCREEN_H);

    //draw screen border
    rect(screen, 0, 12, SCREEN_W-1, SCREEN_H-1, TAN);
    rect(screen, 1, 13, SCREEN_W-2, SCREEN_H-2, TAN);
}

void main(void)
{
    //initialize the game
    allegro_init();
    install_keyboard();
    install_timer();
```

```
    srand(time(NULL));
    setupscreen();
    setuptanks();

    //game loop
    while(!gameover)
    {
        //erase the tanks
        erasetank(0);
        erasetank(1);

        //move the tanks
        movetank(0);
        movetank(1);

        //draw the tanks
        drawtank(0);
        drawtank(1);

        //update the bullets
        updatebullet(0);
        updatebullet(1);

        //check for keypresses
        if (keypressed())
            getinput();

        //slow the game down
        rest(20);
    }

    //end program
    allegro_exit();
}
END_OF_MAIN();
```

Summary

This chapter was absolutely packed with advanced sprite code! You learned about animation, a subject that could take up an entire book of its own. (For instance, see Ari Feldman's book *Designing Arcade Computer Game Graphics*—http://www.arifeldman.com/reference.) Indeed, there is much to animation that I didn't get into in this chapter, but the most

important points were covered here and as a result, you have some great code that will be used in the rest of the book (especially that `grabframe` function) and perhaps many of your own Allegro game projects. You also learned about a couple subjects that are seldom discussed in game programming books—compiled and compressed sprite images. Using run-length encoded sprites, your game will use less memory, and by using compiled sprites, your game will run much faster. But possibly the most important subject in this chapter is the discussion of collision detection and how to implement it.

What comes next? We aren't done with sprites yet, not by a long shot! The next chapter delves into scrolling backgrounds. Get ready for some huge changes to *Tank War* because I've got some huge plans for the battlefield!

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Which function draws a standard sprite?
   A. `draw_standard_sprite`
   B. `standard_sprite`
   C. `draw_sprite`
   D. `blit_sprite`

2. What is a frame in the context of sprite animation?
   A. A single image in the animation sequence
   B. The bounding rectangle of a sprite
   C. The source image for the animation sequence
   D. A buffer image used to store temporary copies of the sprite

3. What is the purpose of a sprite handler?
   A. To provide a consistent way to animate and manipulate many sprites on the screen
   B. To prevent sprites from moving beyond the edges of the screen
   C. To provide a reusable sprite drawing function
   D. To keep track of the sprite position

4. What is a struct element?
   A. A property of a struct
   B. A sprite behavior
   C. The underlying Allegro sprite handler
   D. A variable in a structure

5. Which term describes a single frame of an animation sequence stored in an image file?
   A. Snapshot
   B. Tile
   C. Piece
   D. Take

6. Which Allegro function is used frequently to erase a sprite?
   A. `rectfill`
   B. `erase_sprite`
   C. `destroy_sprite`
   D. `blit`

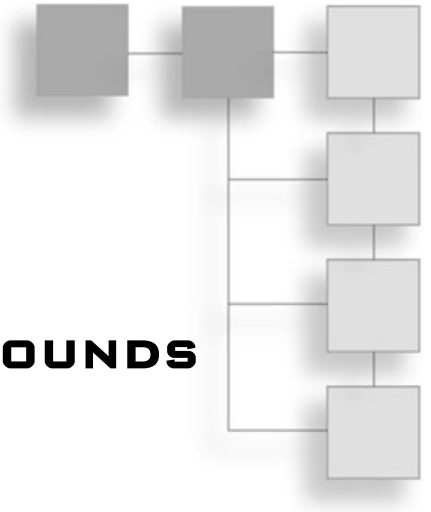
7. Which term describes a reusable activity for a sprite that is important in a game?
   A. Collision
   B. Animation
   C. Bounding
   D. Behavior

8. Which function converts a normal sprite into a run-length encoded sprite?
   A. `convert_sprite`
   B. `get_rle_sprite`
   C. `convert_to_rle`
   D. `load_rle_sprite`

9. Which function draws a compiled sprite to a destination bitmap?
   A. `draw_compiled`
   B. `draw_comp_sprite`
   C. `draw_compiled_sprite`
   D. `compiled_sprite`

10. What is the easiest (and most efficient) way to detect sprite collisions?
    A. Bounding rectangle intersection
    B. Pixel comparison
    C. Bilinear quadratic evaluation
    D. Union of two spheres

CHAPTER 10



Programming Tile-Based Backgrounds with Scrolling

Allegro has a history that goes way back to the 1980s, when it was originally developed for the Atari ST computer, which was a game programmer's dream machine (as were the Atari 800 that preceded it and the Commodore Amiga that was in a similar performance class). While IBM PC users were stuck playing text adventures and ASCII role-playing games (in which your player was represented by @ or P), Atari and Amiga programmers were playing with tile-based scrolling, hardware-accelerated sprites, and digital sound. If you revel in nostalgia as I do, I recommend you pick up *High Score! The Illustrated History of Electronic Games* by DeMaria and Wilson (McGraw-Hill Osborne Media, 2003). Given such roots, it is no surprise that Allegro has such terrific support for scrolling and sprites.

However, there is a drawback to the scrolling functionality—it is very platform dependent. Modern games simply don't use video memory for scrolling any longer. Back in the old days, it was a necessity because system memory was so limited. We take for granted a gigabyte of memory today, but that figure was as unbelievable in the 1980s as a manned trip to Mars is today. Allegro's scrolling functionality works with console-based operating systems such as MS-DOS and console Linux, where video memory is not a graphical handle provided by the operating system as it is today. Even so, the virtual screen buffers were very limited because they were designed for video cards with 256 to 1024 KB of video memory. You were lucky to have two 320×240 screens, let alone enough for a large scrolling world. Therefore, this chapter will focus on creating tile-based backgrounds with scrolling using secondary buffers. As you will discover, this is far easier than trying to wrangle memory out of a video card as programmers were forced to do years ago. A memory buffer will work well with either full-screen or windowed mode.

Here is a breakdown of the major topics in this chapter:

- Scrolling
- Working with tile-based backgrounds
- Enhancing *Tank War*

Introduction to Scrolling

What is scrolling? In today's gaming world, where 3D is the focus of everyone's attention, it's not surprising to find gamers and programmers who have never heard of scrolling. What a shame! The heritage of modern games is a long and fascinating one that is still relevant today, even if it is not understood or appreciated. The console industry puts great effort and value into scrolling, particularly on handheld systems, such as the Game Boy Advance. Given the extraordinary sales market for the GBA, would you be surprised to learn that more 2D games may be sold in a given day than 3D games? Oh, you're already sold on 2D games? Right; I digress. Figure 10.1 illustrates the concept of scrolling.

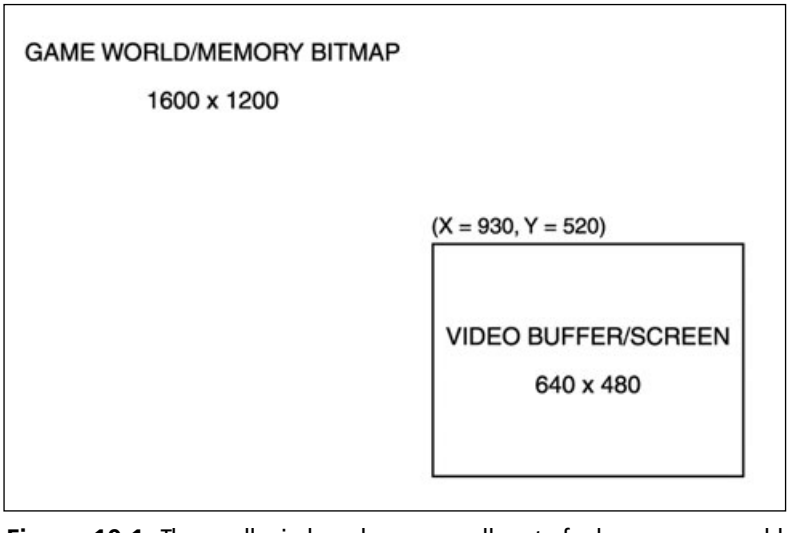


Figure 10.1 The scroll window shows a small part of a larger game world.

note

Scrolling is the process of displaying a small window of a larger virtual game world.

The key to scrolling is actually having something in the virtual game world to display in the scroll window. Also, I should point out that the entire screen need not be used as the

scroll window. It is common to use the entire screen in scrolling-shooter games, but role-playing games often use a smaller window on the screen for scrolling, using the rest of the screen for gameplay (combat, inventory, and so on) and player/party information (see Figure 10.2).

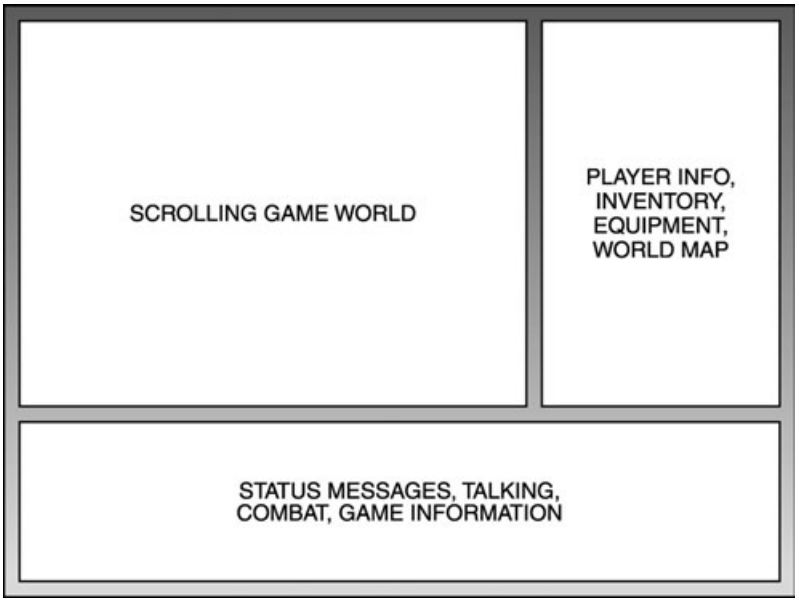


Figure 10.2 Some games use a smaller scroll window on the game screen.

You could display one huge bitmap image in the virtual game world representing the current level of the game, and then copy (blit) a portion of that virtual world onto the screen. This is the simplest form of scrolling. Another method uses tiles to create the game world, which I'll cover shortly. First, you'll write a short program to demonstrate how to use bitmap scrolling.

A Limited View of the World

I have written a program called *ScrollScreen* that I will show you. The \chapter10\ScrollScreen folder on the CD-ROM contains the bigbg.bmp file used in this program. Although I encourage you to write the program yourself, feel free to load the project in either KDevelop, Dev-C++, or Visual C++. Figure 10.3 shows the bigbg.bmp file.

When you run the program, the program will load the bigbg.bmp image into the virtual buffer and display the upper-left corner in the 640×480 screen. (You can change the resolution if you want, and I also encourage you to try running the program in full-screen mode using `GFX_AUTODETECT_FULLSCREEN` for the best effect.) The program detects when the

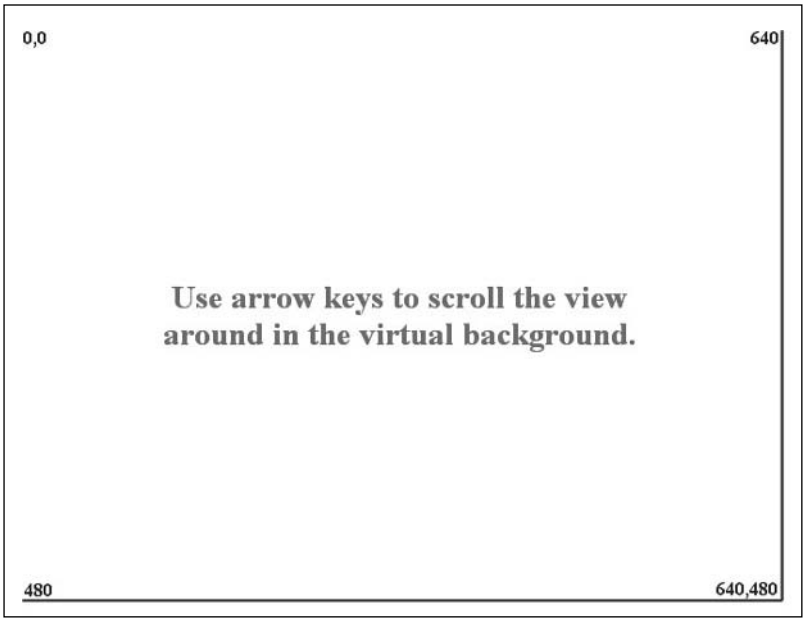


Figure 10.3 The bigbg.bmp file is loaded into the virtual memory buffer for scrolling.

arrow keys have been pressed and adjusts the x and y variables accordingly. Displaying the correct view is then a simple matter of blitting with the x and y variables (see Figure 10.4).

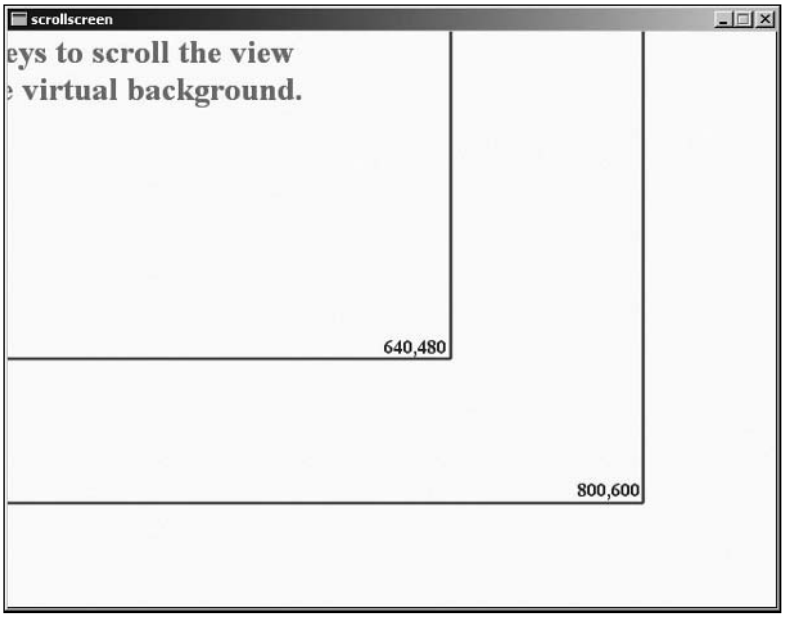


Figure 10.4 The *ScrollScreen* program demonstrates how to perform virtual buffer scrolling.

note

You could just as easily create a large virtual memory bitmap at run time and draw on that bitmap using the Allegro drawing functions you have learned thus far. I have chosen to create the bitmap image beforehand and load it into the program to keep the code listing shorter. Either method works the same way.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

//define some convenient constants
#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define STEP 8

//virtual buffer variable
BITMAP *scroll;

//position variables
int x=0, y=0;

//main function
void main(void)
{
    //initialize allegro
    allegro_init();
    install_keyboard();
    install_timer();
    set_color_depth(16);
    if (set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0) != 0)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(allegro_error);
        return;
    }

    //load the large bitmap image from disk
    scroll = load_bitmap("bigbg.bmp", NULL);
    if (scroll == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
```

```
        allegro_message("Error loading bigbg.bmp file");
        return;
    }

    //main loop
    while (!key[KEY_ESC])
    {
        //check right arrow
        if (key[KEY_RIGHT])
        {
            x += STEP;
            if (x > scroll->w - WIDTH)
                x = scroll->w - WIDTH;
        }

        //check left arrow
        if (key[KEY_LEFT])
        {
            x -= STEP;
            if (x < 0)
                x = 0;
        }

        //check down arrow
        if (key[KEY_DOWN])
        {
            y += STEP;
            if (y > scroll->h - HEIGHT)
                y = scroll->h - HEIGHT;
        }

        //check up arrow
        if (key[KEY_UP])
        {
            y -= STEP;
            if (y < 0)
                y = 0;
        }

        //draw the scroll window portion of the virtual buffer
        blit(scroll, screen, x, y, 0, 0, WIDTH-1, HEIGHT-1);
```

```
        //slow it down
        rest(20);
    }
    destroy_bitmap(scroll);
    return;
}
END_OF_MAIN();
```

The first thing I would do to enhance this program is create two variables, `lastx` and `lasty`, and set them to equal `x` and `y`, respectively, at the end of the main loop. Then, before blitting the window, check to see whether `x` or `y` has changed since the last frame and skip the `blit` function. There is no need to keep blitting the same portion of the virtual background if it hasn't moved.

> If you have gotten the *ScrollScreen* program to work, then you have taken the first step to creating a scrolling arcade-style game (or one of the hundred-thousand or so games released in the past 20 years). In the old days, getting the scroller working was usually the first step to creating a sports game. In fact, that was my first assignment at Semi-Logic Entertainments back in 1994, during the prototype phase of *Wayne Gretzky and the NHLPA All-Stars*—to get a hockey rink to scroll as fast as possible.
>
> Back then, I was using Borland C++ 4.5, and it just wasn't fast enough. First of all, this was a 16-bit compiler, while the 80×86- and Pentium-class PCs of the day were capable of 32-bit memory copies (`mov` instruction) that could effectively draw four pixels at a time in 8-bit color mode or two pixels at a time in 16-bit mode. Fortunately, Allegro already uses high-speed assembly instructions for blitting, as the low-level functions are optimized for each operating system using assembly language.

Introduction to Tile-Based Backgrounds

You have seen what a simple scroller looks like, even though it relied on keyboard input to scroll. A high-speed scrolling arcade game would automatically scroll horizontally or vertically, displaying a ground-, air-, or space-based terrain below the player (usually represented by an airplane or a spaceship). The point of these games is to keep the action moving so fast that the player doesn't have a chance to rest from one wave of enemies to the next. Two upcoming chapters have been dedicated to these very subjects! For the time being, I want to keep things simple to cover the basics of scrolling before you delve into these advanced chapters.

tip

For an in-depth look at vertical scrolling, see Chapter 13, "Vertical Scrolling Arcade Games." If you prefer to go horizontal, you can look forward to Chapter 14, "Horizontal Scrolling Platform Games."

Backgrounds and Scenery

A background is comprised of imagery or terrain in one form or another, upon which the sprites are drawn. The background might be nothing more than a pretty picture behind the action in a game, or it might take an active part, as in a scroller. When you are talking about scrollers, they need not be relegated only to the high-speed arcade games. Role-playing games are usually scrollers too, as are most sports games.

You should design the background around the goals of your game, not the other way around. You should not come up with some cool background and then try to build the game around it. (However, I admit that this is often how games are started.) You never want to rely on a single cool technology as the basis for an entire game, or the game will be forever remembered as a trendy game that tried to cash in on the latest fad. Instead of following and imitating, set your own precedents and make your own standards!

What am I talking about, you might ask? You might have the impression that anything and everything that could possibly have been done with a scrolling game has already been done ten times over. Not true. Not true! Remember when *Doom* first came out? Everyone had been imitating *Wolfenstein 3-D* when Carmack and Romero bumped up the notch a few hundred points and raised everyone's expectations so high that shockwaves reverberated throughout the entire game industry—console and PC alike.

Do you really think it has all been done before and there is no more room for innovation, that the game industry is saturated and it's impossible to make a successful "indie" game? That didn't stop Bungie from going for broke on their first game project. *Halo* has made its mark in gaming history by upping everyone's expectations for superior physics and intelligent opponents. Now, a few years hence, what kinds of games are coming out? What is the biggest industry buzzword? *Physics*. Design a game today without it, and suddenly your game is *so 1990s* in the gaming press. It's all about physics and AI now, and that started with *Halo*. Rather, it was perfected with *Halo*—I can't personally recall a game with that level of interaction before *Halo* came along. There is absolutely no reason why you can't invent the next innovation or revolution in gaming, even in a 2D game.

tip

Eh…all this philosophizing is giving me a headache. Time for some Strong Bad. Check out http://www.homestarrunner.com/sbemail94.html for one of my favorites. Okay, back to business.

Creating Backgrounds from Tiles

The real power of a scrolling background comes from a technique called tiling. *Tiling* is a process in which there really is no background, just an array of tiles that make up the background as it is displayed. In other words, it is a virtual virtual background and it takes up very little memory compared to a full bitmapped background (such as the one in *ScrollScreen*). Take a look at Figure 10.5 for an example.

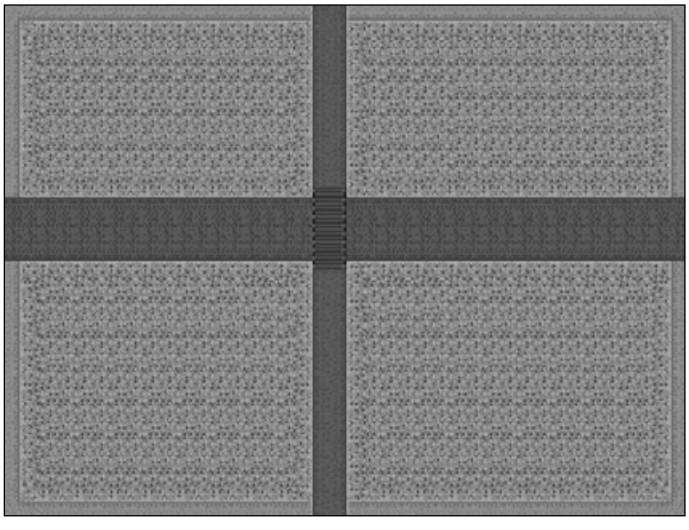


Figure 10.5 A bitmap image constructed of tiles

Can you count the number of tiles used to construct the background in Figure 10.5? Eighteen tiles make up this image, actually. Imagine that—an entire game screen built using a handful of tiles, and the result is pretty good! Obviously a real game would have more than just grass, roads, rivers, and bridges; a real game would have sprites moving on top of the background. How about an example? I thought you'd like that idea.

Tile-Based Scrolling

The *TileScroll* program uses tiles to fill the large background bitmap when the program starts. Other than that initial change, the program functions exactly like the *ScrollScreen* program. Take a look at Figure 10.6.

You might wonder why the screen looks like such a mess. That was intentional, not a mistake. The tiles are drawn to the background randomly, so they're all jumbled incoherently—which is, after all, the nature of randomness. After this, I'll show you how to place the tiles in an actual order that makes sense. Also, you can look forward to an entire chapter dedicated to this subject in Chapter 12, "Creating a Game World: Editing Tiles and Levels."

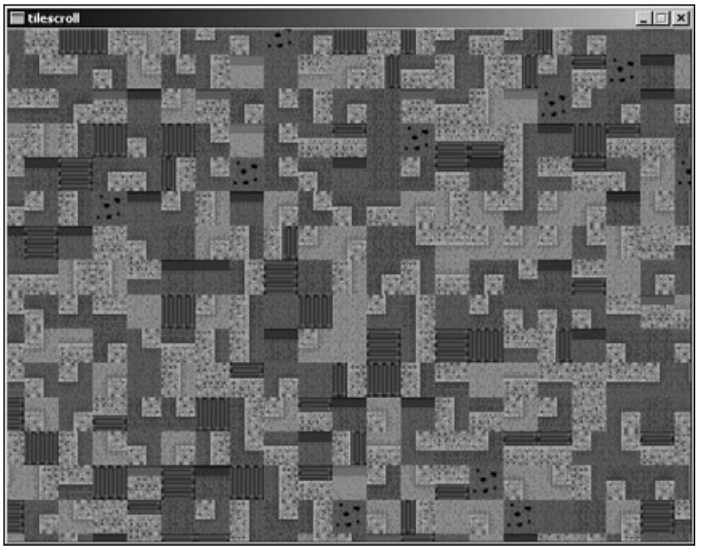


Figure 10.6 The *TileScroll* program demonstrates how to perform tile-based background scrolling.

Why an entire chapter just for this subject? Because it's huge! You're just getting into the basics here, but Chapter 12 will explore map editors, creating game worlds, and other higher-level concepts. The actual bitmap containing the tiles is shown in Figure 10.7.

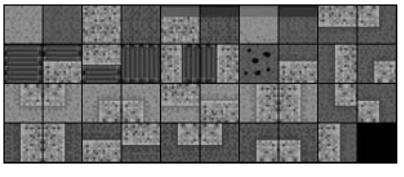


Figure 10.7 The source file containing the tiles used in the *TileScroll* program

Here's the source code for the *TileScroll* program:

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

//define some convenient constants
#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480

#define STEP 8
#define TILEW 32
#define TILEH 32
#define TILES 39
#define COLS 10

//temp bitmap
BITMAP *tiles;

//virtual background buffer
BITMAP *scroll;

//position variables
int x=0, y=0, n;
```

```
int tilex, tiley;

//reuse our friendly tile grabber from chapter 9
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}

//main function
void main(void)
{
    //initialize allegro
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    set_color_depth(16);

    //set video mode
    if (set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0) != 0)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(allegro_error);
        return;
    }

    //create the virtual background
    scroll = create_bitmap(1600, 1200);
    if (scroll == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating virtual background");
        return;
```

```
}

//load the tile bitmap
tiles = load_bitmap("tiles.bmp", NULL);
if (tiles == NULL)
{
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Error loading tiles.bmp file");
    return;
}

//now draw tiles randomly on virtual background
for (tiley=0; tiley < scroll->h; tiley+=TILEH)
{
    for (tilex=0; tilex < scroll->w; tilex+=TILEW)
    {
        //pick a random tile
        n = rand() % TILES;
        //use the result of grabframe directly in blitter
        blit(grabframe(tiles, TILEW+1, TILEH+1, 0, 0, COLS, n),
            scroll, 0, 0, tilex, tiley, TILEW, TILEH);
    }
}

//main loop
while (!key[KEY_ESC])
{
    //check right arrow
    if (key[KEY_RIGHT])
    {
        x += STEP;
        if (x > scroll->w - WIDTH)
            x = scroll->w - WIDTH;
    }

    //check left arrow
    if (key[KEY_LEFT])
    {
        x -= STEP;
        if (x < 0)
            x = 0;
    }
```

```
        //check down arrow
        if (key[KEY_DOWN])
        {
            y += STEP;
            if (y > scroll->h - HEIGHT)
                y = scroll->h - HEIGHT;
        }

        //check up arrow
        if (key[KEY_UP])
        {
            y -= STEP;
            if (y < 0)
                y = 0;
        }

        //draw the scroll window portion of the virtual buffer
        blit(scroll, screen, x, y, 0, 0, WIDTH-1, HEIGHT-1);

        //slow it down
        rest(20);
    }

    destroy_bitmap(scroll);
    destroy_bitmap(tiles);
    return;
}
END_OF_MAIN();
```

Creating a Tile Map

Displaying random tiles just to make a proof-of-concept is one thing, but it is not very useful. True, you have some code to create a virtual background, load tiles onto it, and then scroll the game world. What you really need won't be covered until Chapter 12, so as a compromise, you can create game levels using an array to represent the game world. In the past, I have generated a realistic-looking game map with source code, using an algorithm that matched terrain curves and straights (such as the road, bridge, and river) so that I created an awesome map from scratch, all by myself. The result, I'm sure you'll agree, is one of the best maps ever made. Some errors in the tile matching occurred, though, and a random map doesn't have much point in general. I mean, building a random landscape is one thing, but constructing it at run time is not a great solution—even if your map-generating routine is very good. For instance, many games, such as *Warcraft III,*

Age of Mythology, and *Civilization III*, can generate the game world on the fly. Obviously, the programmers spent a lot of time perfecting the world-generating routines. If your game would benefit by featuring a randomly generated game world, then your work is cut out for you but the results will be worth it. This is simply one of those design considerations that you must make, given that you have time to develop it.

Assuming you don't have the means to generate a random map at this time, you can simply create one within an array. Then you can modify the *TileScroll* program so it uses the array. Where do you start? First of all, you should realize that the tiles are numbered and should be referenced this way in the map array.

Here is what the array looks like, as defined in the *GameWorld* program:

```
int map[MAPW*MAPH] = {
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};
```

It's not complicated—simply a bunch of twos (grass) bordered by zeroes (stone). The trick here is that this is really only a single-dimensional array, but the listing makes it obvious how the map will look because there are 25 numbers in each row—the same number

of tiles in each row. I did this intentionally so you can use this as a template for creating your own maps. And you can create more than one map if you want. Simply change the name of each map and reference the map you want in the `blit` function so that your new map will show up. You are not limited in adding more tiles to each row. One interesting thing you can try is making `map` a two-dimensional array containing many maps, and then changing the map at run time! How about looking for the keys 1–9 (`KEY_1`, `KEY_2`,…`KEY_9`), and then changing the map number to correspond to the key that was pressed? It would be interesting to see the map change right before your eyes without re-running the program (sort of like warping). Now are you starting to see the potential? You could use this simple scrolling code as the basis for any of a hundred different games if you have the creative gumption to do so.

I have prepared a legend of the tiles and the value for each in Figure 10.8. You can use the legend while building your own maps.

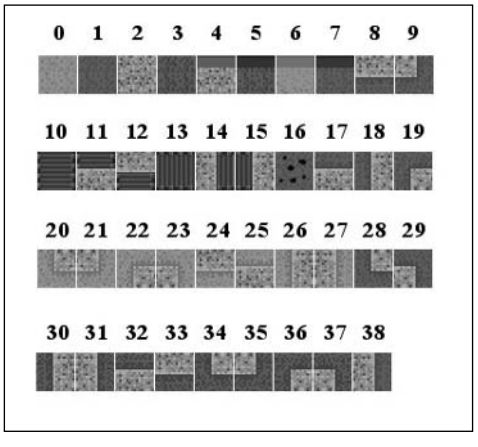


Figure 10.8 A legend of the tiles and their reference numbers used to create a map in the *GameWorld* program

note

All of the tiles used in this chapter were created by Ari Feldman, and I also owe him a debt of gratitude for creating most of the artwork used in this book. If you would like to contact Ari to ask him about custom artwork for your own game, you can reach him at http://www.arifeldman.com.

Call the new program *GameWorld*. This new demo will be similar to *TileScroll*, but it will use a map array instead of placing the tiles randomly. This program will also use a smaller virtual background to cut down on the size of the map array. Why? Not to save memory, but to make the program more manageable. Because the virtual background was 1600×1200 in the previous program, it would require 50 columns of tiles across and 37 rows of tiles down to fill it! That is no problem at all for a map editor program, but it's too much data to type in manually. To make it more manageable, the new virtual background will be 800 pixels across. I know, I know—that's not much bigger than the 640×480 screen. The point is to demonstrate how it will work, not to build a game engine, so don't worry about it. If you want to type in the values to create a bigger map, by all means, go for it! That would be a great learning experience, as a matter of fact. For your purposes here (and with my primary goal of being able to print an entire row of numbers in a single source code line in the book), I'll stick to 25 tiles across and 25 tiles down. You can work with a map that is deeper than it is wide, which will allow you to test scrolling up and down fairly well. Figure 10.9 shows the output from the *GameWorld* program.

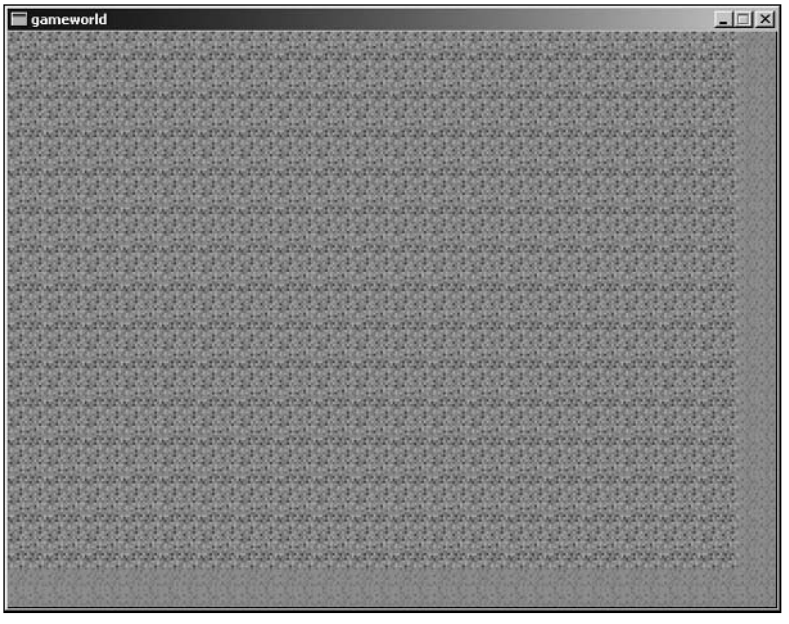


Figure 10.9 The *GameWorld* program scrolls a map that was defined in the `map` array.

How about that source code? Let's just add a few lines to the *TileScroll* program to come up with this new version. I recommend creating a new project called *GameWorld*, setting up the linker options for Allegro's library file, and then pasting the source code from *TileScroll* into the new main.c file in the *GameWorld* program. If you don't feel like doing all that, fine; go ahead and mess up the *TileScroll* program!

First, up near the top with the other defines, add these lines:

```
#define MAP_ACROSS 25
#define MAP_DOWN 25
#define MAPW MAP_ACROSS * TILEW
#define MAPH MAP_DOWN * TILEH
```

Then, of course, add the `map` array definition below the defines. (Refer back a few pages for the listing.) Only one more change and you're finished. You need to make a slight change to the section of code that draws the tiles onto the virtual background bitmap. You can remove the line that sets `n` to a random number; simply change the `blit` line, noting the change in bold. Note the last parameter of `grabframe`, which was changed from `n` to `map[n++]`. That's the only change you need to make. Now go ahead and build this puppy, and take it for a spin.

```
//now draw tiles randomly on virtual background
for (tiley=0; tiley < scroll->h; tiley+=TILEH)
```

```
{
    for (tilex=0; tilex < scroll->w; tilex+=TILEW)
    {
        //use the result of grabframe directly in blitter
        blit(grabframe(tiles, TILEW+1, TILEH+1, 0, 0, COLS, map[n++]),
            scroll, 0, 0, tilex, tiley, TILEW, TILEH);
    }
}
```

It's a lot more interesting with a real map to scroll instead of jumbled tiles randomly thrown about. I encourage you to modify and experiment with the *GameWorld* program to see what it can do. Before you start making a lot of modifications, you'll likely need the help of some status information printed on the screen. If you want, here is an addition you can make to the main loop, just following the `blit`. Again, this is optional.

```
//display status info
text_mode(-1);
textprintf(screen,font,0,0,makecol(0,0,0),
    "Position = (%d,%d)", x, y);
```

Enlarge the map to see how big you can make it. Try having the program scroll the map (with wrapping) without requiring user input. This is actually a fairly advanced topic that will be covered in future chapters on scrolling. You should definitely play around with the map array to come up with your own map, and you can even try a different set of tiles. If you have found any free game tiles on the Web (or if you have hired an artist to draw some custom tiles for your game), note the layout and size of each tile, and then you can modify the constants in the *GameWorld* program to accommodate the new tile set. See what you can come up with; experimentation is what puts the "science" in computer science.

Enhancing Tank War

I have been looking forward to this edition of *Tank War* since the first chapter in which the program was introduced (Chapter 4). If you thought the previous chapter introduced many changes to *Tank War*, you will be pleasantly surprised by all that will be put into the game in this chapter! The only drawback is that at least half of the game has been revised, but the result is well worth the effort. The game now features two (that's right, *two*!) scrolling game windows on the screen—one for each player. Shall I count the improvements? There's a new bitmap to replace the border and title; the game now uses scrolling backgrounds that you can edit to create your own custom battlefields (one for each player); the game is now double-buffered; debug messages have been removed; and the interface has been spruced up. Take a look at Figure 10.10 for a glimpse of the new game.

Terrific, isn't it? This game could seriously use some new levels with more creativity. Remember, this is a tech demo at best, something to be used as a learning experience, so

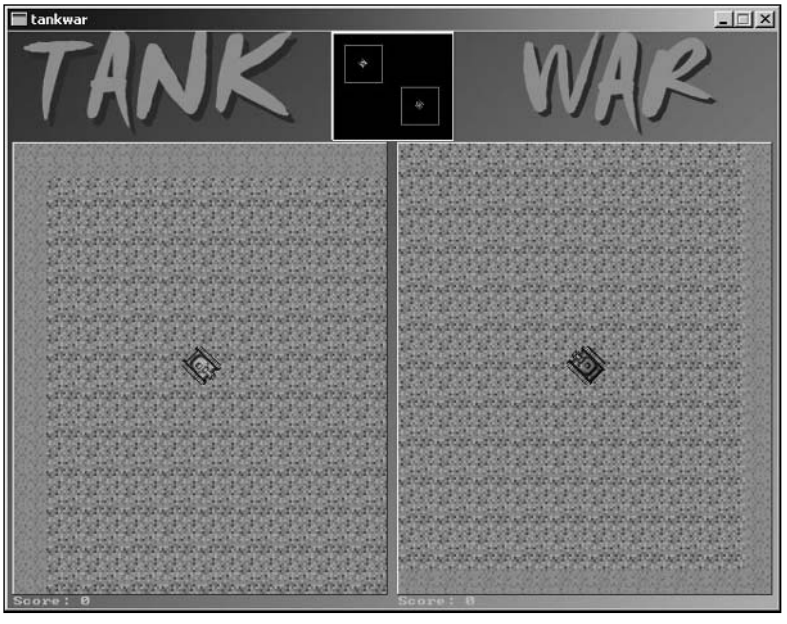


Figure 10.10 *Tank War* now features two scrolling windows, one for each player.

it has to be easy to understand, not necessarily as awesome as it could be. I leave that to you! After I've done the hard work and walked you through each step of the game, it's your job to create awesome new levels for the game. Of course, the game would also greatly benefit from some sound effects, but that will have to wait for Chapter 15, "Mastering the Audible Realm: Allegro's Sound Support."

Exploring the All-New Tank War

Since you'll be spending so much time playing this great game with your friends (unless you suffer from multiple personality disorder and are able to control both tanks at the same time), let me give you a quick tour of the game, and then we'll get started on the source code. Figure 10.11 shows what the game looks like when player 2 hits player 1. The explosion should occur on both windows at the same time, but herein lies a problem: We haven't covered timers yet! Soon enough; the next chapter covers this very important (and sorely needed) subject.

Figure 10.12 shows both tanks engulfed in explosions. D'oh! Talk about mutually assured destruction. You might be wondering where these ultra-cool explosions came from. Again, thanks to Ari Feldman's wonderful talent, we have an explosion sprite that can be rotated, tweaked, and blitted to make those gnarly boom-booms. Imagine what this game will be like with sound effects. I'm tempted to jump to that chapter right now so I can find out!

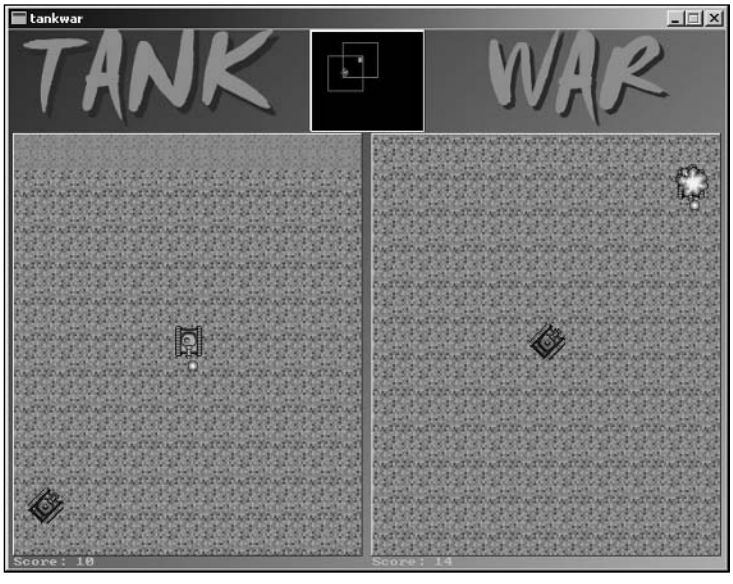


Figure 10.11 Both of the scrolling windows in *Tank War* display the bullets and explosions.

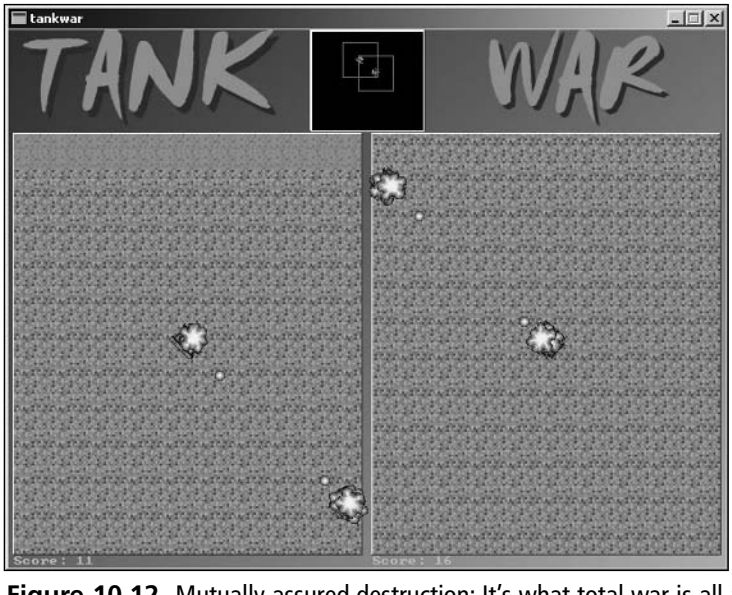


Figure 10.12 Mutually assured destruction: It's what total war is all about.

The next two figures show a sequence that is sad but true: Someone is going to die. Figure 10.13 shows player 1 having fired a bullet.

Referring to Figure 10.14—ooooh, direct hit; he's toast.

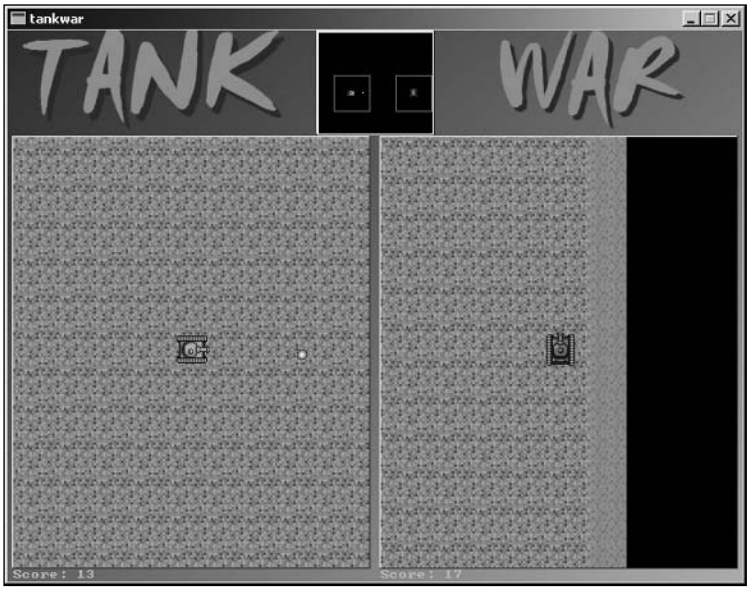


Figure 10.13 Player 1 has fired. Bullet trajectory looks good….

Figure 10.14 Player 1 would like to thank his parents, his commander, and all his new fans.

The last image shows something interesting that I want to bring to your attention when you are designing levels. Take a look at Figure 10.15.

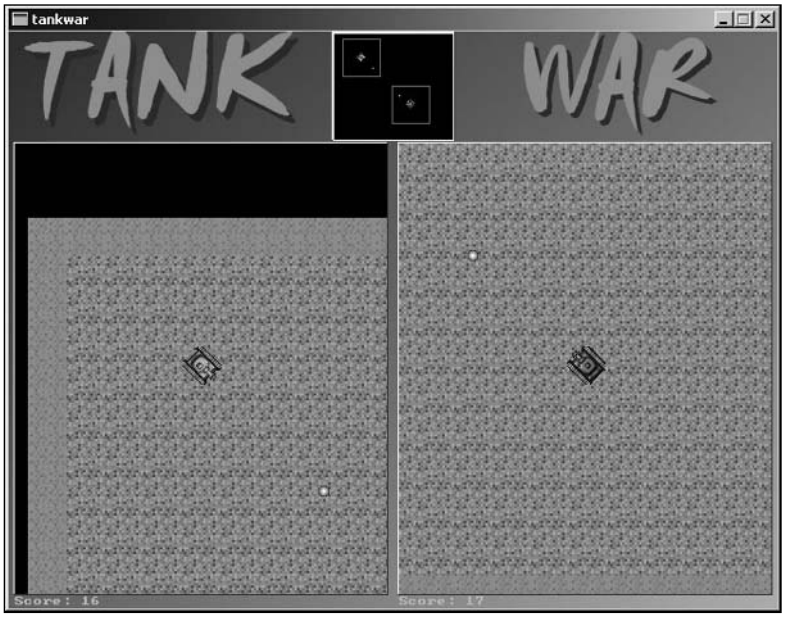


Figure 10.15 The border around the game world is filled with a blank tile.

See how the border of the game world is black? That's not just empty space; it's a blank tile from the tiles.bmp image. It is necessary to insert blanks around the edges of the map so the tanks will seem to actually move up to the edge of the map. If you omit a border like this, the tanks will not be able to reach the true border of the map. Just a little trick for you at no cost, although I'm fairly certain someone has written a book about this.

The New Tank War Source Code

It's time to get down and dirty with the new source code for *Tank War*. Let me paint the picture this way and explain things straight up. Almost everything about the source has been changed. I'm afraid a line-by-line change list isn't possible this time because more than half the game has been modified. I mean, come on—it's got dual scrolling. What do you expect, a couple of line changes? Er, sorry about that—been watching too much Strong Bad. Let's get started.

The first significant change to the game is that it is now spread across several source code files. I decided this was easier to maintain and would be easier for you to understand, so you don't have to wade through the 10-page source code listing in a single main.c file. I'll go over this with you, but you feel free to load the project from \chapter10\tankwar on the CD-ROM if you are in a hurry. I heartily recommend you follow along because there's a lot of real-world experience to be gained by watching how this game is built. Don't be a copy-paster!

Header Definitions

First up is the tankwar.h file containing all the definitions for the game.

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - tankwar.h
////////////////////////////////////////////////////////////////////////

#ifndef _TANKWAR_H
#define _TANKWAR_H

#include <conio.h>
#include <stdlib.h>
#include "allegro.h"

//define some game constants
#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define MAXSPEED 4
#define BULLETSPEED 10
#define TILEW 32
#define TILEH 32
#define TILES 39
#define COLS 10
#define MAP_ACROSS 31
#define MAP_DOWN 33
#define MAPW MAP_ACROSS * TILEW
#define MAPH MAP_DOWN * TILEH
#define SCROLLW 310
#define SCROLLH 375

//define some colors
#define TAN makecol(255,242,169)
#define BURST makecol(255,189,73)
#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)
#define GRAY makecol(128,128,128)
#define GREEN makecol(0,255,0)

//define the sprite structure
typedef struct SPRITE
```

```
{
//new elements
    int dir, alive;

    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

SPRITE mytanks[2];
SPRITE *tanks[2];
SPRITE mybullets[2];
SPRITE *bullets[2];

//declare some variables
int gameover;
int scores[2];
int scrollx[2], scrolly[2];
int startx[2], starty[2];
int tilex, tiley, n;
int radarx, radary;

//sprite bitmaps
BITMAP *tank_bmp[2][8];
BITMAP *bullet_bmp;
BITMAP *explode_bmp;

//the game map
extern int map[];

//double buffer
BITMAP *buffer;

//bitmap containing source tiles
BITMAP *tiles;

//virtual background buffer
BITMAP *scroll;
```

```
//screen background
BITMAP *back;

//function prototypes
void drawtank(int num);
void erasetank(int num);
void movetank(int num);
void explode(int num, int x, int y);
void movebullet(int num);
void drawbullet(int num);
void fireweapon(int num);
void forward(int num);
void backward(int num);
void turnleft(int num);
void turnright(int num);
void getinput();
void setuptanks();
void setupscreen();
int inside(int,int,int,int,int,int);
BITMAP *grabframe(BITMAP *, int, int, int, int, int, int);

#endif
```

Bullet Functions

I have transplanted all of the routines related to handling bullets and firing the weapons into a file called bullet.c. Isolating the bullet code in this file makes it easy to locate these functions without wading through a huge single listing. If you haven't already, add a new file to your *Tank War* project named bullet.c and type the code into this new file.

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - bullet.c
////////////////////////////////////////////////////////////////////////

#include "tankwar.h"

void explode(int num, int x, int y)
{
    int n;

    //load explode image
    if (explode_bmp == NULL)
```

```
    {
        explode_bmp = load_bitmap("explode.bmp", NULL);
    }

    //draw the explosion bitmap several times
    for (n = 0; n < 5; n++)
    {
        rotate_sprite(screen, explode_bmp,
            x + rand()%10 - 20, y + rand()%10 - 20,
            itofix(rand()%255));
        rest(30);
    }
}

void drawbullet(int num)
{
    int n;
    int x, y;

    x = bullets[num]->x;
    y = bullets[num]->y;

    //is the bullet active?
    if (!bullets[num]->alive) return;

    //draw bullet sprite
    for (n=0; n<2; n++)
    {
        if (inside(x, y, scrollx[n], scrolly[n],
            scrollx[n] + SCROLLW - bullet_bmp->w,
            scrolly[n] + SCROLLH - bullet_bmp->h))

            //draw bullet, adjust for scroll
            draw_sprite(buffer, bullet_bmp, startx[n] + x-scrollx[n],
                starty[n] + y-scrolly[n]);
    }

    //draw bullet on radar
    putpixel(buffer, radarx + x/10, radary + y/12, WHITE);

}

void movebullet(int num)
```

```
{
    int x, y, tx, ty;

    x = bullets[num]->x;
    y = bullets[num]->y;

    //is the bullet active?
    if (!bullets[num]->alive) return;

    //move bullet
    bullets[num]->x += bullets[num]->xspeed;
    bullets[num]->y += bullets[num]->yspeed;
    x = bullets[num]->x;
    y = bullets[num]->y;

    //stay within the virtual screen
    if (x < 0 || x > MAPW-6 || y < 0 || y > MAPH-6)
    {
        bullets[num]->alive = 0;
        return;
    }

    //look for a direct hit using basic collision
    tx = scrollx[!num] + SCROLLW/2;
    ty = scrolly[!num] + SCROLLH/2;
    if (inside(x,y,tx-15,ty-15,tx+15,ty+15))
    {
        //kill the bullet
        bullets[num]->alive = 0;

        //blow up the tank
            x = scrollx[!num] + SCROLLW/2;
            y = scrolly[!num] + SCROLLH/2;

            if (inside(x, y,
                scrollx[num], scrolly[num],
                scrollx[num] + SCROLLW, scrolly[num] + SCROLLH))
            {
                //draw explosion in my window
                explode(num, startx[num]+x-scrollx[num],
                    starty[num]+y-scrolly[num]);
            }
```

```
        //draw explosion in enemy window
        explode(num, tanks[!num]->x, tanks[!num]->y);
        scores[num]++;
    }
}

void fireweapon(int num)
{
    int x = scrollx[num] + SCROLLW/2;
    int y = scrolly[num] + SCROLLH/2;

    //ready to fire again?
    if (!bullets[num]->alive)
    {
        bullets[num]->alive = 1;

        //fire bullet in direction tank is facing
        switch (tanks[num]->dir)
        {
            //north
            case 0:
                bullets[num]->x = x-2;
                bullets[num]->y = y-22;
                bullets[num]->xspeed = 0;
                bullets[num]->yspeed = -BULLETSPEED;
                break;
            //NE
            case 1:
                bullets[num]->x = x+18;
                bullets[num]->y = y-18;
                bullets[num]->xspeed = BULLETSPEED;
                bullets[num]->yspeed = -BULLETSPEED;
                break;
            //east
            case 2:
                bullets[num]->x = x+22;
                bullets[num]->y = y-2;
                bullets[num]->xspeed = BULLETSPEED;
                bullets[num]->yspeed = 0;
                break;
            //SE
            case 3:
                bullets[num]->x = x+18;
```

```
                bullets[num]->y = y+18;
                bullets[num]->xspeed = BULLETSPEED;
                bullets[num]->yspeed = BULLETSPEED;
                break;
            //south
            case 4:
                bullets[num]->x = x-2;
                bullets[num]->y = y+22;
                bullets[num]->xspeed = 0;
                bullets[num]->yspeed = BULLETSPEED;
                break;
            //SW
            case 5:
                bullets[num]->x = x-18;
                bullets[num]->y = y+18;
                bullets[num]->xspeed = -BULLETSPEED;
                bullets[num]->yspeed = BULLETSPEED;
                break;
            //west
            case 6:
                bullets[num]->x = x-22;
                bullets[num]->y = y-2;
                bullets[num]->xspeed = -BULLETSPEED;
                bullets[num]->yspeed = 0;
                break;
            //NW
            case 7:
                bullets[num]->x = x-18;
                bullets[num]->y = y-18;
                bullets[num]->xspeed = -BULLETSPEED;
                bullets[num]->yspeed = -BULLETSPEED;
                break;
        }
    }
}
```

Tank Functions

Next up is a listing containing the code for managing the tanks in the game. This includes the `drawtank` and `movetank` functions. Note that `erasetank` has been erased from this version of the game. As a matter of fact, you might have noticed that there is no more erase code in the game. The scrolling windows erase everything, so there's no need to erase sprites. Add a new file to your *Tank War* project named tank.c and type this code into the new file.

```
//////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - tank.c
//////////////////////////////////////////////////////////////////////////

#include "tankwar.h"

void drawtank(int num)
{
    int dir = tanks[num]->dir;
    int x = tanks[num]->x-15;
    int y = tanks[num]->y-15;
    draw_sprite(buffer, tank_bmp[num][dir], x, y);

    //what about the enemy tank?
    x = scrollx[!num] + SCROLLW/2;
    y = scrolly[!num] + SCROLLH/2;
    if (inside(x, y,
        scrollx[num], scrolly[num],
        scrollx[num] + SCROLLW, scrolly[num] + SCROLLH))
    {
        //draw enemy tank, adjust for scroll
        draw_sprite(buffer, tank_bmp[!num][tanks[!num]->dir],
            startx[num]+x-scrollx[num]-15, starty[num]+y-scrolly[num]-15);
    }
}

void movetank(int num){
    int dir = tanks[num]->dir;
    int speed = tanks[num]->xspeed;

    //update tank position
    switch(dir)
    {
        case 0:
            scrolly[num] -= speed;
            break;
        case 1:
            scrolly[num] -= speed;
            scrollx[num] += speed;
            break;
        case 2:
            scrollx[num] += speed;
            break;
```

```
        case 3:
            scrollx[num] += speed;
            scrolly[num] += speed;
            break;
        case 4:
            scrolly[num] += speed;
            break;
        case 5:
            scrolly[num] += speed;
            scrollx[num] -= speed;
            break;
        case 6:
            scrollx[num] -= speed;
            break;
        case 7:
            scrollx[num] -= speed;
            scrolly[num] -= speed;
            break;
    }

    //keep tank inside bounds
    if (scrollx[num] < 0)
        scrollx[num] = 0;
    if (scrollx[num] > scroll->w - SCROLLW)
        scrollx[num] = scroll->w - SCROLLW;
    if (scrolly[num] < 0)
        scrolly[num] = 0;
    if (scrolly[num] > scroll->h - SCROLLH)
        scrolly[num] = scroll->h - SCROLLH;
}
```

Keyboard Input Functions

The next listing encapsulates (I just love that word!) the keyboard input functionality of the game in a single file named input.c. Herein you will find the `forward`, `backward`, `turnleft`, `turnright`, and `getinput` functions. Add a new file to your *Tank War* project named input.c and type the code into this new file.

```
//////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - input.c
//////////////////////////////////////////////////////////////////////
```

```
#include "tankwar.h"

void forward(int num)
{
    //use xspeed as a generic "speed" variable
    tanks[num]->xspeed++;
    if (tanks[num]->xspeed > MAXSPEED)
        tanks[num]->xspeed = MAXSPEED;
}

void backward(int num)
{
    tanks[num]->xspeed--;
    if (tanks[num]->xspeed < -MAXSPEED)
        tanks[num]->xspeed = -MAXSPEED;
}

void turnleft(int num)
{
    tanks[num]->dir--;
    if (tanks[num]->dir < 0)
        tanks[num]->dir = 7;
}

void turnright(int num)
{
    tanks[num]->dir++;
    if (tanks[num]->dir > 7)
        tanks[num]->dir = 0;
}

void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])   gameover = 1;

    //WASD - SPACE keys control tank 1
    if (key[KEY_W])     forward(0);
    if (key[KEY_D])     turnright(0);
    if (key[KEY_A])     turnleft(0);
    if (key[KEY_S])     backward(0);
    if (key[KEY_SPACE]) fireweapon(0);
```

```
    //arrow - ENTER keys control tank 2
    if (key[KEY_UP])    forward(1);
    if (key[KEY_RIGHT]) turnright(1);
    if (key[KEY_DOWN])  backward(1);
    if (key[KEY_LEFT])  turnleft(1);
    if (key[KEY_ENTER]) fireweapon(1);

    //short delay after keypress
    rest(20);

}
```

Game Setup Functions

The game setup functions are easily the most complicated functions of the entire game, so it is a good thing that they are run only once when the game starts. Here you will find the `setupscreen` and `setuptanks` functions. Add a new file to your *Tank War* project named setup.c and type the following code into this new file.

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - setup.c
////////////////////////////////////////////////////////////////////////

#include "tankwar.h"

void setuptanks()
{
    int n;

    //configure player 1's tank
    tanks[0] = &mytanks[0];
    tanks[0]->x = 30;
    tanks[0]->y = 40;
    tanks[0]->xspeed = 0;
    scores[0] = 0;
    tanks[0]->dir = 3;

    //load first tank bitmap
    tank_bmp[0][0] = load_bitmap("tank1.bmp", NULL);

    //rotate image to generate all 8 directions
    for (n=1; n<8; n++)
```

```
{
    tank_bmp[0][n] = create_bitmap(32, 32);
    clear_to_color(tank_bmp[0][n], makecol(255,0,255));
    rotate_sprite(tank_bmp[0][n], tank_bmp[0][0],
        0, 0, itofix(n*32));
}

//configure player 2's tank
tanks[1] = &mytanks[1];
tanks[1]->x = SCREEN_W-30;
tanks[1]->y = SCREEN_H-30;
tanks[1]->xspeed = 0;
scores[1] = 0;
tanks[1]->dir = 7;

//load second tank bitmap
tank_bmp[1][0] = load_bitmap("tank2.bmp", NULL);

//rotate image to generate all 8 directions
for (n=1; n<8; n++)
{
    tank_bmp[1][n] = create_bitmap(32, 32);
    clear_to_color(tank_bmp[1][n], makecol(255,0,255));
    rotate_sprite(tank_bmp[1][n], tank_bmp[1][0],
        0, 0, itofix(n*32));
}

//load bullet image
if (bullet_bmp == NULL)
    bullet_bmp = load_bitmap("bullet.bmp", NULL);

//initialize bullets
for (n=0; n<2; n++)
{
   bullets[n] = &mybullets[n];
   bullets[n]->x = 0;
   bullets[n]->y = 0;
   bullets[n]->width = bullet_bmp->w;
   bullets[n]->height = bullet_bmp->h;
}

//center tanks inside scroll windows
tanks[0]->x = 5 + SCROLLW/2;
```

```
    tanks[0]->y = 90 + SCROLLH/2;
    tanks[1]->x = 325 + SCROLLW/2;
    tanks[1]->y = 90 + SCROLLH/2;
}

void setupscreen()
{
    int ret;

    //set video mode
    set_color_depth(16);
    ret = set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    text_mode(-1);

    //create the virtual background
    scroll = create_bitmap(MAPW, MAPH);
    if (scroll == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating virtual background");
        return;
    }

    //load the tile bitmap
    tiles = load_bitmap("tiles.bmp", NULL);
    if (tiles == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error loading tiles.bmp file");
        return;
    }

    //now draw tiles on virtual background
    for (tiley=0; tiley < scroll->h; tiley+=TILEH)
    {
        for (tilex=0; tilex < scroll->w; tilex+=TILEW)
        {
            //use the result of grabframe directly in blitter
```

```
            blit(grabframe(tiles, TILEW+1, TILEH+1, 0, 0, COLS,
                map[n++]), scroll, 0, 0, tilex, tiley, TILEW, TILEH);
        }
    }

    //done with tiles
    destroy_bitmap(tiles);

    //load screen background
    back = load_bitmap("background.bmp", NULL);
    if (back == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error loading background.bmp file");
        return;
    }

    //create the double buffer
    buffer = create_bitmap(WIDTH, HEIGHT);
    if (buffer == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating double buffer");
        return;
    }

    //position the radar
    radarx = 270;
    radary = 1;

    //position each player
    scrollx[0] = 100;
    scrolly[0] = 100;
    scrollx[1] = MAPW - 400;
    scrolly[1] = MAPH - 500;

    //position the scroll windows
    startx[0] = 5;
    starty[0] = 93;
    startx[1] = 325;
    starty[1] = 93;

}
```

Main Function

You have greatly simplified the main.c source code file for *Tank War* by moving so much code into separate source files. Now in main.c, you have a declaration for the map array. Why? Because it was not possible to include the declaration inside the tankwar.h header file, only an `extern` reference to the array definition inside a source file. As with the previous code listings, this one is heavily commented so you can examine it line by line. Take particular note of the map array definition. To simplify and beautify the listing, I have defined `B` equal to 39; as you can see, this refers to the blank space tile around the edges of the map.

The game also features a new background image to improve the appearance of the game. Figure 10.16 shows the image, which acts as a template for displaying game graphics.

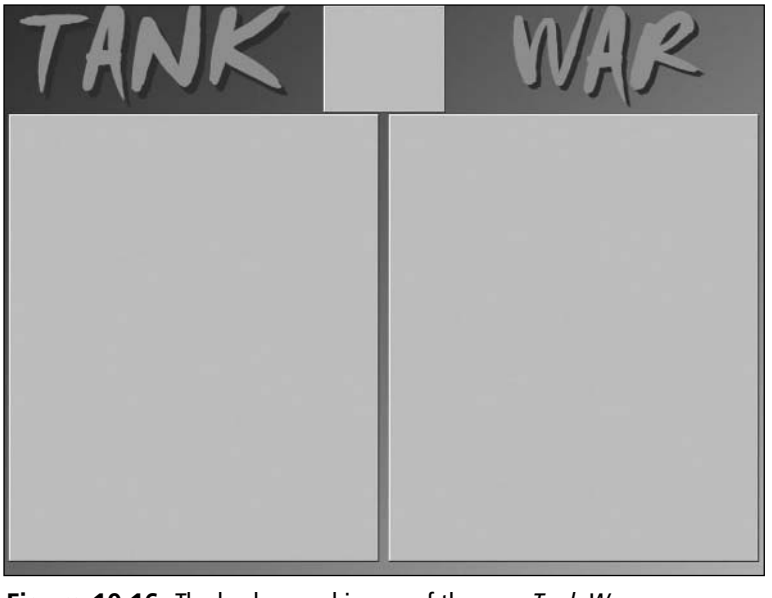


Figure 10.16 The background image of the new *Tank War*

```
//////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 5 - main.c
//////////////////////////////////////////////////////////////////////////

#include "tankwar.h"

#define B 39
```

```
int map[MAPW*MAPH] = {
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,B,B,B,
    B,B,B,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B
};

//perform basic collision detection
int inside(int x,int y,int left,int top,int right,int bottom)
{
    if (x > left && x < right && y > top && y < bottom)
        return 1;
    else
        return 0;
```

```
}

//reuse our friendly tile grabber from chapter 9
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}

//main function
void main(void)
{
    //initialize the game
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    setupscreen();
    setuptanks();

    //game loop
    while(!gameover)
    {
        //move the tanks and bullets
        for (n=0; n<2; n++)
        {
            movetank(n);
            movebullet(n);
        }

        //draw background bitmap
        blit(back, buffer, 0, 0, 0, 0, back->w, back->h);
```

```
//draw scrolling windows
for (n=0; n<2; n++)
    blit(scroll, buffer, scrollx[n], scrolly[n],
        startx[n], starty[n], SCROLLW, SCROLLH);

//update the radar
rectfill(buffer,radarx+1,radary+1,radarx+99,radary+88,BLACK);
rect(buffer,radarx,radary,radarx+100,radary+89,WHITE);

//draw mini tanks on radar
for (n=0; n<2; n++)
    stretch_sprite(buffer, tank_bmp[n][tanks[n]->dir],
        radarx + scrollx[n]/10 + (SCROLLW/10)/2-4,
        radary + scrolly[n]/12 + (SCROLLH/12)/2-4,
        8, 8);

//draw player viewport on radar
for (n=0; n<2; n++)
    rect(buffer,radarx+scrollx[n]/10, radary+scrolly[n]/12,
        radarx+scrollx[n]/10+SCROLLW/10,
        radary+scrolly[n]/12+SCROLLH/12, GRAY);

//display score
for (n=0; n<2; n++)
    textprintf(buffer, font, startx[n], HEIGHT-10,
        BURST, "Score: %d", scores[n]);

//draw the tanks and bullets
for (n=0; n<2; n++)
{
    drawtank(n);
    drawbullet(n);
}

//refresh the screen
acquire_screen();
blit(buffer, screen, 0, 0, 0, 0, WIDTH, HEIGHT);
release_screen();

//check for keypresses
if (keypressed())
    getinput();
```

```
        //slow the game down
        rest(20);
    }

    //destroy bitmaps
    destroy_bitmap(explode_bmp);
    destroy_bitmap(back);
    destroy_bitmap(scroll);
    destroy_bitmap(buffer);
    for (n=0; n<8; n++)
    {
        destroy_bitmap(tank_bmp[0][n]);
        destroy_bitmap(tank_bmp[1][n]);
    }
    return;
}
END_OF_MAIN();
```

Summary

This marks the end of yet another graphically intense chapter. In it, I talked about scrolling backgrounds and spent most of the time discussing tile-based backgrounds—how they are created and how to use them in a game. Working with tiles to create a scrolling game world is by no means an easy subject! If you skimmed over any part of this chapter, be sure to read through it again before you move on because the next three chapters dig even deeper into scrolling. You also opened up the *Tank War* project and made some huge changes to the game, not the least of which was creating dual scrolling windows—one for each player! This is the last major change to the game. From this point forward, you will make only minor additions (such as sound effects, music, and timing) in upcoming chapters. So, be happy in the knowledge that you have completed the vast majority of the work on *Tank War*.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Does Allegro provide support for background scrolling?
   A. Yes, but the functionality is obsolete.
   B. Yes, and it works great!
   C. Yes, but it needs some work.
   D. Not even.

2. What does a scroll window show?
   A. A small part of a larger game world.
   B. A window filled with sprites.
   C. A scroll that explains the rules of the game.
   D. A portion of the double-buffer.

3. Which of the programs in this chapter demonstrated bitmap scrolling for the first time?
   A. *Tank War*
   B. *TileScroll*
   C. *ScrollScreen*
   D. *GameWorld*

4. Why should a scrolling background be designed?
   A. To sell it as a marketable game engine.
   B. To devise a new programming technique.
   C. To mesmerize the gaming public.
   D. To achieve the goals of the game.

5. Which process uses an array of images to construct the background as it is displayed?
   A. Iterating
   B. Blitting
   C. Tiling
   D. Constructing

6. What is the best way to create a tile map of the game world?
   A. By using a map editor.
   B. By randomly generating the map at run time.
   C. By using an array.
   D. By stealing maps off the Internet.

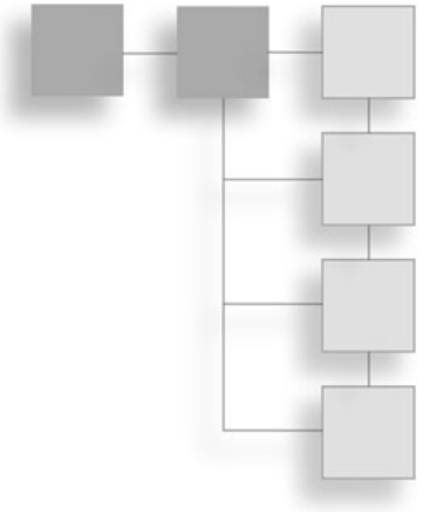
7. What type of object comprises a typical tile map?
   A. Variables
   B. Arrays
   C. Numbers
   D. Breakpoints

8. What was the size of the virtual background in the *GameWorld* program?
   A. 800×800
   B. 16384×65536
   C. 640×480
   D. 1600×1200

9. How many virtual backgrounds are used in the new version of *Tank War*?
   A. 0
   B. 1
   C. 2
   D. 3

10. How many scrolling windows are used in the new *Tank War*?
    A. 0
    B. 1
    C. 2
    D. 3

CHAPTER 11



Timers, Interrupt Handlers, and Multi-Threading

This chapter covers the extremely critical subject of timing as it relates to game programming. Until now, you have used the primitive `rest` function to slow down your example programs in the past 10 chapters, and it has been hit or miss as far as how well it worked. In this chapter, I'll go over Allegro's support for timers and interrupt handlers to calculate the frame rate and slow down a program to a fixed rate. This chapter also delves into the compelling subject of multi-threading with an explanation of how to use threads to enhance a game. It also contains a demonstration program.

Here is a breakdown of the major topics in this chapter:

- Understanding timers
- Working with interrupt handlers
- Using timed game loops
- Understanding multi-threading

Timers

Timing is critical in a game. Without an accurate means to slow down a game to a fixed rate, the game will be influenced by the speed of the computer running it, adversely affecting gameplay. (This usually renders the game unplayable.) Allegro has support for timing a game using `rest`, but a far more powerful feature is the interrupt handler, which you can use to great effect.

Installing and Removing the Timer

You have already used Allegro's timer functions without much explanation in prior chapters because it's almost impossible to write even a simple demonstration program without

some kind of timing involved. To install the primary timer in Allegro that makes it possible to use the timer functions and interrupt handlers, you use the `install_timer` function.

```
int install_timer();
```

You must be sure to call `install_timer` before you create any timer routines and also before you display a mouse pointer or play FLI animations or MIDI music because these features all rely on the timer. So it's up to you! This function returns zero on success, although it is so unlikely to error out that I never check it.

Allegro will automatically remove the timer when the program ends (or when `allegro_exit` is called), but you can call the `remove_timer` function if you want to remove the timer before the program ends.

```
void remove_timer();
```

Slowing Down the Program

You have seen the `rest` function used frequently in the sample programs in prior chapters, so it should be familiar to you. For reference, here is the declaration:

```
void rest(long time);
```

You can pass any number of milliseconds to `rest` and the program will pause for that duration, after which control will pass to the next line in the program. This is very effective for slowing down a game, of course, but it can also be used to pause for a short time when you are waiting for threads to terminate (as you'll learn about later in this chapter). Once Allegro has taken over the timer, the standard `delay` function will no longer work, although you haven't been using `delay` so that should not come as a surprise.

One feature that I haven't gone over yet is the `rest_callback` function. Have you noticed that Allegro provides a callback for almost everything it does? This is a fine degree of control seldom found in game development libraries; obviously, Allegro was developed by individuals with a great deal of experience, who had the foresight to include some very useful callback functions. Here is the declaration:

```
void rest_callback(long time, void (*callback)())
```

This function works like `rest`, but instead of doing nothing, a callback function is called during the delay period so your program can continue working even while timing is in effect to slow the game down.

Here's an example of how you would call the function:

```
//slow the game down
rest_callback(8, rest1);
```

The `rest1` callback function is very simple; it contains no parameters.

```
void rest1(void)
{
    //time to rest, or do some work?
}
```

This is a good time to update some values, such as the frame rate, but I would not recommend doing any time-intensive processing during the rest callback because it must return quickly to avoid messing up the game's timing. The *TimedLoop* program later in this chapter will demonstrate how to use the `rest_callback` function.

The TimerTest Program

Because none of the sample programs in the book up to this point have used effective timing techniques, I've written a program to calculate the frame rate and display this value along with a count of seconds passing. The *TimerTest* program will be used in the next two segments of the chapter, so its listing is somewhat extensive at this point. However, the next two segments will provide simple code changes to this program to save time and space.

Figure 11.1 shows the *TimerTest* program running. As you can see, it is very graphical, with a background and many sprites moving across the screen. I owe a debt of thanks to Ari Feldman (http://www.arifeldman.com) again for allowing me to use his excellent SpriteLib to populate this chapter with such interesting, high-quality sprites.

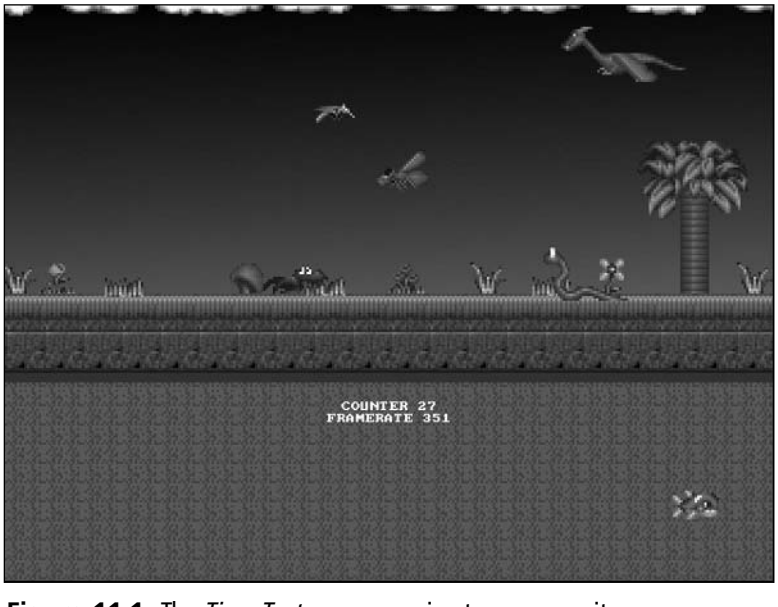


Figure 11.1 The *TimerTest* program animates many sprites over a background scene. Sprites courtesy of Ari Feldman.

The first section of code includes the defines, structs, and variables.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "allegro.h"

#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define MAX 6
#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

//define the sprite structure
typedef struct SPRITE
{
    int dir, alive;
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

//variables
BITMAP *back;
BITMAP *temp;
BITMAP *sprite_images[10][10];
SPRITE *sprites[10];
BITMAP *buffer;
int n, f;

//timer variables
int start;
int counter;
int ticks;
int framerate;
```

The next section of code for the *TimerTest* program includes the sprite-handling functions `updatesprite`, `warpsprite`, `grabframe`, and `loadsprites`. These functions should be familiar from previous chapters.

```
void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void warpsprite(SPRITE *spr)
{
    //simple screen warping behavior
    //Allegro takes care of clipping
    if (spr->x < 0 - spr->width)
    {
        spr->x = SCREEN_W;
    }
```

```
    else if (spr->x > SCREEN_W)
    {
        spr->x = 0 - spr->width;
    }

    if (spr->y < 0)
    {
        spr->y = SCREEN_H - spr->height-1;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
        spr->y = 0;
    }

}

//reuse our friendly tile grabber from chapter 9
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}

void loadsprites(void)
{
    //load dragon sprite
    temp = load_bitmap("dragon.bmp", NULL);
    for (n=0; n<6; n++)
        sprite_images[0][n] = grabframe(temp,128,64,0,0,3,n);
    destroy_bitmap(temp);

    //initialize the dragon (sprite 0)
    sprites[0] = malloc(sizeof(SPRITE));
```

```
sprites[0]->x = 500;
sprites[0]->y = 0;
sprites[0]->width = sprite_images[0][0]->w;
sprites[0]->height = sprite_images[0][0]->h;
sprites[0]->xdelay = 1;
sprites[0]->ydelay = 0;
sprites[0]->xcount = 0;
sprites[0]->ycount = 0;
sprites[0]->xspeed = -5;
sprites[0]->yspeed = 0;
sprites[0]->curframe = 0;
sprites[0]->maxframe = 5;
sprites[0]->framecount = 0;
sprites[0]->framedelay = 5;
sprites[0]->animdir = 1;

//load fish sprite
temp = load_bitmap("fish.bmp", NULL);
for (n=0; n<3; n++)
    sprite_images[1][n] = grabframe(temp,64,32,0,0,3,n);
destroy_bitmap(temp);

//initialize the fish (sprite 1)
sprites[1] = malloc(sizeof(SPRITE));
sprites[1]->x = 300;
sprites[1]->y = 400;
sprites[1]->width = sprite_images[1][0]->w;
sprites[1]->height = sprite_images[1][0]->h;
sprites[1]->xdelay = 1;
sprites[1]->ydelay = 0;
sprites[1]->xcount = 0;
sprites[1]->ycount = 0;
sprites[1]->xspeed = 3;
sprites[1]->yspeed = 0;
sprites[1]->curframe = 0;
sprites[1]->maxframe = 2;
sprites[1]->framecount = 0;
sprites[1]->framedelay = 8;
sprites[1]->animdir = 1;

//load crab sprite
temp = load_bitmap("crab.bmp", NULL);
for (n=0; n<4; n++)
```

```
        sprite_images[2][n] = grabframe(temp,64,32,0,0,4,n);
    destroy_bitmap(temp);

    //initialize the crab (sprite 2)
    sprites[2] = malloc(sizeof(SPRITE));
    sprites[2]->x = 300;
    sprites[2]->y = 212;
    sprites[2]->width = sprite_images[2][0]->w;
    sprites[2]->height = sprite_images[2][0]->h;
    sprites[2]->xdelay = 6;
    sprites[2]->ydelay = 0;
    sprites[2]->xcount = 0;
    sprites[2]->ycount = 0;
    sprites[2]->xspeed = 2;
    sprites[2]->yspeed = 0;
    sprites[2]->curframe = 0;
    sprites[2]->maxframe = 3;
    sprites[2]->framecount = 0;
    sprites[2]->framedelay = 20;
    sprites[2]->animdir = 1;

    //load bee sprite
    temp = load_bitmap("bee.bmp", NULL);
    for (n=0; n<6; n++)
        sprite_images[3][n] = grabframe(temp,50,40,0,0,6,n);
    destroy_bitmap(temp);

    //initialize the bee (sprite 3)
    sprites[3] = malloc(sizeof(SPRITE));
    sprites[3]->x = 100;
    sprites[3]->y = 120;
    sprites[3]->width = sprite_images[3][0]->w;
    sprites[3]->height = sprite_images[3][0]->h;
    sprites[3]->xdelay = 1;
    sprites[3]->ydelay = 0;
    sprites[3]->xcount = 0;
    sprites[3]->ycount = 0;
    sprites[3]->xspeed = -3;
    sprites[3]->yspeed = 0;
    sprites[3]->curframe = 0;
    sprites[3]->maxframe = 5;
    sprites[3]->framecount = 0;
    sprites[3]->framedelay = 8;
```

```
  sprites[3]->animdir = 1;

//load skeeter sprite
  temp = load_bitmap("skeeter.bmp", NULL);
  for (n=0; n<6; n++)
      sprite_images[4][n] = grabframe(temp,50,40,0,0,6,n);
  destroy_bitmap(temp);

  //initialize the skeeter (sprite 4)
  sprites[4] = malloc(sizeof(SPRITE));
  sprites[4]->x = 500;
  sprites[4]->y = 70;
  sprites[4]->width = sprite_images[4][0]->w;
  sprites[4]->height = sprite_images[4][0]->h;
  sprites[4]->xdelay = 1;
  sprites[4]->ydelay = 0;
  sprites[4]->xcount = 0;
  sprites[4]->ycount = 0;
  sprites[4]->xspeed = 4;
  sprites[4]->yspeed = 0;
  sprites[4]->curframe = 0;
  sprites[4]->maxframe = 4;
  sprites[4]->framecount = 0;
  sprites[4]->framedelay = 2;
  sprites[4]->animdir = 1;

//load snake sprite
  temp = load_bitmap("snake.bmp", NULL);
  for (n=0; n<8; n++)
      sprite_images[5][n] = grabframe(temp,100,50,0,0,4,n);
  destroy_bitmap(temp);

  //initialize the snake (sprite 5)
  sprites[5] = malloc(sizeof(SPRITE));
  sprites[5]->x = 350;
  sprites[5]->y = 200;
  sprites[5]->width = sprite_images[5][0]->w;
  sprites[5]->height = sprite_images[5][0]->h;
  sprites[5]->xdelay = 1;
  sprites[5]->ydelay = 0;
  sprites[5]->xcount = 0;
  sprites[5]->ycount = 0;
  sprites[5]->xspeed = -2;
```

```
    sprites[5]->yspeed = 0;
    sprites[5]->curframe = 0;
    sprites[5]->maxframe = 4;
    sprites[5]->framecount = 0;
    sprites[5]->framedelay = 6;
    sprites[5]->animdir = 1;
}
```

The last section of code for the *TimerTest* program includes the `main` function, which initializes the program and includes the main loop. This program is lengthy in setup but efficient in operation because all the sprites are contained within arrays that can be updated as a group within a `for` loop. I have highlighted timer-related code in bold.

```
void main(void)
{
    //initialize
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    srand(time(NULL));
    text_mode(-1);
    install_keyboard();
    install_timer();

    //create double buffer
    buffer = create_bitmap(SCREEN_W,SCREEN_H);

    //load and draw the blocks
    back = load_bitmap("background.bmp", NULL);
    blit(back,buffer,0,0,0,0,back->w,back->h);

    //load and set up sprites
    loadsprites();

    //game loop
    while (!key[KEY_ESC])
    {
        //restore the background
        for (n=0; n<MAX; n++)
            blit(back, buffer, sprites[n]->x, sprites[n]->y,
                sprites[n]->x, sprites[n]->y,
                sprites[n]->width, sprites[n]->height);
```

```
    //update the sprites
    for (n=0; n<MAX; n++)
    {
        updatesprite(sprites[n]);
        warpsprite(sprites[n]);
        draw_sprite(buffer, sprite_images[n][sprites[n]->curframe],
            sprites[n]->x, sprites[n]->y);
    }

    //update ticks
    ticks++;

    //calculate framerate once per second
    if (clock() > start + 1000)
    {
        counter++;
        start = clock();
        framerate = ticks;
        ticks = 0;
    }

    //display framerate
    blit(back, buffer, 320-70, 330, 320-70, 330, 140, 20);
    textprintf_centre(buffer,font,320,330,WHITE,"COUNTER %d",
        counter);
    textprintf_centre(buffer,font,320,340,WHITE,"FRAMERATE %d",
        framerate);

    //update the screen
    acquire_screen();
    blit(buffer,screen,0,0,0,0,SCREEN_W-1,SCREEN_H-1);
    release_screen();
}

//remove objects from memory
destroy_bitmap(back);
destroy_bitmap(buffer);

for (n=0; n<MAX; n++)
{
    for (f=0; f<sprites[n]->maxframe+1; f++)
        destroy_bitmap(sprite_images[n][f]);
```

```
        free(sprites[n]);
    }

    return;
}

END_OF_MAIN();
```

Interrupt Handlers

The `rest` and `rest_callback` functions are useful for slowing down a game, but Allegro's support for interrupt handlers is the real power of the timer functionality. Allegro allows you to easily create an interrupt handler routine that will execute at a specified interval. This is a limited form of multi-threading in concept, although interrupt handlers do not run in parallel, but sequentially and based on the interval. Because no two interrupt handlers will ever be running at the same time, you don't need to worry about corrupted data as you do with threading.

Creating an Interrupt Handler

After you have installed the timer using `install_timer`, you can create one or more interrupt handlers using the `install_int` function.

```
int install_int(void (*proc)(), int speed);
```

This function accepts the name of an interrupt handler callback function and the duration by which that function should be called. After you install the interrupt handler, you don't need to call it because the handler function is called automatically at the interval specified (in milliseconds).

tip

If you forget to call `install_timer` before you create an interrupt handler, don't worry; Allegro is smart enough to call `install_timer` automatically if it is not already running.

There are a limited number of interrupt handlers available for your program's use, so if the function fails to create a new handler it will return a non-zero, with a zero on success. The interrupt callback function is called by Allegro, not the operating system, so it doesn't need any special wrapper code (as with traditional interrupt handlers); it can be a regular C function. Because timing is crucial, I recommend that you don't use an interrupt callback function for any real processing; use it to set flags, increment a frame rate counter, and that sort of thing, and then do any real work in the main function using these flags or counters. Try not to take too much time in an interrupt callback, in other words.

Not all operating systems require it, but Allegro provides a means to secure variables and functions that are used by an interrupt. You can use `LOCK_VARIABLE` and `LOCK_FUNCTION` to identify them to Allegro. You will also want to declare any global variables used by the interrupt as `volatile`.

Removing an Interrupt Handler

It is not absolutely necessary to remove an interrupt handler from your program because `allegro_exit` will remove the handler for you, but it is nevertheless a good idea to have your programs clean up after themselves to eliminate even the possibility of a difficult-to-find bug. You can use the `remove_int` function to remove an interrupt handler.

```
void remove_int(void (*proc)());
```

Simply pass the name of the interrupt callback function to `remove_int` and that will stop the interrupt from calling the function.

The InterruptTest Program

The real power of an interrupt handler is obvious in practice, when you do something essential (such as calculate the frame rate) inside the interrupt callback function. I have made some changes to the *TimerTest* program you saw in the last section. Instead of using a `ticks` variable and the `clock` function to determine when to mark each second, this new program uses an interrupt handler that is set to 1,000 milliseconds to automatically tick off a second. To make things easier, I have modified the *TimerTest* program (which was quite lengthy) to use an interrupt instead of a simple timer; only a few lines of code need to be changed. Figure 11.2 shows the output of the new version of the program, which is now called *InterruptTest*.

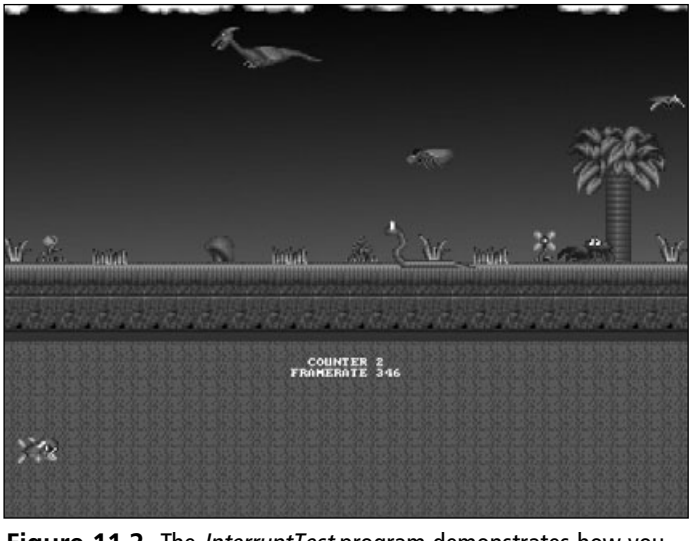


Figure 11.2 The *InterruptTest* program demonstrates how you can use an interrupt callback function to calculate the frame rate.

If you look back a few pages to Figure 11.1, you might notice that it had a slightly higher frame rate than this new *InterruptTest* program (from 351 fps to 346 fps). The difference is negligible and would not be noticed in a timed game loop in which the frame rate is fixed. However, this does demonstrate that the interrupt handler adds some overhead to the program; it is further proof that the callback function should run as quickly as possible to avoid adding to that overhead.

Let's get started on the changes, few that they are. The first change is up near the top of the program, where the `counter`, `ticks`, and `framerate` variables are declared. Add `volatile` to their definitions.

```
//timer variables
volatile int counter;
volatile int ticks;
volatile int framerate;
```

Next, you need to add the interrupt handler callback function, `timer1`, to the program. You can add this function right above `main` or up at the top of the program, as long as it's visible to `main`. Note how simple this function is; it increments `counter` (for seconds), sets the `framerate` variable, and resets the `ticks` variable.

```
//calculate framerate once per second
void timer1(void)
{
    counter++;
    framerate = ticks;
    ticks=0;
}
END_OF_FUNCTION(timer1)
```

The next change takes place in `main`, where the variables and callback function are identified to Allegro as interrupt-aware and the interrupt handler is created. You can add this code right above the `while` loop inside `main`.

```
//lock interrupt variables
LOCK_VARIABLE(counter);
LOCK_VARIABLE(framerate);
LOCK_VARIABLE(ticks);
LOCK_FUNCTION(timer1);
//create the interrupt handler
install_int(timer1, 1000);
```

Okay, now for the last change, which is really only a deletion. Because the timer code was moved into the interrupt callback function, you need to delete it from `main`. Look for the code highlighted in bold in the following listing (and commented out), and remove those lines from the program.

```
//update ticks
ticks++;

//calculate framerate once per second
//if (clock() > start + 1000)
//{
//    counter++;
//    start = clock();
//    framerate = ticks;
//    ticks = 0;
//}

//display framerate
blit(back, buffer, 320-70, 330, 320-70, 330, 140, 20);
textprintf_centre(buffer,font,320,330,WHITE,"COUNTER %d", counter);
textprintf_centre(buffer,font,320,340,WHITE,"FRAMERATE %d", framerate);
```

Using Timed Game Loops

You have now learned how to use a timer to calculate the frame rate of the program with a simple timer and also an interrupt handler. But so what if you know the frame rate; how does that keep the game running at a stable rate regardless of the computer hardware running it? You need to use this new functionality to actually limit the speed of the game so it will look the same on any computer.

Slowing Down the Gameplay…Not the Game

The key point here is not to slow down the gameplay, but the graphics rendering on the screen. Any blitting going on will (and should) be as fast as possible, but the pace of the game must be maintained or it will be unplayable. You have already seen what a high-speed game loop looks like by running the *TimerTest* and *InterruptTest* programs. What you need now is a way to slow down the program to a predictable rate.

Now you return to the `rest_callback` function introduced at the start of this chapter to help create a timed game loop. There is no new functionality in this section, just an example of how to use what you've learned so far to improve gameplay. You are free to use any target frame rate you want for your game, but as a general rule a value between 30 and 60 fps is a good target to shoot for. Why? Any slower than 30 fps and the game will seem sluggish; any faster than 60 and the game will feel too frenetic. You do want to blit all the graphics as quickly as possible, and then if there are cycles left over after that is done, you need to slow down the game so one frame of the game is displayed at a fixed interval.

The TimedLoop Program

Now you can modify the program again to give it a timed loop that will keep the program running fluidly and predictably whether it's running on a Pentium II 450 or an Athlon XP 3700+ CPU. First, open up the *InterruptTest* program as a basis, so the program will still include the interrupt handler to calculate the frame rate. The new program, which will be called *TimedLoop*, is simply a modification of that previous program, so only a few line changes are needed. Figure 11.3 shows the program running. Take note of the new status message that displays the resting value.

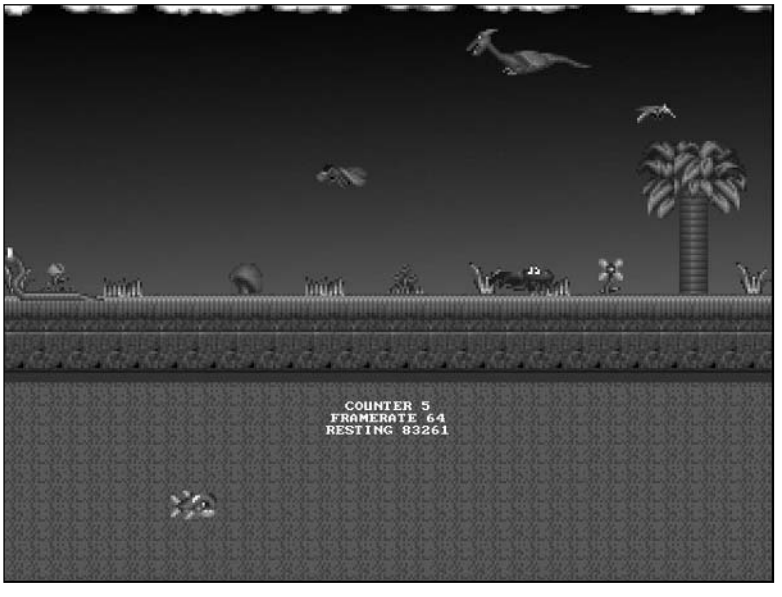


Figure 11.3 The *TimedLoop* program demonstrates how to slow a program down to a consistent frame rate.

First, up near the top of the program, add another `volatile` variable.

```
//timer variables
volatile int counter;
volatile int ticks;
volatile int framerate;
volatile int resting, rested;
```

Scroll down to the `timer1` interrupt callback function and add a line to it.

```
//calculate framerate every second
void timer1(void)
{
    counter++;
```

```
    framerate = ticks;
    ticks=0;
    rested=resting;
}
END_OF_FUNCTION(timer1)
```

Now you create the function that is called by `rest_callback`. You can add this function below `timer1`.

```
//do something while resting (?)
void rest1(void)
{
    resting++;
}
```

The next change takes place in `main`, adding the code to call the `rest_callback` function, which is a call to `rest1`, just added. Note also the changes to the section of code that displays the counter and frame rate. I have changed the last parameter of `blit` from 20 to 30 to erase the new line, which is also listed below, highlighted in bold. This displays the number of ticks that transpired while the program was waiting inside the `rest1` callback function.

```
//update ticks
ticks++;

//slow the game down
resting=0;
rest_callback(8, rest1);

//display framerate
blit(back, buffer, 320-70, 330, 320-70, 330, 140, 30);
textprintf_centre(buffer,font,320,330,WHITE,"COUNTER %d", counter);
textprintf_centre(buffer,font,320,340,WHITE,"FRAMERATE %d", framerate);
textprintf_centre(buffer,font,320,350,WHITE,"RESTING %d", rested);
```

Multi-Threading

Every modern operating system uses threads for essential and basic operation and would not be nearly as versatile without threads. A *thread* is a process that runs within the memory space of a single program but is executed separately from that program. This section will provide a short overview of multi-threading and how it can be used (fairly easily) to enhance a game. I will not go into the vast details of threaded programming because the topic is too huge and unwieldy to fully explain in only a few pages. Instead, I will provide you with enough information and example code that you will be able to start using threads.

To be multi-threaded, a program will create at least one thread that will run in addition to that program's main loop. Any time a program uses more than one thread, you must take extreme caution when working with data that is potentially shared between threads. It is generally safe for a program to share data with a single thread (although it is not recommended), but when more than one thread is in use, you must use a protection scheme to protect the data from being manipulated by two threads at the same time.

To protect data, you can make use of mutexes that will lock data inside a single thread until it is safe to unlock the data for use in the main program or in another thread. The locking and unlocking must be done inside a loop that runs continuously inside the thread callback function. Note that if you do not have a loop inside your thread function, it will run once and terminate. The idea is to keep the thread running—doing something—while the main program is doing the delegating work. You should think of a thread as a new employee who has been hired to alleviate the amount of work done by the program (or rather, by the main thread). To demonstrate, at the end of this section I'll walk you through a multi-threaded example in which two distinct threads control two identical sprites on the screen, with one thread running faster than the other, while the program's main loop does nothing more than blit the double-buffer to the screen.

Abstracting the Parallel Processing Problem

We disseminate the subject as if it's just another C function, but threads were at one time an extraordinary achievement that was every bit as exciting as the first connection of ARPAnet in 1969 or the first working version of UNIX. In the 1980s, parallel programming was as hip as virtual reality, but like the latter term, it was not to be a true reality until the early 1990s. *Multi-threaded programming* is the engineer's term for parallel processing and is a solution that has been proven to work. The key to parallel processing came when software engineers realized that the processor is not the focus; rather, software design is. In the words of Agent Smith from *The Matrix*, "We lacked a programming language with which to construct your world."

A single-processor system should be able to run multiple threads. Once that goal was realized, adding two or more processors to a system provided the ability to delegate those threads, and this was a job for the operating system. No longer tasked with designing a parallel-processing architecture, engineers in both the electronics and software fields abstracted the problem so the two were not reliant upon each other. A single program can run on a motherboard with four CPUs and push all of those processors to the limit, if that single program invokes multiple threads. As such, the programs themselves were treated as single threads. And yet, there can be many non-threaded programs running on our fictional quad-processor system, and it might not be taxed at all. It depends on what each program is doing.

Math-intensive processes, such as 3D rendering, can eat a CPU for breakfast. But with the advent of threading in modern operating systems, programs such as 3D Studio Max, Maya, Lightwave, and Photoshop can invoke threads to handle intense processes, such as scene rendering and image manipulation. Suddenly, that dual-G5 Mac is able to process a Photoshop image in four seconds, whereas it took 45 seconds on your G3 Mac! Why? Threads.

However, just because a single program is able to share four CPUs, that doesn't mean each thread is an independent entity. Any global variables in the program (main thread) can be used by the invoked threads as long as care is taken that data is not damaged. Imagine 10 children grasping for an ice cream cone at the same time and you get the picture. What your threaded program must do is isolate the ice cream cone for each child, and only make the ice cream cone available to the others after that child has released it. Get the picture?

How does this concept of threading relate to processes? As you know, modern operating systems treat each program as a separate process, allocating a certain number of milliseconds to each process. This is where you get the term *multi-tasking*; many processes can be run at the same time using a time-slicing mechanism. A process has its own separate heap and stack and can contain many threads. A thread, on the other hand, has its own stack but shares the heap with other threads within the process. This is called a *thread group*.

The Pthreads-Win32 Library

The vast majority of Linux and UNIX operating system flavors will already have the pthread library installed because it is a core feature of the kernel. Other systems might not be so lucky. Windows uses its own multi-threading library. Of course, a primary goal of this book is to keep this code 100-percent portable. So what you need is a pthread library that is compatible with the POSIX systems. After all, that is what the "p" in pthreads stands for—POSIX threads.

An important thing you should know about the Windows implementation of pthread is that it abstracts the Windows threading functionality, molding it to conform to pthread standards.

There is one excellent open-source pthreads library for Windows systems, distributed by Red Hat, that I have chosen for this chapter because it includes makefiles for Visual C++ and Dev-C++. I have included the compiled version of pthread for Visual C++ and Dev-C++ on the CD-ROM in the \pthread folder, as Table 11.1 shows. These files are also provided in the MultiThread project folder on the CD-ROM. I recommend copying the lib file to your compiler's lib folder (for Visual C++ 6, this will usually be C:\Program Files\Microsoft Visual Studio\VC98\Lib) and the header files (pthread.h and sched.h) to your compiler's include folder (for Visual C++ 6, this will usually be C:\Program Files\Microsoft Visual Studio\VC98\Include). The dll can reside with the executable.

Table 11.1 pthread Library Files

Compiler	Lib	DLL
Visual C++	pthreadVC.lib	pthreadVC.dll
Dev-C++	libpthreadGC.a	pthreadGC.dll

Although Red Hat's pthread library is open source, I have chosen not to distribute it with the book and have only included the libs, dlls, and key headers. You can download the pthread library and find extensive documentation at http://sources.redhat.com/pthreads-win32. I encourage you to browse the site and get the latest version of Pthreads-Win32 from Red Hat. Makefiles are provided so it is easy to make the pthread library using whatever recent version of the sources you have downloaded. If you are intimidated by the prospect of having to compile sources, I encourage you to try. I, too, was once intimidated by downloading open source projects; I wasn't sure what to do with all the files. These packages were designed to be easy to make using GCC or Visual C++. All you really need to do is open a command prompt, change to the folder where the source code files are located, and set the path to your compiler. If you are using Dev-C++, for instance, you can type the following command to bring the GCC compiler online.

```
path=C:\Dev-Cpp\bin;%path%
```

What next? Simply type `make GC` and presto, the sources will be compiled. You'll have the libpthreadGC.a and pthreadGC.dll files after it's finished. The GC option is a parameter used by the makefile. If you want to see the available options, simply type `make` and the options will be displayed.

If you are really interested in this subject and you want more in-depth information, look for Butenhof's *Programming with POSIX Threads* (Addison-Wesley, 1997). Because the Pthreads-Win32 library is functionally compatible with Posix threads, the information in this book can be applied to pthread programming under Windows.

Programming with Posix Threads

I am going to cover the key functions in this section and let you pursue the full extent of multi-threaded programming on your own using the references I have suggested. For the purposes of this chapter, I want you to be able to control sprites using threads outside the main loop. Incidentally, the `main` function in any Allegro program is a thread too, although it is only a single thread. If you create an additional thread, then your program will be using two threads.

Creating a New Thread

First of all, how do you create a new thread? New threads are created with the `pthread_create` function.

```
int pthread_create (
    pthread_t *tid,
    const pthread_attr_t *attr,
    void *(*start) (void *),
    void *arg);
```

Yeah! That's what I thought at first, but it's not a problem. Here, let me explain. The first parameter is a `pthread_t` struct variable. This struct is large and complex, and you really don't need to know about the internals to use it. If you want more details, I encourage you to pick up Butenhof's book as a reference.

The second parameter is a `pthread_attr_t` struct variable that usually contains attributes for the new thread. This is usually not used, so you can pass NULL to it.

The third parameter is a pointer to the thread function used by this thread for processing. This function should contain its own loop, but should have exit logic for the loop when it's time to kill the thread. (I used a `done` variable.)

The fourth parameter is a pointer to a numeric value for this thread to uniquely identify it. You can just create an `int` variable and set it to a value before passing it to `pthread_create`.

Here's an example of how to create a new thread:

```
int id;
pthread_t pthread0;
int threadid0 = 0;
id = pthread_create(&pthread0, NULL, thread0, (void*)&threadid0);
```

So you've created this thread, but what about the callback function? Oh, right. Here's a minimal example:

```
void* thread0(void* data)
{
    int my_thread_id = *((int*)data);
    while(!done)
    {
        //do something!
    }
    pthread_exit(NULL);
    return NULL;
}
```

Killing a Thread

This brings us to the `pthread_exit` function, which terminates the thread. Normally you'll want to call this function at the end of the function after the loop has exited. Here's the definition for the function:

```
void pthread_exit (void *value_ptr);
```

You can get away with just passing NULL to this function because `value_ptr` is an advanced topic for gaining more control over the thread.

Mutexes: Protecting Data from Threads

At this point you can write a multi-threaded program with only the `pthread_create` and `pthread_exit` functions, knowing how to create the callback function and use it. That is enough if you only want to create a single thread to run inside the process with your program's main thread. But more often than not, you will want to use two or more threads in a game to delegate some of the workload. Therefore, it's a good idea to use a mutex for all your threads. Recall the ice cream cone analogy. Are you sure that new thread won't interfere with any globals? Have you considered timing? When you call `rest` to slow down the main loop, it has absolutely no effect on other threads. Each thread can call `rest` for timing independently of the others. What if you are using a thread to blit the double-buffer to the screen while another thread is writing to the buffer? Most memory chips cannot read and write data at the same time. It is very likely is that you'll update a small portion of the buffer (by drawing a sprite, for instance) while the buffer is being blitted to the screen. The result is some unwanted flicker—yes, even when using a double-buffer. What you have here is a situation that is similar to a vertical refresh conflict, only it is occurring in memory rather than directly on the screen. Do you need a `dbsync` type of function that is similar to `vsync`? I wouldn't go that far. What I am trying to point out is that threads can step on each other's toes, so to speak, if you aren't careful to use a mutex.

A *mutex* is a block used in a thread function to prevent other threads from running until that block is released. Assuming, of course, that all threads use the same mutex, it is possible to use more than one mutex in your program. The easiest way is to create a single mutex, and then block the mutex at the start of each thread's loop, unblocking at the end of the loop.

Creating a mutex doesn't require a function; rather, it requires a struct variable.

```
//create a new thread mutex to protect variables
pthread_mutex_t threadsafe = PTHREAD_MUTEX_INITIALIZER;
```

This line of code will create a new mutex called `threadsafe` that, when used by all the thread functions, will prevent data read/write conflicts.

You must destroy the mutex before your program ends; you can do so using the `pthread_ mutex_destroy` function.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Here is an example of how it would be used:

```
pthread_mutex_destroy(&threadsafe);
```

Next, you need to know how to lock and unlock a mutex inside a thread function. The `pthread_mutex_lock` function is used to lock a mutex.

```
int pthread_mutex_lock (pthread_mutex_t * mutex);
```

This has the effect of preventing any other threads from locking the same mutex, so any variables or functions you use or call (respectively) while the mutex is locked will be safe from manipulation by any other threads. Basically, when a thread encounters a locked mutex, it waits until the mutex is available before proceeding. (It uses no processor time; it simply waits.)

Here is the unlock function:

```
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

The two functions just shown will normally return zero if the lock or unlock succeeded immediately; otherwise, a non-zero value will be returned to indicate that the thread is waiting for the mutex. This should not happen for unlocking, only for locking. If you have a problem with `pthread_mutex_unlock` returning non-zero, it means the mutex was locked while that thread was supposedly in control over the mutex—a bad situation that should never happen. But when it comes to game programming, bad things do often happen while you are developing a new game, so it's helpful to print an error message for any non-zero return.

The MultiThread Program

At this point, you have all the information you need to use multi-threading in your own games and other programs. To test this program in a true parallel environment, I used my dual Athlon MP 1.2-GHz system under Windows 2000 and also under Windows XP. I like how XP is more thread-friendly (the Task Manager shows the number of threads used by each program), but any single-processor system will run this program just fine. Most dual systems should blow away even high-end single systems with this simple sprite demo because each sprite has its own thread. I have seen rates on my dual Athlon MP system that far exceed a much faster Pentium 4 system, but all that has changed with Intel's Hyper-Threading technology built into their high-end CPUs. This essentially means that Intel CPUs are thread-friendly and able to handle multiple threads in a single CPU.

Processors have boasted multiple pipelines for a decade, but now those pipelines are optimized to handle multiple threads.

The *MultiThread* program (shown in Figure 11.4) creates two threads (`thread0` and `thread1`) with similar functionality. Each thread moves a sprite on the screen with a bounce behavior, with full control over erasing, moving, and drawing the sprite on the double-buffer. This leaves the program's main loop with just a single task of blitting the buffer to the screen.

Figure 11.4 The *MultiThread* program uses threads to control sprite animation on the screen.

If you are using Visual C++, you'll want to create a new Win32 Application project, add a new source code file called main.c to the project, and then open the Project Settings dialog box, as shown in Figure 11.5.

On the Link tab, you'll want to type in **alleg.lib** and **pthreadVC.lib** separated by a space in the Object/Library Modules field, like this:

```
alleg.lib pthreadVC.lib
```

If you are using Dev-C++, you'll want to create a new Windows Application C-language project. Open the Project Options dialog box, go to the Parameters tab, and add the following two options:

```
-lalleg -lpthreadGC
```

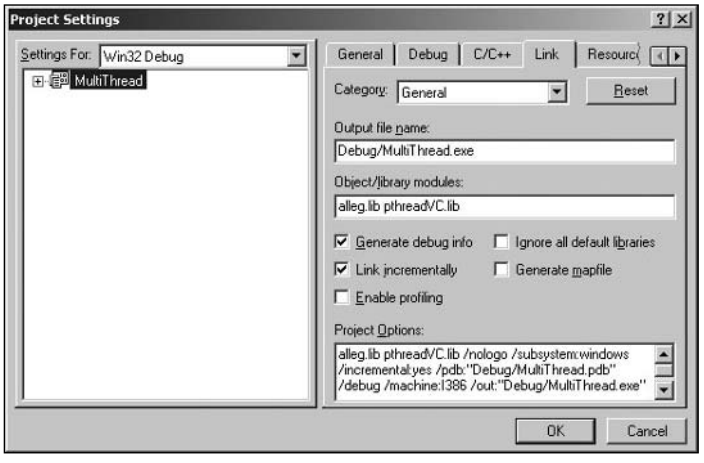


Figure 11.5 Adding pthreadVC.lib as a library file required by *MultiThread* program

Now you are ready to type in the source code for the *MultiThread* program. This project uses the sphere.bmp image containing the 32-frame animated ball from the *CollisionTest* project in Chapter 9. The project is located in completed form in the \chapter11\multi-thread directory on the CD-ROM. Here is the first section of code for the program:

```
#include <pthread.h>
#include "allegro.h"

#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define BLACK makecol(0,0,0)
#define WHITE makecol(255,255,255)

//define the sprite structure
typedef struct SPRITE
{
    int dir, alive;
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;
```

```
//variables
BITMAP *buffer;
BITMAP *ballimg[32];
SPRITE theballs[2];
SPRITE *balls[2];
int done;
int n;

//create a new thread mutex to protect variables
pthread_mutex_t threadsafe = PTHREAD_MUTEX_INITIALIZER;
```

As you can see, you just created the new mutex as a struct variable. Really, there is no processing done on a mutex at the time of creation; it is just a value that threads recognize when you pass `&threadsafe` to the `pthread_mutex_lock` and `pthread_mutex_unlock` functions.

The next section of code in the *MultiThread* program includes the usual sprite-handling functions that you should recognize.

```
void erasesprite(BITMAP *dest, SPRITE *spr)
{
    //erase the sprite
    rectfill(dest, spr->x, spr->y, spr->x + spr->width,
        spr->y + spr->height, BLACK);
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
```

```
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

//this version doesn't change speed, just direction
void bouncesprite(SPRITE *spr)
{
    //simple screen bouncing behavior
    if (spr->x < 0)
    {
        spr->x = 0;
        spr->xspeed = -spr->xspeed;
        spr->animdir *= -1;
    }

    else if (spr->x > SCREEN_W - spr->width)
    {
        spr->x = SCREEN_W - spr->width;
        spr->xspeed = -spr->xspeed;
        spr->animdir *= -1;
    }

    if (spr->y < 0)
    {
        spr->y = 0;
        spr->yspeed = -spr->yspeed;
        spr->animdir *= -1;
    }

    else if (spr->y > SCREEN_H - spr->height)
    {
        spr->y = SCREEN_H - spr->height;
        spr->yspeed = -spr->yspeed;
```

```
            spr->animdir *= -1;
        }
    }

    BITMAP *grabframe(BITMAP *source,
                      int width, int height,
                      int startx, int starty,
                      int columns, int frame)
    {
        BITMAP *temp = create_bitmap(width,height);
        int x = startx + (frame % columns) * width;
        int y = starty + (frame / columns) * height;
        blit(source,temp,x,y,0,0,width,height);
        return temp;
    }

    void loadsprites()
    {
        BITMAP *temp;

        //load sprite images
        temp = load_bitmap("sphere.bmp", NULL);
        for (n=0; n<32; n++)
            ballimg[n] = grabframe(temp,64,64,0,0,8,n);
        destroy_bitmap(temp);

        //initialize the sprite
        for (n=0; n<2; n++)
        {
            balls[n] = &theballs[n];
            balls[n]->x = rand() % (SCREEN_W - ballimg[0]->w);
            balls[n]->y = rand() % (SCREEN_H - ballimg[0]->h);
            balls[n]->width = ballimg[0]->w;
            balls[n]->height = ballimg[0]->h;
            balls[n]->xdelay = 0;
            balls[n]->ydelay = 0;
            balls[n]->xcount = 0;
            balls[n]->ycount = 0;
            balls[n]->xspeed = 5;
            balls[n]->yspeed = 5;
            balls[n]->curframe = rand() % 32;
            balls[n]->maxframe = 31;
```

```
        balls[n]->framecount = 0;
        balls[n]->framedelay = 0;
        balls[n]->animdir = 1;
    }
}
```

Now you come to the first thread callback function, `thread0`. I should point out that you can use a single callback function for all of your threads if you want. You can identify the thread by the parameter passed to it, which is retrieved into `my_thread_id` in the function listing that follows. You will want to pay particular attention to the calls to `pthread_mutex_lock` and `pthread_mutex_unlock` to see how they work. Note that these functions are called in pairs above and below the main piece of code inside the loop. Note also that `pthread_exit` is called after the loop. You should always provide a way to exit the loop, so this function can be called before the program ends. More than likely, all threads will terminate with the main process, but it is good programming practice to free memory before exiting.

```
//this thread updates sprite 0
void* thread0(void* data)
{
    //get this thread id
    int my_thread_id = *((int*)data);

    //thread's main loop
    while(!done)
    {
        //lock the mutex to protect variables
        if (pthread_mutex_lock(&threadsafe))
            textout(buffer,font,"ERROR: thread mutex was locked",
                0,0,WHITE);

        //erase sprite 0
        erasesprite(buffer, balls[0]);

        //update sprite 0
        updatesprite(balls[0]);

        //bounce sprite 0
        bouncesprite(balls[0]);

        //draw sprite 0
        draw_sprite(buffer, ballimg[balls[0]->curframe],
            balls[0]->x, balls[0]->y);
```

```
        //print sprite number
        textout(buffer, font, "0", balls[0]->x, balls[0]->y,WHITE);

        //display sprite position
        textprintf(buffer,font,0,10,WHITE,
            "THREAD ID %d, SPRITE (%3d,%3d)",
            my_thread_id, balls[0]->x, balls[0]->y);

        //unlock the mutex
        if (pthread_mutex_unlock(&threadsafe))
            textout(buffer,font,"ERROR: thread mutex unlock error",
                0,0,WHITE);

        //slow down (this thread only!)
     rest(10);
    }

    // terminate the thread
    pthread_exit(NULL);

    return NULL;
}
```

The second thread callback function, `thread1`, is functionally equivalent to the previous thread function. In fact, these two functions could have been combined and could have used `my_thread_id` to determine which sprite to update. This is something you should keep in mind if you want to add more sprites to the program to see what it can do. I separated the functions in this way to better illustrate what is happening. Just remember that many threads can share a single callback function, and that function is executed inside each thread separately.

```
//this thread updates sprite 1
void* thread1(void* data)
{
    //get this thread id
    int my_thread_id = *((int*)data);

    //thread's main loop
    while(!done)
    {
        //lock the mutex to protect variables
        if (pthread_mutex_lock(&threadsafe))
            textout(buffer,font,"ERROR: thread mutex was locked",
                0,0,WHITE);
```

```
        //erase sprite 1
        erasesprite(buffer, balls[1]);

        //update sprite 1
        updatesprite(balls[1]);

        //bounce sprite 1
        bouncesprite(balls[1]);

        //draw sprite 1
        draw_sprite(buffer, ballimg[balls[1]->curframe],
            balls[1]->x, balls[1]->y);

        //print sprite number
        textout(buffer, font, "1", balls[1]->x, balls[1]->y,WHITE);

        //display sprite position
        textprintf(buffer,font,0,20,WHITE,
            "THREAD ID %d, SPRITE (%3d,%3d)",
            my_thread_id, balls[1]->x, balls[1]->y);

        //unlock the mutex
        if (pthread_mutex_unlock(&threadsafe))
            textout(buffer,font,"ERROR: thread mutex unlock error",
                0,0,WHITE);

        //slow down (this thread only!)
     rest(20);
    }

    // terminate the thread
    pthread_exit(NULL);

    return NULL;
}
```

The final section of code for the *MultiThread* program contains the `main` function of the program, which creates the threads and processes the main loop to update the screen. Note that I have used the mutex in the main loop as well, just to be safe. You wouldn't want the double-buffer to get hit by multiple threads at the same time, which is what would happen without the mutex being called. Of course, that doesn't stop the main loop from impacting the buffer while a thread is using it. That is a situation you would want to take into account in a real game.

```
//program's primary thread
void main(void)
{
    int id;
    pthread_t pthread0;
    pthread_t pthread1;
    int threadid0 = 0;
    int threadid1 = 1;

    //initialize
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    srand(time(NULL));
    install_keyboard();
    install_timer();

    //create double buffer
    buffer = create_bitmap(SCREEN_W,SCREEN_H);

    //load ball sprite
    loadsprites();

    //create the thread for sprite 0
    id = pthread_create(&pthread0, NULL, thread0, (void*)&threadid0);

    //create the thread for sprite 1
    id = pthread_create(&pthread1, NULL, thread1, (void*)&threadid1);

    //main loop
    while (!key[KEY_ESC])
    {
        //lock the mutex to protect double buffer
        pthread_mutex_lock(&threadsafe);

        //display title
        textout(buffer, font, "MultiThread Program (ESC to quit)",
            0, 0, WHITE);

        //update the screen
        acquire_screen();
        blit(buffer,screen,0,0,0,0,SCREEN_W-1,SCREEN_H-1);
        release_screen();
```

```
        //unlock the mutex
        pthread_mutex_unlock(&threadsafe);

        //note there is no delay
    }

    //tell threads it's time to quit
    done++;
    rest(100);

    //kill the mutex (thread protection)
    pthread_mutex_destroy(&threadsafe);

    //remove objects from memory
    destroy_bitmap(buffer);

    //delete sprites
    for (n=0; n<32; n++)
        destroy_bitmap(ballimg[n]);

    return;
}

END_OF_MAIN();
```

Enhancing Tank War

The current version of *Tank War* (from Chapter 10) includes two scrolling windows (one for each player), a radar screen, tank sprites, bullet sprites, and scorekeeping. The game needs a few more things to make it complete. First of all, it needs better timing, particularly for explosions (which momentarily pause the game), and it could use a little more animation. In Chapter 15, "Mastering the Audible Realm: Allegro's Sound Support," you'll add sound support to the game.

For the time being, you can work on adding some better animation, as well as on that terrible explosion code that pauses the game. I'd like the explosions to be drawn on the screen without affecting the timing of the game. As for the new animation, I'd like the tank treads to move with respect to the speed that the tank is moving. So let's work on the sixth enhancement to the game now!

Description of New Improvements

To draw animated treads, I have modified the tank1.bmp and tank2.bmp files, adding seven additional frames to each tank from Ari Feldman's SpriteLib (from which the tanks were originally derived). Figure 11.6 shows the updated tank bitmaps.

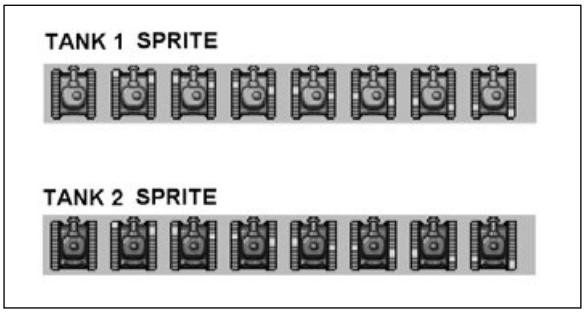


Figure 11.6 *Tank War* now features animated tanks.

To plug these new animated tanks into the game, you'll need to make some modifications to the routines that load, move, and draw the tanks, and you'll need to add a new function to animate the tanks. Figure 11.7 shows the game running with the animated tanks.

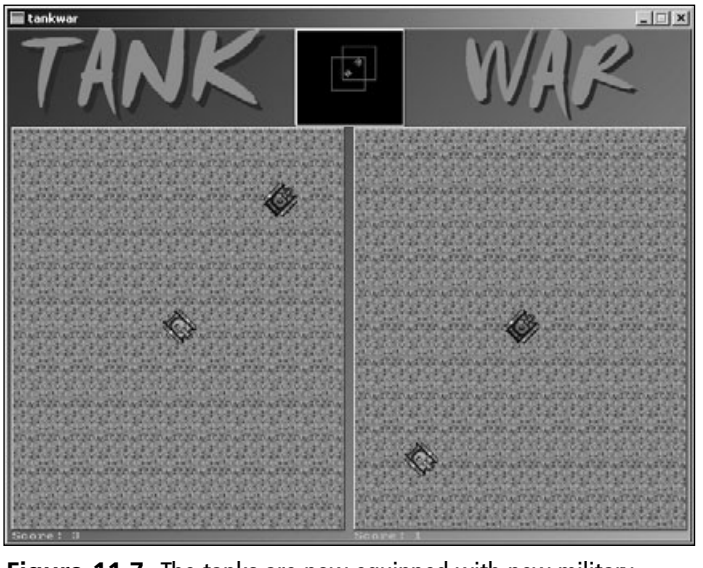


Figure 11.7 The tanks are now equipped with new military technology—animated treads.

The next enhancement to *Tank War* that I'll show you is an update to the `explode` function and addition of some new explosion sprites to handle the explosions so the game won't pause to render them. Figure 11.8 shows an explosion drawn over one of the tanks without pausing gameplay. Now both explosions can occur at the same time (instead of one after the other).

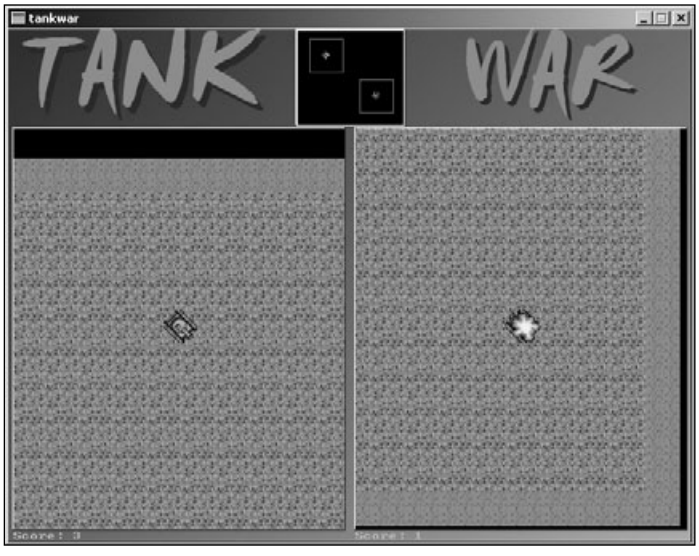


Figure 11.8 *Tank War* now draws animated explosions in the game loop without pausing the game.

Modifying the Tank War Project

The complete new version of *Tank War* is available in \chapter11\tankwar on the CD-ROM; you can load up the project or simply run the game from that location if you want. I recommend you follow along and make the changes yourself because it is a valuable learning experience. To do so, you'll want to open the *Tank War* project from Chapter 10 to make the following changes. Be sure to copy the tank1.bmp and tank2.bmp files off the CD-ROM so the new version of the game will work, because these bitmap files contain the new animated tanks.

Updating tankwar.h

First, you need to make a few minor changes to the tankwar.h header file. Look for the section of code that defines the sprites and add the new line of code shown in bold.

```
SPRITE mytanks[2];
SPRITE *tanks[2];
SPRITE mybullets[2];
SPRITE *bullets[2];
SPRITE *explosions[2];
```

Next, modify the `tank_bmp` array, which contains the bitmap images for the tanks. Scroll down in tankwar.h a little further to find the sprite bitmap definitions and make the change noted in bold. (It's a small change to the `tank_bmp` array—just add another dimension to the array as shown.)

```
//sprite bitmaps
BITMAP *tank_bmp[2][8][8];
BITMAP *bullet_bmp;
BITMAP *explode_bmp;
```

Now scroll down a little further in tankwar.h to the function prototypes and add the following three function definitions noted in bold:

```
//function prototypes
void animatetank(int num);
void updateexplosion(int num);
void loadsprites();
void drawtank(int num);
void erasetank(int num);
void movetank(int num);
```

Updating tank.c

Now you can make some changes to the tank.c source code file, which contains all the code for loading, moving, and drawing the tanks. Add a new function to the top of tank.c to accommodate the new animated tanks.

```
//new function added in chapter 11
void animatetank(int num)
{
    if (++tanks[num]->framecount > tanks[num]->framedelay)
    {
        tanks[num]->framecount = 0;
        tanks[num]->curframe += tanks[num]->animdir;
        if (tanks[num]->curframe > tanks[num]->maxframe)
            tanks[num]->curframe = 0;
        else if (tanks[num]->curframe < 0)
            tanks[num]->curframe = tanks[num]->maxframe;
    }
}
```

Now you have to make some changes to `drawtank`, the most important function in tank.c, because it is responsible for actually drawing the tanks. You need to add support for the new animated frames in the `tank_bmp` array. Make the changes noted in bold. (You'll notice that the only changes are made to `draw_sprite` function calls.)

```
void drawtank(int num)
{
    int dir = tanks[num]->dir;
    int x = tanks[num]->x-15;
    int y = tanks[num]->y-15;
```

```
    draw_sprite(buffer, tank_bmp[num][tanks[num]->curframe][dir], x, y);

    //what about the enemy tank?
    x = scrollx[!num] + SCROLLW/2;
    y = scrolly[!num] + SCROLLH/2;
    if (inside(x, y,
        scrollx[num], scrolly[num],
        scrollx[num] + SCROLLW, scrolly[num] + SCROLLH))
    {
        //draw enemy tank, adjust for scroll
        draw_sprite(buffer, tank_bmp[!num][tanks[!num]->curframe][tanks[!num]->dir],
            startx[num]+x-scrollx[num]-15, starty[num]+y-scrolly[num]-15);

    }
}
```

Next, you need to make some changes to the `movetank` function to accommodate the new animated tanks. The way this works now is that the tank is animated only when it is moving. You need to determine when the tank is moving by looking at the speed of the tank, and then update the sprite frame accordingly. You also need to make some changes to the code that keeps the tanks inside the bounds of the map so that when a tank reaches the edge, it will stop animating. Make the changes noted in bold.

```
void movetank(int num)
{
    int dir = tanks[num]->dir;
    int speed = tanks[num]->xspeed;

    //animate tank when moving
    if (speed > 0)
    {
        tanks[num]->animdir = 1;
        tanks[num]->framedelay = MAXSPEED - speed;
    }
    else if (speed < 0)
    {
        tanks[num]->animdir = -1;
        tanks[num]->framedelay = MAXSPEED - abs(speed);
    }
    else
        tanks[num]->animdir = 0;

    //update tank position
    switch(dir)
```

```
{
    case 0:
        scrolly[num] -= speed;
        break;
    case 1:
        scrolly[num] -= speed;
        scrollx[num] += speed;
        break;
    case 2:
        scrollx[num] += speed;
        break;
    case 3:
        scrollx[num] += speed;
        scrolly[num] += speed;
        break;
    case 4:
        scrolly[num] += speed;
        break;
    case 5:
        scrolly[num] += speed;
        scrollx[num] -= speed;
        break;
    case 6:
        scrollx[num] -= speed;
        break;
    case 7:
        scrollx[num] -= speed;
        scrolly[num] -= speed;
        break;
}

//keep tank inside bounds
if (scrollx[num] < 0)
{
    scrollx[num] = 0;
    tanks[num]->xspeed = 0;
}
else if (scrollx[num] > scroll->w - SCROLLW)
{
    scrollx[num] = scroll->w - SCROLLW;
    tanks[num]->xspeed = 0;
}
if (scrolly[num] < 0)
```

```
    {
        scrolly[num] = 0;
        tanks[num]->xspeed = 0;
    }
    else if (scrolly[num] > scroll->h - SCROLLH)
    {
        scrolly[num] = scroll->h - SCROLLH;
        tanks[num]->xspeed = 0;
    }
}
```

That is the last change to tank.c. Now you can move on to the setup.c file.

Updating setup.c

You must make extensive changes to setup.c to load the new animation frames for the tanks and initialize the new explosion sprites. You'll end up with a new `loadsprites` function and a lot of changes to `setuptanks`. First, add the new `loadsprites` function to the top of the setup.c file. I won't use bold because you need to add the whole function to the program.

```
void loadsprites()
{
    //load explosion image
    if (explode_bmp == NULL)
    {
        explode_bmp = load_bitmap("explode.bmp", NULL);
    }

    //initialize explosion sprites
    explosions[0] = malloc(sizeof(SPRITE));
    explosions[1] = malloc(sizeof(SPRITE));
}
```

Next up, the changes to `setuptanks`. There are a lot of changes to be made in this function to load the new tank1.bmp and tank2.bmp files, and then extract the individual animation frames. Make all changes noted in bold.

```
void setuptanks()
{
    BITMAP *temp;
    int anim;
    int n;

    //configure player 1's tank
    tanks[0] = &mytanks[0];
```

```
tanks[0]->x = 30;
tanks[0]->y = 40;
tanks[0]->xspeed = 0;
tanks[0]->dir = 3;
tanks[0]->curframe = 0;
tanks[0]->maxframe = 7;
tanks[0]->framecount = 0;
tanks[0]->framedelay = 10;
tanks[0]->animdir = 0;
scores[0] = 0;

//load first tank
temp = load_bitmap("tank1.bmp", NULL);
for (anim=0; anim<8; anim++)
{
    //grab animation frame
    tank_bmp[0][anim][0] = grabframe(temp, 32, 32, 0, 0, 8, anim);

    //rotate image to generate all 8 directions
    for (n=1; n<8; n++)
    {
        tank_bmp[0][anim][n] = create_bitmap(32, 32);
        clear_to_color(tank_bmp[0][anim][n], makecol(255,0,255));
        rotate_sprite(tank_bmp[0][anim][n], tank_bmp[0][anim][0],
            0, 0, itofix(n*32));
    }
}
destroy_bitmap(temp);

//configure player 2's tank
tanks[1] = &mytanks[1];
tanks[1]->x = SCREEN_W-30;
tanks[1]->y = SCREEN_H-30;
tanks[1]->xspeed = 0;
tanks[1]->dir = 7;
tanks[1]->curframe = 0;
tanks[1]->maxframe = 7;
tanks[1]->framecount = 0;
tanks[1]->framedelay = 10;
tanks[1]->animdir = 0;
scores[1] = 0;
```

```
    //load second tank
    temp = load_bitmap("tank2.bmp", NULL);
    for (anim=0; anim<8; anim++)
    {
        //grab animation frame
        tank_bmp[1][anim][0] = grabframe(temp, 32, 32, 0, 0, 8, anim);

        //rotate image to generate all 8 directions
        for (n=1; n<8; n++)
        {
            tank_bmp[1][anim][n] = create_bitmap(32, 32);
            clear_to_color(tank_bmp[1][anim][n], makecol(255,0,255));
            rotate_sprite(tank_bmp[1][anim][n], tank_bmp[1][anim][0],
                0, 0, itofix(n*32));
        }
    }
    destroy_bitmap(temp);

    //load bullet image
    if (bullet_bmp == NULL)
        bullet_bmp = load_bitmap("bullet.bmp", NULL);

    //initialize bullets
    for (n=0; n<2; n++)
    {
       bullets[n] = &mybullets[n];
       bullets[n]->x = 0;
       bullets[n]->y = 0;
       bullets[n]->width = bullet_bmp->w;
       bullets[n]->height = bullet_bmp->h;
    }

    //center tanks inside scroll windows
    tanks[0]->x = 5 + SCROLLW/2;
    tanks[0]->y = 90 + SCROLLH/2;
    tanks[1]->x = 325 + SCROLLW/2;
    tanks[1]->y = 90 + SCROLLH/2;
}
```

That wasn't so bad because the game was designed well and the new code added in Chapter 10 was highly modifiable. It always pays to write clean, tight code right from the start.

Updating bullet.c

Now you can make the necessary changes to the bullet.c source file to accommodate the new friendly explosions. (How's that for a contradiction of terms?) What I mean by *friendly* is that the explosions will no longer use the `rest` function to draw. This is really bad because it causes the whole game to hiccup every time there is an explosion to be drawn. There weren't many bullets flying around in this game, or I never would have gotten away with this quick solution. Now let's correct the problem.

Open the bullet.c file. You'll be adding a new function called `updateexplosion` and modifying the existing `explode` function. Here is the new `updateexplosion` you should add to the top of the bullet.c file.

```
//new function added in chapter 11
void updateexplosion(int num)
{
    int x, y;

    if (!explosions[num]->alive) return;

    //draw explosion (maxframe) times
    if (explosions[num]->curframe++ < explosions[num]->maxframe)
    {
        x = explosions[num]->x;
        y = explosions[num]->y;

        //draw explosion in enemy window
        rotate_sprite(buffer, explode_bmp,
            x + rand()%10 - 20, y + rand()%10 - 20,
            itofix(rand()%255));

        //draw explosion in "my" window if enemy is visible
        x = scrollx[!num] + SCROLLW/2;
        y = scrolly[!num] + SCROLLH/2;
        if (inside(x, y,
            scrollx[num], scrolly[num],
            scrollx[num] + SCROLLW, scrolly[num] + SCROLLH))
        {
            //but only draw if explosion is active
            if (explosions[num]->alive)
                rotate_sprite(buffer, explode_bmp,
                    startx[num]+x-scrollx[num] + rand()%10 - 20,
                    starty[num]+y-scrolly[num] + rand()%10 - 20,
                    itofix(rand()%255));
```

```
        }
    }
    else
    {
        explosions[num]->alive = 0;
        explosions[num]->curframe = 0;
    }
}
```

Now modify `explode` so it will properly set up the explosion, which is actually drawn by `updateexplosion` later on in the animation process of the game loop. Make the changes noted in bold. The entire function has been rewritten, so simply delete existing code and add the new lines to `explode`.

```
void explode(int num, int x, int y)
{
    //initialize the explosion sprite
    explosions[num]->alive = 1;
    explosions[num]->x = x;
    explosions[num]->y = y;
    explosions[num]->curframe = 0;
    explosions[num]->maxframe = 20;
}
```

That's the end of the changes to bullet.c. Now you can make the last few changes needed to update the game. Next you'll turn to the main.c file.

Updating main.c

The last changes will be made to main.c to call the new functions (such as `animatetank` and `updateexplosion`). The only changes to be made will be to the `main` function. You need to add a line that creates a new variable and calls `loadsprites` and `animatetank`, and finally, you need a call to `updateexplosion`. Be careful to catch the changes to `tank_bmp` and note the cleanup code at the end. Make the changes noted in bold.

```
//main function
void main(void)
{
    int anim;

    //initialize the game
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
```

```
setupscreen();
setuptanks();
loadsprites();

//game loop
while(!gameover)
{
    //move the tanks and bullets
    for (n=0; n<2; n++)
    {
        movetank(n);
        animatetank(n);
        movebullet(n);
    }

    //draw background bitmap
    blit(back, buffer, 0, 0, 0, 0, back->w, back->h);

    //draw scrolling windows
    for (n=0; n<2; n++)
        blit(scroll, buffer, scrollx[n], scrolly[n],
            startx[n], starty[n], SCROLLW, SCROLLH);

    //update the radar
    rectfill(buffer,radarx+1,radary+1,radarx+99,radary+88,BLACK);
    rect(buffer,radarx,radary,radarx+100,radary+89,WHITE);

    //draw mini tanks on radar
    for (n=0; n<2; n++)
        stretch_sprite(buffer, tank_bmp[n][tanks[n]->curframe][tanks[n]->dir],
            radarx + scrollx[n]/10 + (SCROLLW/10)/2-4,
            radary + scrolly[n]/12 + (SCROLLH/12)/2-4,
            8, 8);

    //draw player viewport on radar
    for (n=0; n<2; n++)
        rect(buffer,radarx+scrollx[n]/10, radary+scrolly[n]/12,
            radarx+scrollx[n]/10+SCROLLW/10,
            radary+scrolly[n]/12+SCROLLH/12, GRAY);

    //display score
    for (n=0; n<2; n++)
```

```
            textprintf(buffer, font, startx[n], HEIGHT-10,
                BURST, "Score: %d", scores[n]);

        //draw the tanks and bullets
        for (n=0; n<2; n++)
        {
            drawtank(n);
            drawbullet(n);
        }

        //explosions come last (so they draw over tanks)
        for (n=0; n<2; n++)
            updateexplosion(n);

        //refresh the screen
        acquire_screen();
        blit(buffer, screen, 0, 0, 0, 0, WIDTH-1, HEIGHT-1);
        release_screen();

        //check for keypresses
        if (keypressed())
            getinput();

        //slow the game down
        rest(20);
    }

    //destroy bitmaps
    destroy_bitmap(explode_bmp);
    destroy_bitmap(back);
    destroy_bitmap(scroll);
    destroy_bitmap(buffer);

    //free tank bitmaps
    for (anim=0; anim<8; anim++)
        for (n=0; n<8; n++)
        {
            destroy_bitmap(tank_bmp[0][anim][n]);
            destroy_bitmap(tank_bmp[1][anim][n]);
        }
```

```
    //free explosion sprites
    for (n=0; n<2; n++)
        free(explosions[n]);

    return;
}
END_OF_MAIN();
```

Future Changes to Tank War

I must admit that this game is really starting to become fun, not only as a very playable game, but also as an Allegro game project. It is true that if you design and program a game that *you* find interesting and fun, others will be attracted to the game as well. I did just that, and I have enjoyed sharing the vision of this game with you. What do you think of the result so far? It needs a little bit more work (such as sound effects), but otherwise it is very playable. If you have any great ideas to make the game even better, by all means, go ahead and try them!

You can use this example game as a basis for your own games. Are you interested in RPGs? Go ahead and convert it to a single scrolling window and replace the tank with your own character sprite, and you almost have an RPG framework right there. As for future changes, the next chapter adds customizable levels to the game with a level-editing program called Mappy.

Summary

This was an advanced chapter that dealt with the intriguing subjects of timers, interrupts, and threads. I started with a *TimerTest* program that animated several sprites on the screen to demonstrate how to calculate and display the frame rate. You then modified the program to use an interrupt handler to keep track of the frame rate outside of the main loop (*InterruptTest*). This was followed by another revision that demonstrated how to set a specific frame rate for the program (*TimedLoop*). The last section of the chapter was devoted to multi-threading, with a tutorial on the Posix Threads library and Red Hat's Pthreads-Win32 project. The result was an interesting program called *MultiThread* that demonstrated how to use threads for sprite control. The potential for increased frame-rate performance in a game is greatly encouraged with the use of threads to delegate functionality from a single loop because this provides support for multiple-processor systems.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Why is it important to use a timer in a game?
   A. To maintain a consistent frame rate
   B. To include support for interrupts
   C. To make the program thread-safe
   D. To delegate code to multiple threads

2. Which Allegro timer function slows down the program using a callback function?
   A. `callback_rest`
   B. `sleep_callback`
   C. `rest`
   D. `rest_callback`

3. What is the name of the function used to initialize the Allegro timer?
   A. `init_timer`
   B. `install_timer`
   C. `timer_reset`
   D. `start_timer`

4. What is the name of the function that creates a new interrupt handler?
   A. `create_handler`
   B. `create_interrupt`
   C. `int_callback`
   D. `install_int`

5. What variable declaration keyword should be used with interrupt variables?
   A. `danger`
   B. `cautious`
   C. `volatile`
   D. `corruptible`

6. What is a process that runs within the memory space of a single program but is executed separately from that program?
   A. Mutex
   B. Process
   C. Thread
   D. Interrupt

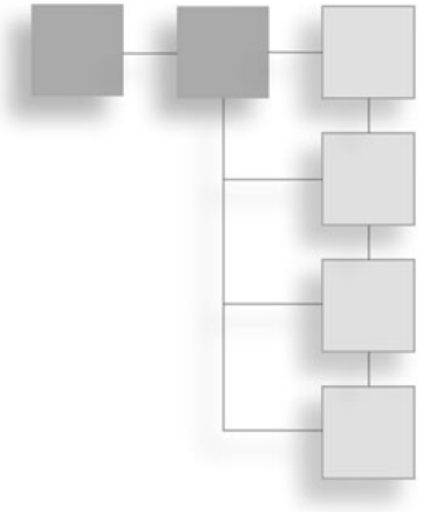
7. What helps protect data by locking it inside a single thread, preventing that data from being used by another thread until it is unlocked?
   A. Mutex
   B. Process
   C. Thread
   D. Interrupt

8. What does pthread stand for?
   A. Protected Thread
   B. Public Thread
   C. Posix Thread
   D. Purple Thread

9. What is the name of the function used to create a new thread?
   A. `create_posix_thread`
   B. `pthread_create`
   C. `install_thread`
   D. `thread_callback`

10. What is the name of the function that locks a mutex?
    A. `lock_pthread_mutex`
    B. `lock_mutex`
    C. `pthread_lock_mutex`
    D. `pthread_mutex_lock`

CHAPTER 12



Creating a Game World: Editing Tiles and Levels

The game world defines the rules of the game and presents the player with all of the obstacles he must overcome to complete the game. Although the game world is the most important aspect of a game, it is not always given proper attention when a game is being designed. This chapter provides an introduction to world building—or more specifically, map editing. Using the skills you learn in this chapter, you will be able to enhance *Tank War* and learn to create levels for your own games. This chapter provides the prerequisite information you'll need in the next two chapters, which discuss horizontal and vertical scrolling games.

Here is a breakdown of the major topics in this chapter:

- Creating the game world
- Loading and drawing Mappy level files

Creating the Game World

Mappy is an awesome map editing program, and it's freeware so you can download and use it to create maps for your games at no cost. If you find Mappy to be as useful as I have, I encourage you to send the author a small donation to express your appreciation for his hard work. The home page for Mappy is http://www.tilemap.co.uk.

Why is Mappy so great, you might ask? First of all, it's easy to use. In fact, it couldn't be any easier to use without sacrificing features. Mappy allows you to edit maps made up of the standard rectangular tiles, as well as isometric and hexagonal tiles! Have you ever played hexagonal games, such as *Panzer General*, or isometric games, such as *Age of Empires*? Mappy lets you create levels that are similar to the ones used in these games. Mappy has been used to create many retail (commercial) games, some of which you might

have played. I personally know of several developers who have used Mappy to create levels for retail games for Pocket PC, Game Boy Advance, Nokia N-Gage, and wireless (cell phones). MonkeyStone's *Hyperspace Delivery Boy* (created by Tom Hall, John Romero, and Stevie Case) for Pocket PC and Game Boy Advance is one example.

Suffice it to say, Mappy is an unusually great map editor released as freeware, and I will explain how to use it in this chapter. You'll also have an opportunity to add Mappy support to *Tank War* at the end of the chapter.

Installing Mappy

Mappy is included in the \mappy folder on the CD-ROM that accompanies this book. You can run Mappy directly without installing it, although I would recommend copying the mapwin.exe file to your hard drive. Mappy is so small (514 KB) that it's not unreasonable to copy it to any folder where you might need it. If you want to check for a newer version of Mappy, the home page is located at http://www.tilemap.co.uk. In addition to Mappy, there are sample games available for download and the Allegro support sources for Mappy. (See the "Loading and Drawing Mappy Level Files" section later in this chapter for more information.) If you do copy the executable without the subfolders, INI file, and so on, you'll miss out on the Lua scripts and settings, so you might want to copy the whole folder containing the executable file.

Creating a New Map

Now it's time to fire up Mappy and create a new map. Locate mapwin.exe and run it. The first time it is run, Mappy displays two blank child windows (see Figure 12.1).

Now open the File menu and select New Map to bring up the New Map dialog box, shown in Figure 12.2.

As the New Map dialog box shows, you must enter the size of each tile in your tile image file. The tiles used in *Tank War* (and in most of the chapters of this book) are 32×32 pixels, so I have typed 32 in the width box and 32 in the height box. Next you must enter the size of the map. The default 100×100 map probably is too large to be useful as a good example at this point. If you recall from Chapter 10, the *GameWorld* program used a map that had an area of 31×33 tiles. You should use that program as a basis for testing Mappy. Of course, you can use any values you want, but be sure to modify the source code (in the next section) to accommodate the dimensions of the map you have created.

tip

Mappy allows you to change the size of the map after it has been created, so if you need more tiles in your map later, it's easy to enlarge the map. Likewise, you can shrink the map; Mappy has an option that lets you choose the part of the map you want to resize.

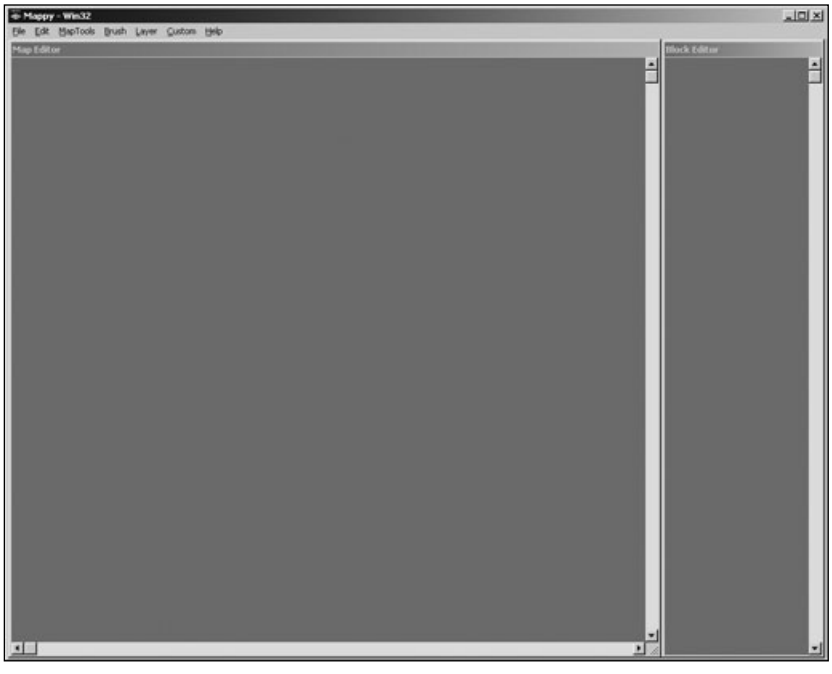


Figure 12.1 Mappy is a simple and unassuming map editor.

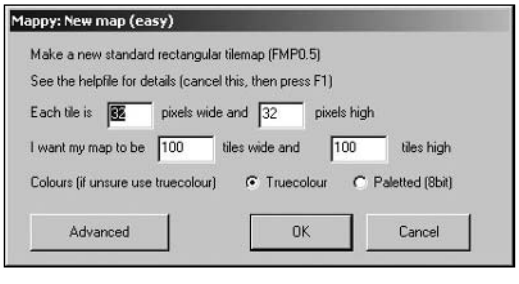


Figure 12.2 You can use the New Map dialog box to configure a new game level.

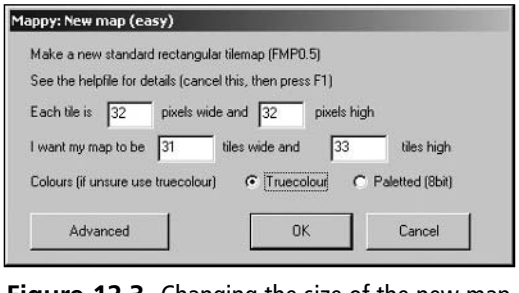


Figure 12.3 Changing the size of the new map

Figure 12.3 shows the dimensions that I have chosen for this new map. Note also the option for color depth. This refers to the source image containing the tiles; in most cases you will want to choose the Truecolour option because most source artwork will be 16-bit, 24-bit, or 32-bit. (Any of these will work with Mappy if you select this option.)

If you click on the Advanced button in the New Map dialog box, you'll see the additional options shown in Figure 12.4. These additional options allow you to select the exact color depth of the source tiles (8-bit through 32-bit), the map file version to use, and dimensions for non-rectangular map tiles (such as hexagonal and isometric).

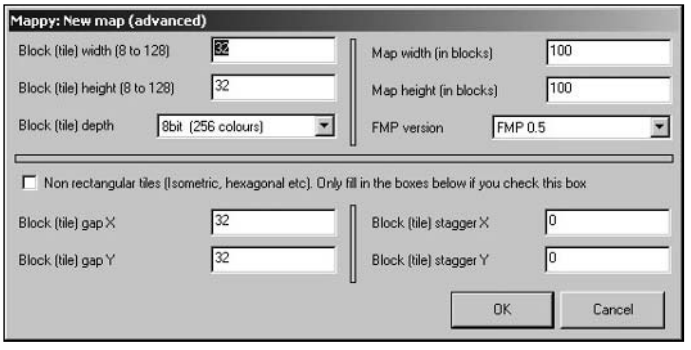


Figure 12.4 The advanced options in the New Map dialog box

When you click on the OK button, a new map will be created and filled with the default black tile (tile #0). At this point, you must import the tile images to be used to create this map. This is where things really get interesting because you can use multiple image files containing source artwork, and Mappy will combine all the source tiles into a new image source with correctly positioned tiles. (Saving the tile bitmap file is an option in the Save As dialog box.)

Importing the Source Tiles

Now open the File menu and select Import. The Open File dialog box will appear, allowing you to browse for an image file, which can be of type BMP, PCX, PNG, or MAR/P (map array file—something that can be exported by Mappy). I have created a larger tile image file containing numerous tiles from Ari Feldman's SpriteLib (http://www.arifeldman.com). The maptiles.bmp file is located in the \chapter12\ArrayMapTest folder on the CD-ROM. After you choose this file, Mappy will import the tiles into the tile palette, as shown in Figure 12.5. Recall that you specified the tile size when you created the map file; Mappy used the dimensions provided to automatically read in all of the tiles. You must make the image resolution reasonably close to the edges of the tiles, but it doesn't need to be perfect—Mappy is smart enough to account for a few pixels off the right or bottom edges and move to the next row.

Now I'd like to show you a convenient feature that I use often. I like to see most of the level on the screen at once to get an overview of the game level. Mappy lets you change the zoom level of the map editor display. Open the MapTools menu and select one of the zoom levels to change the zoom. Then, select a tile from the tile palette and use the mouse to draw that tile on the map edit window to see how the chosen zoom level appears. I frequently use 0.5 (1/2 zoom). Until you have added some tiles to the map window, you won't see anything happen after you change the zoom.

Now let me show you a quick shortcut for filling the entire map with a certain tile. Select a neutral tile that is good as a backdrop, such as the grass, dirt, or stone tile. Open the Custom menu. This menu contains scripts that you can run to manipulate a map. (You can write your own scripts if you learn the Lua language—visit http://www.lua.org for more information.) Select the script called Solid Rectangle, which brings up the dialog box shown in Figure 12.6.

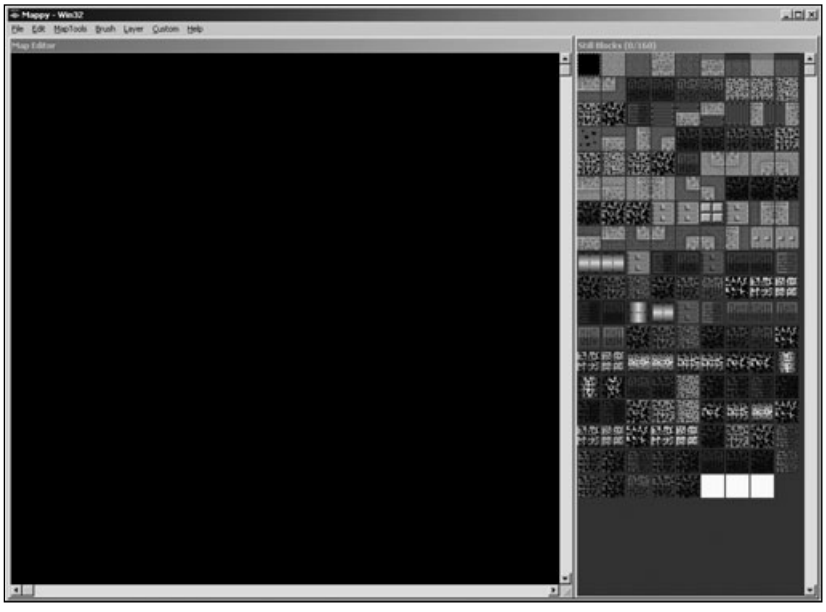


Figure 12.5 The SpriteLib tiles have been imported into Mappy's tile palette for use in creating game levels.

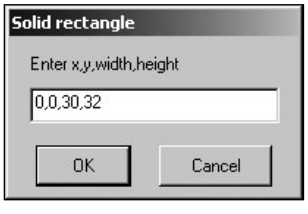


Figure 12.6 Mappy includes scripts that can manipulate a map, and you can create new scripts.

Modify the width and height parameters for the rectangle, using one less than the value you entered for the map when it was created (31–1 = 30 and 33–1 = 32). Click on OK, and the map will be filled with the currently selected tile, as shown in Figure 12.7.

Play around with Mappy to gain familiarity with it. You can erase tiles using the right mouse button and select tiles in the palette using the left button. You can use the keyboard arrow keys to scroll the map in any direction, which is very handy when you want to keep your other hand on the mouse for quick editing. Try to create an interesting map, and then I'll show you how to save the map in two different formats you'll use in the sample programs that follow.

Saving the Map File as FMP

Have you created an interesting map that can be saved? If not, go ahead and create a map, even if it's just a hodgepodge of tiles, because I want to show you how to save and use the map file in an Allegro program. Are you ready yet? Good! As a reference for the figures that follow in this chapter, the map I created is shown in Figure 12.8.

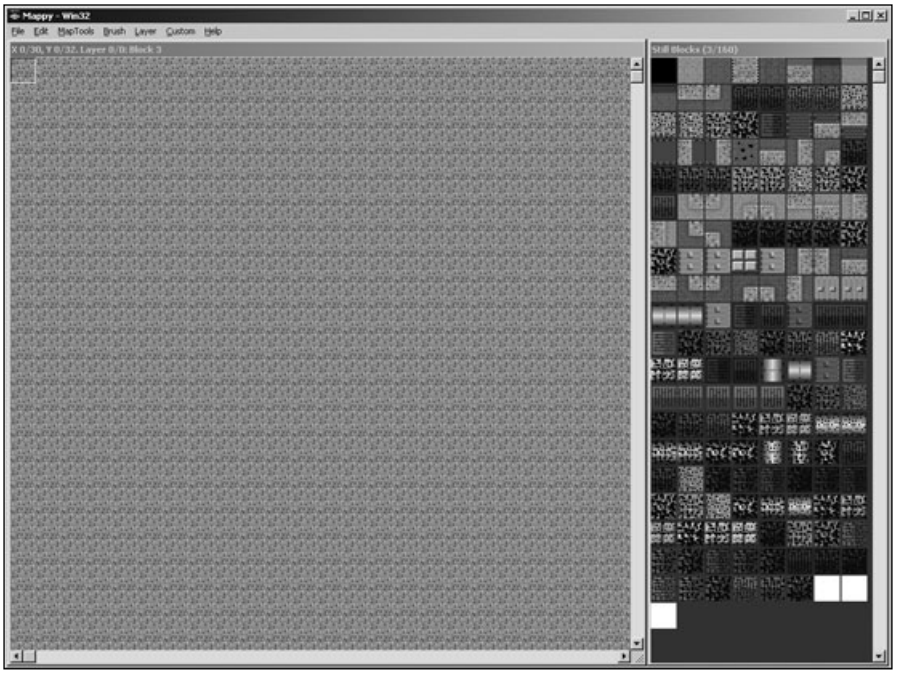


Figure 12.7 The Solid Rectangle script fills a region of the map with a tile.

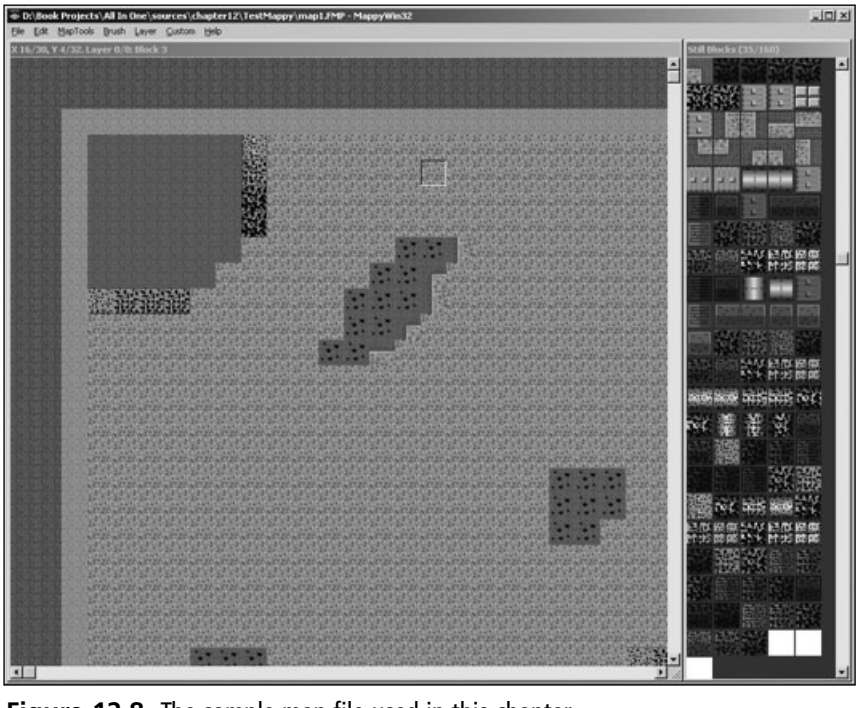


Figure 12.8 The sample map file used in this chapter

I'll show you how to save the map file first, and then you'll export the map to a text file and try to use it in sample programs later. For now, open the File menu and select Save As to bring up the Save As dialog box shown in Figure 12.9.

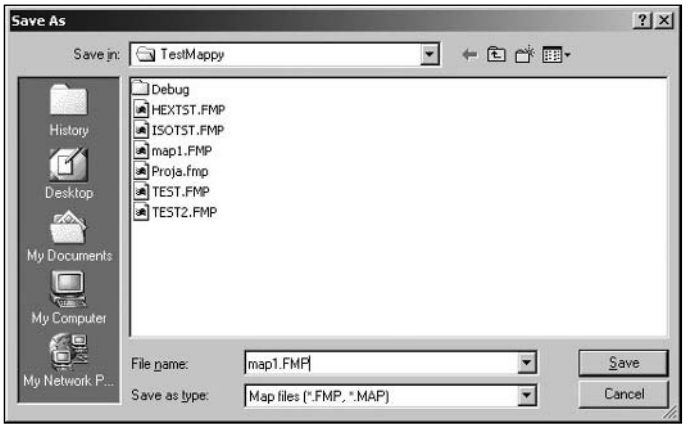


Figure 12.9 The Save As dialog box in Mappy is used to save a map file.

Type a map filename, such as map1.fmp, and click on Save. The interesting thing about the FMP file format is that the tile images are stored along with the map data, so you don't need to load the tiles *and* the map file to create your game world. You might not like losing control over the tile images, but in a way it's a blessing—one less thing to worry about when you'd rather focus your time on gameplay.

Saving the Map File as Text

Now that you have saved the new level in the standard Mappy file format, I'd like to show you how to export the map to a simple text file that you can paste into a program. The result will be similar to the *GameWorld* program from Chapter 10, in which the map tile data was stored in an array in the program's source code.

Open the File menu and select Export. Do *not* select Export As Text. That is an entirely different option used to export a map to a binary array used for the Game Boy Advance and other systems. Just select Export to bring up the Export dialog box shown in Figure 12.10.

You can explore the uses for the various formats in the Export dialog box when you have an opportunity; I will only explain the one option you need to export the map data as text. You want to select the third check box from the top, labeled Map Array as Comma Values Only (?.CSV).

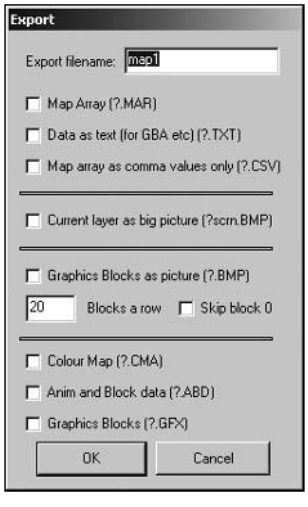


Figure 12.10 The Export dialog in Mappy lets you choose options for exporting the map.

If you want to build an image containing the tiles in the proper order, as they were in Mappy, you can also select the check box labeled Graphics Blocks as Picture (?.BMP). I strongly recommend exporting the image. For one thing, Mappy adds the blank tile that you might have used in some parts of the map; it also numbers the tiles consecutively starting with this blank tile unless you check the option Skip Block 0. Normally, you should be able to leave the default of 20 in the Blocks a Row input field. Click on OK to export the map.

Mappy outputs the map with the name provided in the Export dialog box as two files—map1.BMP and map1.CSV. (Your map name might differ.) The CSV format is recognized by Microsoft Excel, but there is no point loading it into Excel (even if you have Microsoft Office installed). Instead, rename the file map1.txt and open it in Notepad or another text editor. You can now copy the map data text and paste it into a source code file, and you have the bitmap image handy as well.

Loading and Drawing Mappy Level Files

Mappy is far more powerful than you need for *Tank War* (or the rest of this book, for that matter). Mappy supports eight-layer maps with animated tiles and has many helpful features for creating game worlds. You can create a map, for instance, with a background layer, a parallax scrolling layer with transparent tiles, and a surface layer that is drawn over sprites (such as bridges and tunnels). I'm sure you will become proficient with Mappy in a very short time after you use it to create a few games, and you will find some of these features useful in your own games. For the purposes of this chapter and the needs of your *Tank War* game, Mappy will be used to create a single-layer map.

There are two basic methods of using map files in your own games. The first method is to export the map from Mappy as a text file. You can then paste the comma-separated map tile numbers into an array in your game. (Recall the *GameWorld* program from Chapter 10, which used a hard-coded map array.) There are drawbacks to this method, of course. Any time you need to make changes to a map file, you'll need to export the map again and paste the lines of numbers into the map array definition in your game's source code. However, storing game levels (once completed) inside an array means that you don't need to load the map files into your game—and further, this prevents players from editing your map files. I'll explain how to store game resources (such as map files) inside an encrypted/compressed data file in Chapter 16, "Using Datafiles to Store Game Resources."

The other method, of course, is to load a Mappy level file into your game. This is a more versatile solution, which makes sense if your game has a lot of levels and/or is expandable. (Will players be able to add their own levels to the game and make them available for download, and will you release expansion packs for your game?)

The choice is obvious for large, complex games, such as *StarCraft*, but for smaller games like *Arkanoid*, my personal preference is to store game levels inside the source code. Given the advanced features in Mappy, it is really only practical to export map files if your game is using a single layer with no animation. When your game needs multiple layers and animated tiles, it is better to load the Mappy level file. Why? Because source code is available to load and draw complex Mappy files. (See the "Using a Mappy Level File" section later in this chapter.). Another consideration you should keep in mind is that Mappy files include both the map data *and* the artwork! That's right; the Mappy file includes the tiles as well as the data, so you don't need to load the tiles separately when you're using a Mappy file directly. This is a great feature, particularly when you are dealing with huge, complex game world maps.

Next, I'll demonstrate how to load a map that has been exported to a text file, and then I'll follow that explanation with another sample program that demonstrates how to load a Mappy file directly.

Using a Text Array Map

I want to write a short program to demonstrate how to load a Mappy level that has been exported to a text file. You'll recall from the previous section that you exported a map to a text file with a bitmap file filled with the source tiles that correspond to the values in the text data. I'm going to open the *GameWorld* program from Chapter 10 and modify it to demonstrate the text map data that was exported. Create a new project and add a reference to the Allegro library as usual. Then, type the following code into the main.c file. Figure 12.11 shows the program running.

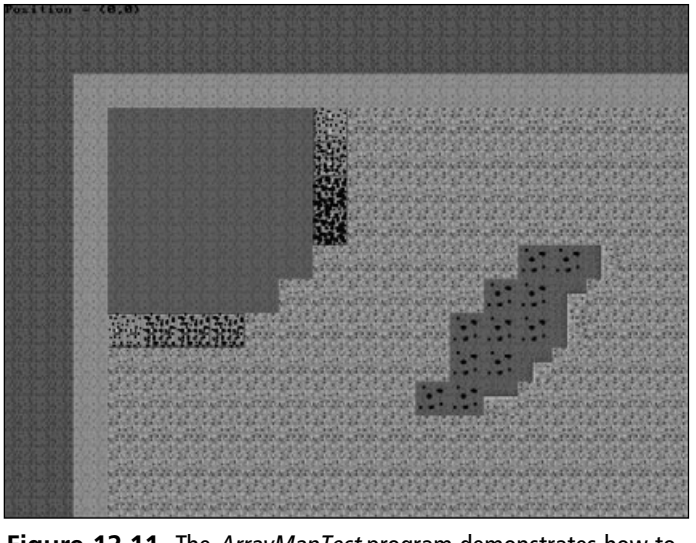


Figure 12.11 The *ArrayMapTest* program demonstrates how to use an exported Mappy level.

If you are using *GameWorld* as a basis, just take note of the differences. On the CD-ROM, this project is called ArrayMapTest, and it is located in the \chapter12\ArrayMapTest folder.

```
#include "allegro.h"

//define some convenient constants
#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define STEP 8

//very important! double check these values!
#define TILEW 32
#define TILEH 32

//20 columns across is the default for a bitmap
//file exported by Mappy
#define COLS 20

//make sure this exactly describes your map data
#define MAP_ACROSS 31
#define MAP_DOWN 33

#define MAPW MAP_ACROSS * TILEW
#define MAPH MAP_DOWN * TILEH

int map[] = {
    //
    //PASTE MAPPY EXPORTED TEXT DATA HERE!!!
    //
};

//temp bitmap
BITMAP *tiles;

//virtual background buffer
BITMAP *scroll;

//position variables
int x=0, y=0, n;
int tilex, tiley;
```

```
//reuse our friendly tile grabber from chapter 9
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);
    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;
    blit(source,temp,x,y,0,0,width,height);
    return temp;
}

//main function
void main(void)
{
    //initialize allegro
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    set_color_depth(16);

    //set video mode
    if (set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0) != 0)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(allegro_error);
        return;
    }

    //create the virtual background
    scroll = create_bitmap(MAPW, MAPH);
    if (scroll == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating virtual background");
        return;
    }

    //load the tile bitmap
    //note that it was renamed from chapter 10
```

```
tiles = load_bitmap("maptiles.bmp", NULL);
if (tiles == NULL)
{
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("Error loading maptiles.bmp file");
    return;
}

//now draw tiles on virtual background
for (tiley=0; tiley < scroll->h; tiley+=TILEH)
{
    for (tilex=0; tilex < scroll->w; tilex+=TILEW)
    {
        //use the result of grabframe directly in blitter
        //change: TILEW-1, TILEH-1 are just TILEW,TILEH now
        blit(grabframe(tiles, TILEW, TILEH, 0, 0, COLS, map[n++]),
            scroll, 0, 0, tilex, tiley, TILEW, TILEH);
    }
}

//main loop
while (!key[KEY_ESC])
{
    //check right arrow
    if (key[KEY_RIGHT])
    {
        x += STEP;
        if (x > scroll->w - WIDTH)
            x = scroll->w - WIDTH;
    }

    //check left arrow
    if (key[KEY_LEFT])
    {
        x -= STEP;
        if (x < 0)
            x = 0;
    }

    //check down arrow
    if (key[KEY_DOWN])
    {
```

```
            y += STEP;
            if (y > scroll->h - HEIGHT)
                y = scroll->h - HEIGHT;
        }

        //check up arrow
        if (key[KEY_UP])
        {
            y -= STEP;
            if (y < 0)
                y = 0;
        }

        //draw the scroll window portion of the virtual buffer
        blit(scroll, screen, x, y, 0, 0, WIDTH-1, HEIGHT-1);

        //display status info
        text_mode(-1);
        textprintf(screen,font,0,0,makecol(0,0,0),
            "Position = (%d,%d)", x, y);

        //slow it down
        rest(20);
    }

    destroy_bitmap(scroll);
    destroy_bitmap(tiles);
    return;
}

END_OF_MAIN();
```

In case you didn't catch the warning (with sirens, red alerts, and beseechings), you must paste your own map data into the source code in the location specified. The map data was exported to a map1.CSV file in the previous section of the chapter, and you should have renamed the file map1.txt to open it in Notepad. Simply copy that data and paste it into the `map` array.

This is the easiest way to use the maps created by Mappy for your game levels, and I encourage you to gain a working knowledge of this method because it is probably the best option for most games. When you have progressed to the point where you'd like to add some advanced features (such as blocking walls and obstacles on the level), you can move on to loading and drawing Mappy files directly.

Using a Mappy Level File

The Mappy file structure is binary and includes not only the data, but also the tiles. A library has been created to support Mappy within Allegro programs and is available for download on the Mappy Web site at http://www.tilemap.co.uk. The library is called MappyAL, and the current release at the time of this writing is 11D. For distribution and licensing reasons, I have chosen not to include this library on the book's CD-ROM (although the author offers it for free on the Web site). When you download MappyAL (which is currently called mapalr11.zip, but that is likely to change), extract the zip file to find some source code files therein.

All you need are the mappyal.c and mappyal.h files from the zip archive to use Mappy map files in your own programs. Because I will not be going into the advanced features of Mappy or the MappyAL library, I encourage you to browse the Mappy home page, view the tutorials, and download the many source code examples (including many complete games) to learn about the more advanced features of Allegro.

The MappyAL library is very easy to use. Basically, you call `MapLoad` to open a Mappy file. `MapDrawBG` is used to draw a background of tiles, and `MapDrawFG` draws foreground tiles (specified by layer number). There is one drawback to the MappyAL library—it was written quite a long time ago, back in the days when VGA mode 13h (320×200) was popular. Unfortunately, the MappyAL library only renders 8-bit (256 color) maps correctly.

You can convert a true color map to 8-bit color. Simply open the MapTools menu and select Useful Functions, Change Block Size/Depth. This will change the color depth of the map file; you can then import 8-bit tiles and the map will be restored. Paint Shop Pro can easily convert the tiles used in this chapter to 8-bit without too much loss of quality. Ideally, I recommend using the simple text map data due to this drawback.

Now it's time to write a short test program to see how to load a native Mappy file containing map data and tiles, and then display the map on the screen with the ability to scroll the map. Create a new project, add a reference to the Allegro library, and add the mappyal.c and mappyal.h files to the project. (These source code files provide support for Mappy in your Allegro programs.) Then, type the following code into the main.c file. You can use the map1.FMP file you saved earlier in this chapter—or you can use any Mappy file you want to test, because this program can render any Mappy file regardless of dimensions (which are stored inside the map file rather than in the source code). Figure 12.12 shows the *TestMappy* program running.

```
#include <conio.h>
#include <stdlib.h>
#include "allegro.h"
#include "mappyal.h"
```

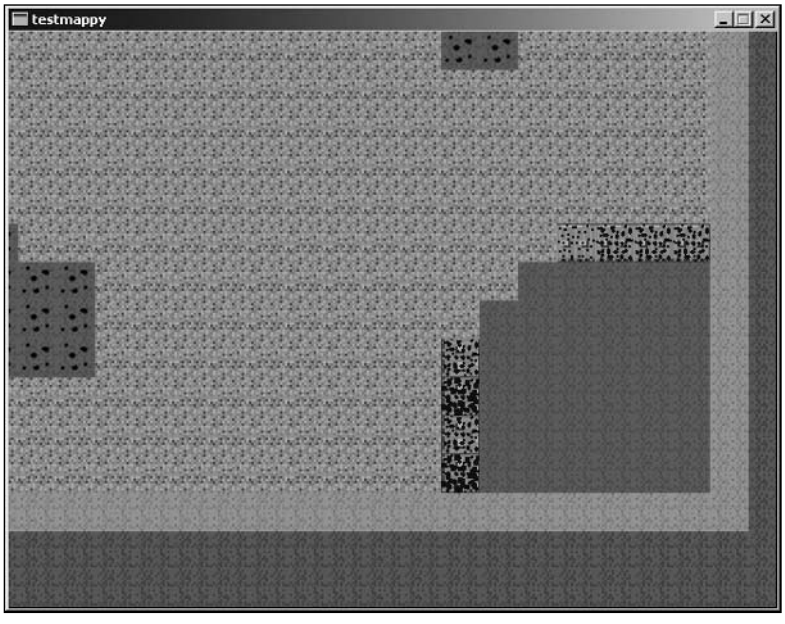


Figure 12.12 The *TestMappy* program demonstrates how to load a native Mappy file.

```
#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define WHITE makecol(255,255,255)

//x, y offset in pixels
int xoffset = 0;
int yoffset = 0;

//double buffer
BITMAP *buffer;

void main (void)
{
    //initialize program
    allegro_init();
    install_timer();
    install_keyboard();
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    text_mode(-1);
```

```
//create the double buffer and clear it
buffer = create_bitmap(SCREEN_W, SCREEN_H);
if (buffer==NULL)
{
    allegro_message("Error creating double buffer");
    return;
}
clear(buffer);

//load the Mappy file
if (MapLoad("map1.fmp"))
{
    allegro_message ("Can't find map1.fmp");
    return;
}

//set palette
MapSetPal8();

//main loop
while (!key[KEY_ESC])
{
    //draw map with single layer
    MapDrawBG(buffer, xoffset, yoffset, 0, 0, SCREEN_W-1, SCREEN_H-1);

    //blit the double buffer
    blit (buffer, screen, 0, 0, 0, 0, SCREEN_W-1, SCREEN_H-1);

    //check for keyboard input
    if (key[KEY_RIGHT])
    {
        xoffset+=4;
        //make sure it doesn't scroll beyond map edge
        if (xoffset > 31*32) xoffset = 31*32;
    }
    if (key[KEY_LEFT])
    {
        xoffset-=4;
        if (xoffset < 0) xoffset = 0;
    }
    if (key[KEY_UP])
    {
```

```
            yoffset-=4;
            if (yoffset < 0) yoffset = 0;
        }
        if (key[KEY_DOWN])
        {
            yoffset+=4;
            //make sure it doesn't scroll beyond map edge
            if (yoffset > 33*32) yoffset = 33*32;
        }

       }

    //delete double buffer
    destroy_bitmap(buffer);

    //delete the Mappy level
    MapFreeMem();

    allegro_exit();
    return;
}

END_OF_MAIN()
```

Enhancing Tank War

Now it's time for an update to *Tank War*—the seventh revision to the game. Chapter 11 provided some great fixes and new additions to the game, including animated tanks and non-interrupting explosions. As you might have guessed, this chapter brings Mappy support to *Tank War*. It should be a lot of fun, so let's get started! This is going to be an easy modification (only a few lines of code) because *Tank War* was designed from the start to be flexible. However, a lot of code that will be *removed* from *Tank War* because MappyAL takes care of all the scrolling for you.

Do you remember the dimensions of the map1.fmp file that was used in this chapter? They were 100 tiles across by 100 tiles down. However, the actual map only uses 30 tiles across and 32 tiles down. This is a bit of a problem for *Tank War* because MappyAL will render the entire map, not just the visible portion. The reason the map was set to 100×100 was to make the Mappy tutorial easier to explain, and at the time it did not matter. Now you're dealing with a map that is 3,200×3,200 pixels, which won't work in *Tank War*. (Actually, it will run just fine, but the tanks won't be bounded by the edge of the map.)

To remedy this situation, I have created a new version of the map file used in this chapter.

It is called map3.fmp, and it is located in \chapter12\tankwar along with the project files for this new revision of *Tank War*.

What's great about this situation? You can create a gigantic battlefield map for *Tank War*! There's no reason why you should limit the game to a mere 30×32 tiles. Go ahead and create a huge map with lots of different terrain so that it isn't so easy to find the other player. Of course, if you create a truly magnificent level, you'll need to modify the bullet code. It wasn't designed for large maps, so you can't fire again until the bullet reaches the edge of the map. Just put in a timer so the bullet will expire if it doesn't hit anything after a few seconds.

Proposed Changes to Tank War

The first thing to do is add mappyal.c and mappyal.h to the project to give *Tank War* support for the MappyAL library. I could show you how to render the tiles directly in *Tank War*, which is how the game works now, but it's far easier to use the functions in MappyAL to draw the two scrolling game windows. You can open the completed project from \chapter12\tankwar, or open the Chapter 11 version of the game and make the following changes.

How about a quick overview? Figure 12.13 shows *Tank War* using the map file from the *TestMappy* program! In Figure 12.14, player two is invading the base of player one!

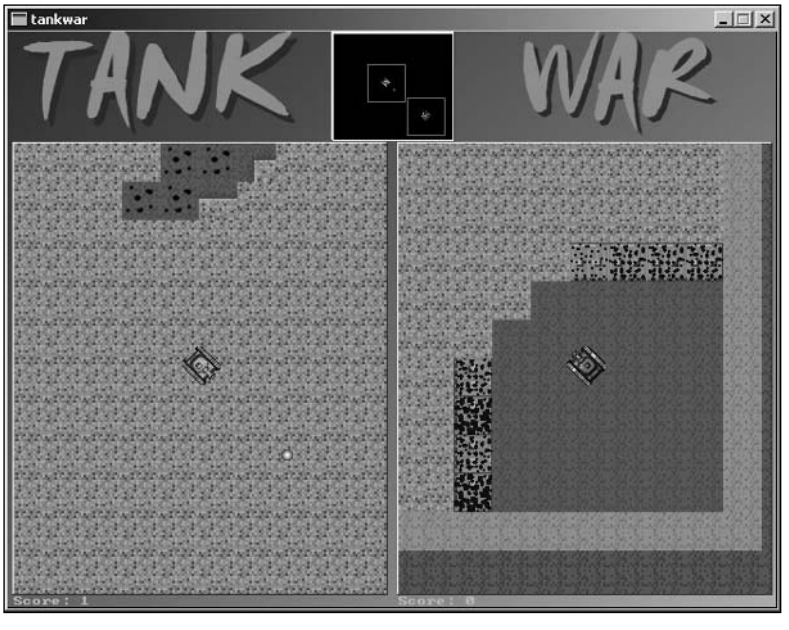


Figure 12.13 *Tank War* now supports the use of Mappy files instead of a hard-coded map.

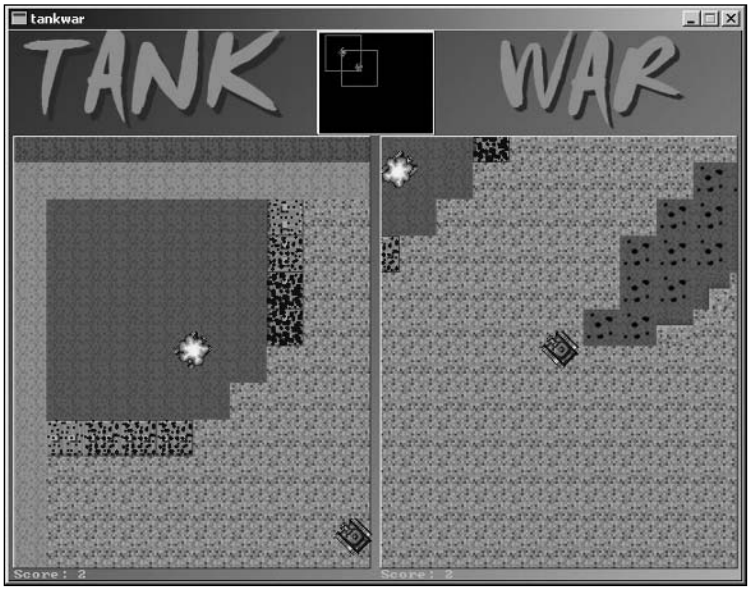


Figure 12.14 Support for Mappy levels gives *Tank War* a lot of new potential because anyone can create a custom battlefield for the game.

Modifying Tank War

Now you can make the necessary changes to *Tank War* to replace the hard-coded background with support for Mappy levels.

Modifying tankwar.h

First up is the tankwar.h header file. Add a new `#define` line to include the mappyal.h file in the project. Note the change in bold.

```
////////////////////////////////////////////////////////////////////////
// Game Programming All In One, Second Edition
// Source Code Copyright (C)2004 by Jonathan S. Harbour
// Tank War Enhancement 7 - tankwar.h
////////////////////////////////////////////////////////////////////////

#ifndef _TANKWAR_H
#define _TANKWAR_H

#include <conio.h>
#include <stdlib.h>
#include "allegro.h"
#include "mappyal.h"
```

Next, remove the reference to the hard-coded map array. (I have commented out the line so you will see what to remove.) This line follows the bitmap definitions.

```
//the game map
//extern int map[];
```

Next, delete the definition for the `tiles` bitmap pointer. Because Mappy levels contain the tiles, your program doesn't need to load them; it only needs to load the map file. (Isn't that great?)

```
//bitmap containing source tiles
//BITMAP *tiles;
```

Finally, delete the reference to the `scroll` bitmap, which is also no longer needed.

```
//virtual background buffer
//BITMAP *scroll;
```

You've ripped out quite a bit of the game with only this first file! That is one fringe benefit to using MappyAL—a lot of source code formerly required to do scrolling is now built into MappyAL.

Modifying setup.c

Next up is the setup.c source code file. Scroll down to the `setupscreen` function and slash the code that loads the tiles and draws them on the virtual background image. You can also delete the section of code that created the virtual background. I'll list the entire function here with the code commented out that you should delete. Note the changes in bold.

```
void setupscreen()
{
    int ret;

    //set video mode
    set_color_depth(16);
    ret = set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    if (ret != 0) {
        allegro_message(allegro_error);
        return;
    }

    text_mode(-1);

/*  REMOVE THIS ENTIRE SECTION OF COMMENTED CODE
    //create the virtual background
    scroll = create_bitmap(MAPW, MAPH);
```

```
    if (scroll == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating virtual background");
        return;
    }

    //load the tile bitmap
    tiles = load_bitmap("tiles.bmp", NULL);
    if (tiles == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error loading tiles.bmp file");
        return;
    }

    //now draw tiles on virtual background
    for (tiley=0; tiley < scroll->h; tiley+=TILEH)
    {
        for (tilex=0; tilex < scroll->w; tilex+=TILEW)
        {
            //use the result of grabframe directly in blitter
            blit(grabframe(tiles, TILEW+1, TILEH+1, 0, 0, COLS, map[n++]),
                scroll, 0, 0, tilex, tiley, TILEW, TILEH);
        }
    }

    //done with tiles
    destroy_bitmap(tiles);

    END OF THE CHOPPING BLOCK
*/

    //load screen background
    back = load_bitmap("background.bmp", NULL);
    if (back == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error loading background.bmp file");
        return;
    }
```

```
    //create the double buffer
    buffer = create_bitmap(WIDTH, HEIGHT);
    if (buffer == NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating double buffer");
        return;
    }

    //position the radar
    radarx = 270;
    radary = 1;

    //position each player
    scrollx[0] = 100;
    scrolly[0] = 100;
    scrollx[1] = MAPW - 400;
    scrolly[1] = MAPH - 500;

    //position the scroll windows
    startx[0] = 5;
    starty[0] = 93;
    startx[1] = 325;
    starty[1] = 93;
}
```

Modifying tank.c

Now open up the tank.c file and scroll down to the `movetank` function. Down at the bottom of the function, you'll see the section of code that keeps the tank inside the boundary of the map. This was based on the virtual background bitmap's width and height, but now it needs to be based on the Mappy level size instead. The `mapwidth`, `mapblockwidth`, `mapheight`, and `mapblockheight` variables are global and found inside mappyal.h. Make the changes noted in bold.

```
    //keep tank inside bounds
    if (scrollx[num] < 0)
    {
        scrollx[num] = 0;
        tanks[num]->xspeed = 0;
    }

    else if (scrollx[num] > mapwidth*mapblockwidth - SCROLLW)
```

```
    {
        scrollx[num] = mapwidth*mapblockwidth - SCROLLW;
        tanks[num]->xspeed = 0;
    }

    if (scrolly[num] < 0)
    {
        scrolly[num] = 0;
        tanks[num]->xspeed = 0;
    }
    else if (scrolly[num] > mapheight*mapblockheight - SCROLLH)
    {
        scrolly[num] = mapheight*mapblockheight - SCROLLH;
        tanks[num]->xspeed = 0;
    }
}
```

Modifying main.c

Now open up the main.c file. The first thing you need to do in main.c is remove the huge `map[]` array definition (with included map tile values). Just delete the whole array, including the `#define B 39` line. I won't list the commented-out code here because the map definition was quite large, but here are the first three lines (for the speed readers out there who tend to miss entire pages at a time):

```
//#define B 39
//int map[MAPW*MAPH] = {
//    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
```

Don't forget to delete the rest of the `map` array definition that follows these lines.

Next, scroll down to the `main` function and add the code that loads the Mappy file, as shown in the bold lines that follow.

```
//main function
void main(void)
{
    int anim;

    //initialize the game
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    setupscreen();
```

```
setuptanks();
loadsprites();

//load the Mappy file
if (MapLoad("map3.fmp"))
{
    allegro_message ("Can't find map3.fmp");
    return;
}

//set palette
MapSetPal8();
```

Next, you need to modify the lines that used to draw the scrolling background and replace them with a call to `MapDrawBG`, which is all you need to draw the background. You can use the same variables as before.

```
//game loop
while(!gameover)
{
    //move the tanks and bullets
    for (n=0; n<2; n++)
    {
        movetank(n);
        animatetank(n);
        movebullet(n);
    }

    //draw background bitmap
    blit(back, buffer, 0, 0, 0, 0, back->w, back->h);

    //draw scrolling windows (now using Mappy)
    for (n=0; n<2; n++)
        MapDrawBG(buffer, scrollx[n], scrolly[n],
            startx[n], starty[n], SCROLLW, SCROLLH);
```

Remove the line of code near the end of `main` that destroys the `scroll` bitmap, which is no longer used.

```
//destroy bitmaps
destroy_bitmap(explode_bmp);
destroy_bitmap(back);
//destroy_bitmap(scroll);
destroy_bitmap(buffer);
```

Only one more change to `main`, and you'll be finished. Add the following line of code at the bottom of `main` to free the MappyAL tile map:

```
    //free the MappyAL memory
    MapFreeMem();

    return;
}
END_OF_MAIN();
```

Summary

This chapter provided the information you need to create maps, levels, and worlds for your games. This very important subject is often glossed over until one finds that a game simply doesn't work without some way to store data to represent the game world. Mappy is an excellent tool for creating game levels. You also gained some experience using Mappy to create some sample maps, along with the source code to load and display those maps. You then added Mappy support to *Tank War*, giving the game a huge boost in playability. Now anyone can create battlefield maps for *Tank War* and fight it out with friends.

Chapter Quiz

1. What is the home site for Mappy?
   A. http://www.mappy.com
   B. http://www.maptiles.com
   C. http://www.tilemap.co.uk
   D. http://www.mappy.co.uk

2. What kind of information is stored in a map file?
   A. Data that represent the tiles comprising a game world
   B. Data that specify the game environment
   C. Data that describe the characters in a game
   D. Data that identify the background images of a game

3. What name is given to the graphic images that make up a Mappy level?
   A. Sprites
   B. Levels
   C. Maps
   D. Tiles

4. What is the default extension of a Mappy file?
   A. SMF
   B. MAP
   C. FMP
   D. BMP

5. Where does Mappy store the saved tile images?
   A. Inside a new bitmap file
   B. Inside the map file
   C. In individual bitmap files
   D. At a location specified by the user

6. What is one example of a retail game that uses Mappy levels?
   A. *Hot Wheels: Stunt Track Driver 2*
   B. *Hyperspace Delivery Boy*
   C. *Real War: Rogue States*
   D. *Wayne Gretzky and the NHLPA All-Stars*

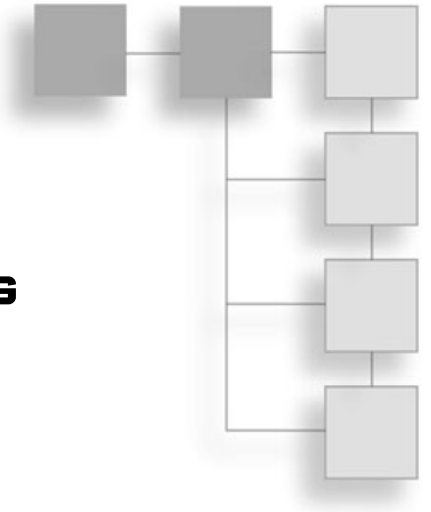
7. What is the recommended format for an exported Mappy level?
   A. Binary
   B. Hexadecimal
   C. C binary array
   D. Text map data

8. Which macro in Mappy fills a map with a specified tile?
   A. Solid Rectangle
   B. Filled Rectangle
   C. Flood Fill Tile
   D. Paste Tiles

9. How much does a licensed copy of Mappy cost?
   A. $10
   B. $20
   C. $50
   D. It's free!

10. Which MappyAL library function loads a Mappy file?
    A. `MapLoad`
    B. `LoadMap`
    C. `MappyLoad`
    D. `OpenMap`

CHAPTER 13



Vertical Scrolling Arcade Games

Most arcade games created and distributed to video arcades in the 1980s and 1990s were scrolling shoot-em-up games (also called simply *shooters*). About an equal number of vertical and horizontal shooters were released. This chapter focuses on vertical shooters (such as *Mars Matrix*) and the next chapter deals with the horizontal variety (although it focuses on platform "jumping" games, not shooters). Why focus two whole chapters on the subject of scrolling games? Because this subject is too often ignored. Most aspiring game programmers know what a shooter is but have no real idea how to develop one. That's where this chapter comes in! This chapter discusses the features and difficulties associated with vertical shooters and explains how to develop a vertical scroller engine, which is used to create a sample game called *Warbirds Pacifica*, a *1942*-style arcade game with huge levels and professionally-drawn artwork.

Here is a breakdown of the major topics in this chapter:

- Building a vertical scroller engine
- Writing a vertical scrolling shooter

Building a Vertical Scroller Engine

Scrolling shooters are interesting programming problems for anyone who has never created one before (and who has benefited from an experienced mentor). In the past, you have created a large memory bitmap and blitted the tiles into their appropriate places on that bitmap, which could then be used as a large game world (for instance, in an earlier revision of *Tank War*). A scrolling shooter, on the other hand, has a game world that is far too large for a single bitmap. For that matter, most games have a world that is too large for a single bitmap, and using such a bitmap goes against good design practices. The world is comprised of tiles, after all, so it would make sense to draw only the tiles needed by the current view.

But for the sake of argument, how big of a world bitmap would you have to use? Mappy (the map editor tool covered in the previous chapter) supports a map of around 30,000 tiles. If you are using a standard 640-pixels-wide screen for a game, that is 20 tiles across, assuming each tile is 32×32. Thirty-thousand tiles divided by 20 tiles across gives you…how many? Fifteen-hundred tiles spread vertically. At 32 pixels each, that is a bitmap image of 640×48,000. That is ridiculously large—so large that I do not need to argue the point any further. Of course, the game world can be much smaller than this, but a good scrolling shooter will have nice, large levels to conquer.

What you need is a vertical scrolling game engine capable of blitting only those tiles needed by the current display. I once wrote a game called *Warbirds* for another book titled *Visual Basic Game Programming with DirectX* (Premier Press, 2002). The game featured a randomly generated vertical scrolling level with warping. This meant that when the scrolling reached the end of the level, it wrapped around to the start of the level and continued scrolling the level without interruption (see Figure 13.1).

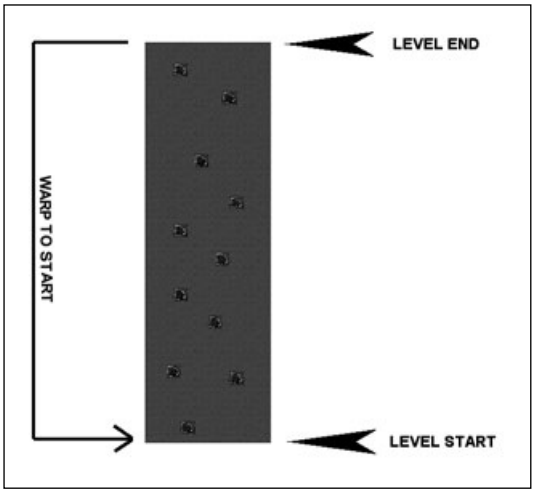


Figure 13.1 Level warping occurs when the end of the level is reached in a scrolling game.

Given that the levels were generated randomly, the game could go on forever without the need for new levels. Unfortunately, as you might have guessed, the levels were quite boring and repetitive. Even with a fairly good warping technique and random map generator, the levels were not very attractive. See Figure 13.2 for a screenshot of *Warbirds*.

If you don't want to use wraparound, or warping, then what happens when the scroller reaches the end? Of course, that's the end of the level. At this point, you want to display the score, congratulate the player, add bonus points, and then proceed to load the next level of the game.

The vertical scroller engine that you'll put together shortly will just sort of stop when it reaches the end of the level; this is a design decision, because I want you to take it from there (load the next level). Then, you can add the custom artwork for a new scrolling shooter, and I'll provide a template by having you build a sample game at the end of this chapter: *Warbirds Pacifica*.

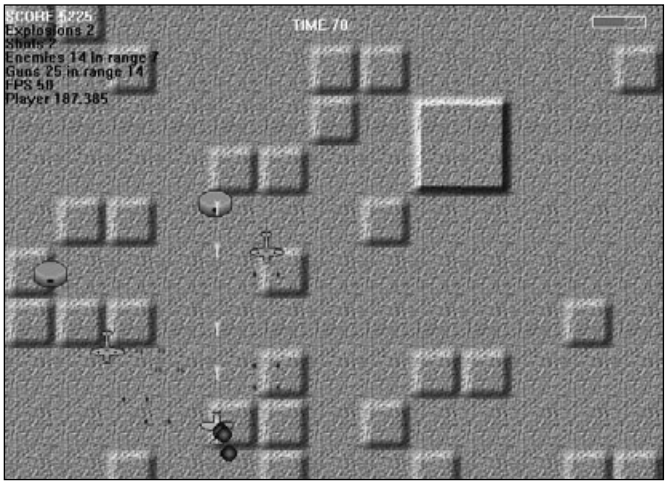


Figure 13.2 *Warbirds* featured a randomly-generated scrolling map.

Creating Levels Using Mappy

The *Warbirds Pacifica* game developed later in this chapter will use high-quality custom levels created with Mappy (which was covered in the previous chapter). Although I suggested using a data array for the maps in simple games, that is not suitable for a game like a scrolling shooter—this game needs variety! To maximize the potential for this game, I'm going to create a huge map file that is 20 tiles wide and 1,500 tiles high! That's equivalent to an image that is 640×48,000 pixels in size. This game *will* be fun; oh yes, it will be!

If you read the previous chapter, then you should have Mappy handy. If not, I recommend you go back and read Chapter 12 because familiarity with Mappy is crucial for getting the most out of this chapter and the one that follows.

Assuming you have Mappy fired up, open the File menu and select New Map. First, be sure to select the Paletted (8bit) option. You want to use simple 8-bit tiles when possible to lighten the memory load with MappyAL, although you may use hi-color or true color tiles if you want. (I wouldn't recommend it generally.) You might recall from the last chapter that MappyAL is a public domain source code library for reading and displaying a Mappy level, and that is what you'll use in this chapter to avoid having to create a tile engine from scratch. Next, for the width and height of each tile, enter 32 and 32, respectively. Next, for the map size, enter 20 for the width and 1500 for the height, as shown in Figure 13.3.

tip

Be sure to select Paletted (8bit) for the color depth of a new map in Mappy if you intend to use the MappyAL library in your Allegro games.

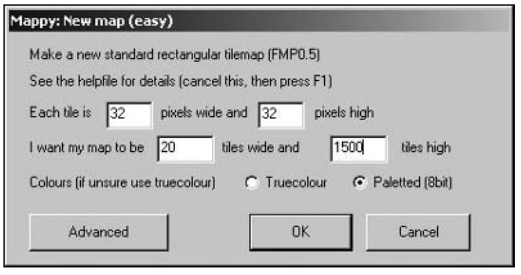


Figure 13.3 Creating a new map in Mappy for the vertical scroller demo

Mappy will create a new map based on your specifications, and then will wait for you to import some tiles (see Figure 13.4).

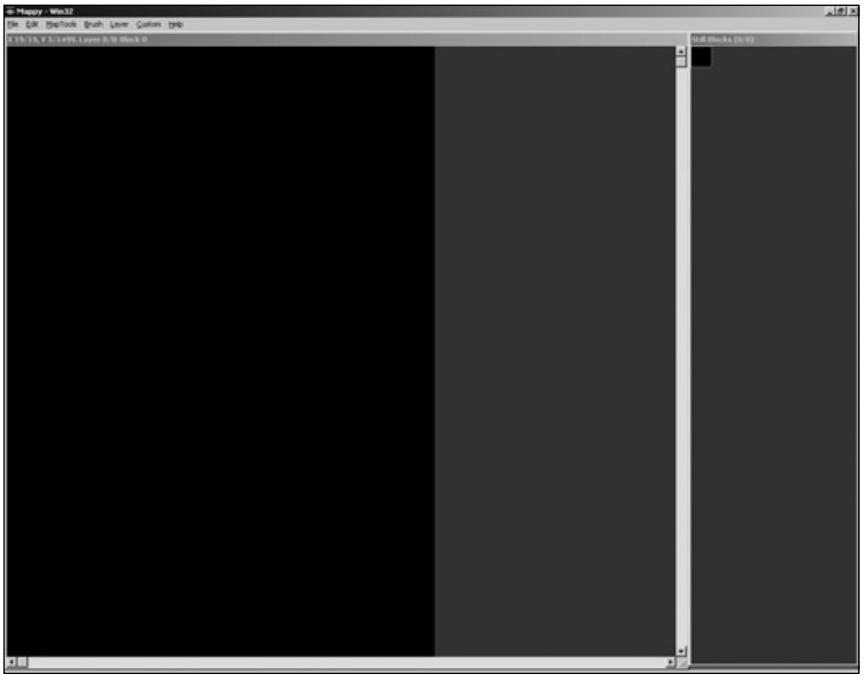


Figure 13.4 Mappy has created the new map and is now waiting for tiles.

Now open the File menu and select Import to bring up the File Open dialog box. This is the part where you have some options. You can use the large collection of tiles I have put together for this chapter or you can create your own tiles and use them. Your results will certainly look different, but if you have your own tiles, by all means use them. Otherwise, I recommend that you copy the maptiles8.bmp file from the CD-ROM to a folder on your hard drive. The tile image is located in \chapter13\VerticalScroller on the CD-ROM under the sources folder for the environment you are using (Visual C++, KDevelop, or Dev-C++). Select this file using the File Open dialog box, and the 32×32 tiles will be added to the tile palette in Mappy (see Figure 13.5).

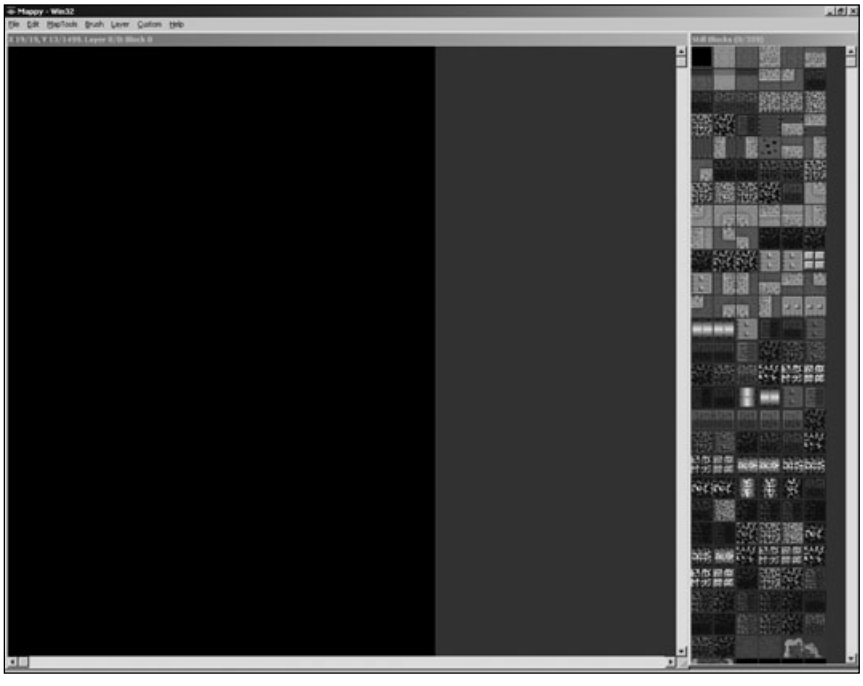


Figure 13.5 The tile palette has been filled with tiles imported from a bitmap file.

If the tiles look familiar, it's because most of them were used in the last chapter. I added new tiles to the maptiles.bmp file while working on the *Warbirds Europa* game. Note that when you add new tiles, you must add them to the bottom row of tiles, not to a column on the right. Mappy reads the tiles from left to right, top to bottom. You can add new tiles to the bottom of the maptiles.bmp file (which I have called maptiles8.bmp to reflect that it is an 8-bit image with 256 colors), and then import the file again into your Mappy map to start using new tiles. Simply select the first tile in the tile palette before you import again, and the existing tiles will be replaced with the new tiles.

Filling in the Tiles

Now that you have a big blank slate for the level, I want to show you how to create a template map file. Because the sample game in this chapter is a World War II shooter based on the arcade game *1942*, you can fill the entire level with a neutral water tile and then save it as a template. At that point, it will be relatively easy to use this template to create a number of levels for the actual game.

tip

All of the graphics in this game are available in the free SpriteLib GPL at http://www.arifeldman.com. Thanks to Ari Feldman for allowing me to use his tiles and sprites in this chapter.

Locate a water tile that is appealing to you. I have added two new water tiles just for this chapter, again from SpriteLib. Again, this was created by Ari Feldman and released into the public domain with his blessing. However, I encourage you to visit Ari's Web site at http://www.arifeldman.com to contact him about commissioning custom artwork for your own games. These are high-quality sprite tiles, and I am grateful to Ari for allowing me to use them.

Because this map is so big, it would take a very long time to fill in all the tiles manually. Thankfully, Mappy suports the Lua scripting language. Although it's beyond the scope of this chapter, you can edit Lua scripts and use them in Mappy. One such script is called Solid Rectangle, and it fills a region of the map with the selected tile. Unfortunately, there's a bug in this Lua script so it leaves out the last row and column of tiles. On a map this big, it takes a long time just to fill in a single column or row. I fixed the bug and have included the script on the CD-ROM. If you have just copied Mappy off the CD-ROM, then you should have the fix. If you have downloaded a new version of Mappy, then you'll have to fill in the unfilled tiles manually.

Having selected an appropriate water tile, open the Custom menu and select Solid Rectangle. A dialog box will appear, asking you to enter four numbers separated by commas. Type in these values:

```
0,0,20,1500
```

If you have the buggy version of this script, then type in:

```
0,0,19,1499
```

Now save the map as template.fmp so it can be reused to create each level of the game. By the way, while you have one large ocean level available, why not have some fun playing with Mappy? See what kind of interesting ocean level you can create using the available tiles. The map should look interesting, but it won't be critical to the game because all the action will take place in the skies.

Let's Scroll It

Now that you have a map ready to use, you can write a short program to demonstrate the feasibility of a very large scrolling level. Figure 13.6 shows the output from the *VerticalScroller* program. As was the case in the last chapter, you will need the MappyAL files to run this program. The mappyal.c and mappyal.h files are located on the CD-ROM under \chapter13\VerticalScroller.

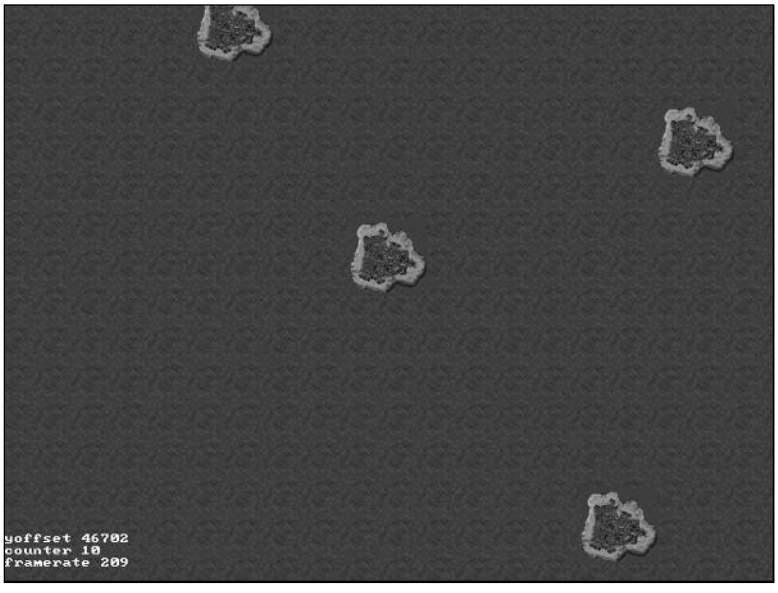


Figure 13.6 The *VerticalScroller* program contains the code for a basic vertical scroller engine.

```
#include "allegro.h"
#include "mappyal.h"

//this must run at 640x480
#define MODE GFX_AUTODETECT_FULLSCREEN
//#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define WHITE makecol(255,255,255)

#define BOTTOM 48000 - HEIGHT
//y offset in pixels
int yoffset = BOTTOM;

//timer variables
volatile int counter;
volatile int ticks;
volatile int framerate;

//double buffer
BITMAP *buffer;
```

```
//calculate framerate every second
void timer1(void)
{
    counter++;
    framerate = ticks;
    ticks=0;
}
END_OF_FUNCTION(timer1)


void main (void)
{
    //initialize program
       allegro_init();
       install_timer();
       install_keyboard();
//    set_color_depth(16);
       set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    text_mode(-1);

    //create the double buffer and clear it
       buffer = create_bitmap(SCREEN_W, SCREEN_H);
       if (buffer==NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating double buffer");
        return;
    }
       clear(buffer);

    //load the Mappy file
       if (MapLoad("level1.fmp"))
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
             allegro_message ("Can't find level1.fmp");
             return;
       }

    //set palette
    MapSetPal8();

    //identify variables used by interrupt function
    LOCK_VARIABLE(counter);
```

```
LOCK_VARIABLE(framerate);
LOCK_VARIABLE(ticks);
LOCK_FUNCTION(timer1);

//create new interrupt handler
install_int(timer1, 1000);

//main loop
while (!key[KEY_ESC])
   {
    //check for keyboard input
         if (key[KEY_PGUP]) yoffset-=4;
         if (key[KEY_PGDN]) yoffset+=4;
    if (key[KEY_UP])   yoffset-=1;
    if (key[KEY_DOWN]) yoffset+=1;

    //make sure it doesn't scroll beyond map edge
    if (yoffset < 0) yoffset = 0;
    if (yoffset > BOTTOM) yoffset = BOTTOM;

    //draw map with single layer
    MapDrawBG(buffer, 0, yoffset, 0, 0, SCREEN_W-1, SCREEN_H-1);

    //update ticks
    ticks++;

    //display some status information
    textprintf(buffer,font,0,440,WHITE,"yoffset %d",yoffset);
    textprintf(buffer,font,0,450,WHITE,"counter %d", counter);
    textprintf(buffer,font,0,460,WHITE,"framerate %d", framerate);

    //blit the double buffer
    acquire_screen();
          blit (buffer, screen, 0, 0, 0, 0, SCREEN_W-1, SCREEN_H-1);
    release_screen();

   }

//delete double buffer
   destroy_bitmap(buffer);

//delete the Mappy level
   MapFreeMem();
```

```
        allegro_exit();
        return;
}

END_OF_MAIN()
```

Writing a Vertical Scrolling Shooter

To best demonstrate a vertical scroller, I have created a simple scrolling shooter as a sample game that you can use as a template for your own games of this genre. Simply replace the map file with one of your own design and replace the basic sprites used in the game, and you can adapt this game for any theme—water, land, undersea, or outer space.

Whereas the player's airplane uses local coordinates reflecting the display screen, the enemy planes use world coordinates that range from 0–639 in the horizontal and 0–47,999 in the vertical. Hey, I told you these maps were huge! The key to making this game work is that a test is performed after each sprite is drawn to determine whether it is within the visible range of the screen. Keep in mind that while the enemy fighters are moving toward the player, the map itself is scrolling downward to simulate forward movement.

Describing the Game

I have called this game *Warbirds Pacifica* because it was based on my earlier *Warbirds* game but set in the Pacific campaign of World War II. The game is set over ocean tiles with frequent islands to help improve the sense of motion (see Figure 13.7).

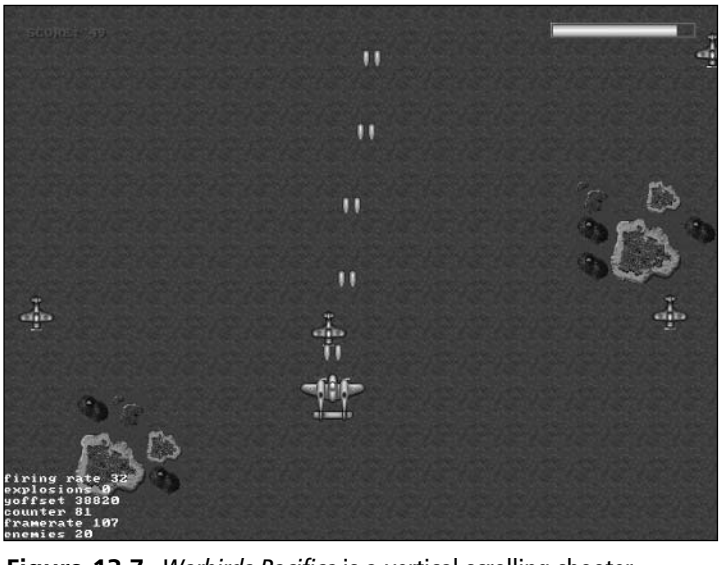


Figure 13.7 *Warbirds Pacifica* is a vertical scrolling shooter.

This is a fast-paced game and even with numerous sprites on the screen, the scrolling engine (provided by MappyAL) doesn't hiccup at all. Take a look at Figure 13.8. The player has a variable firing rate that is improved by picking up power-ups.

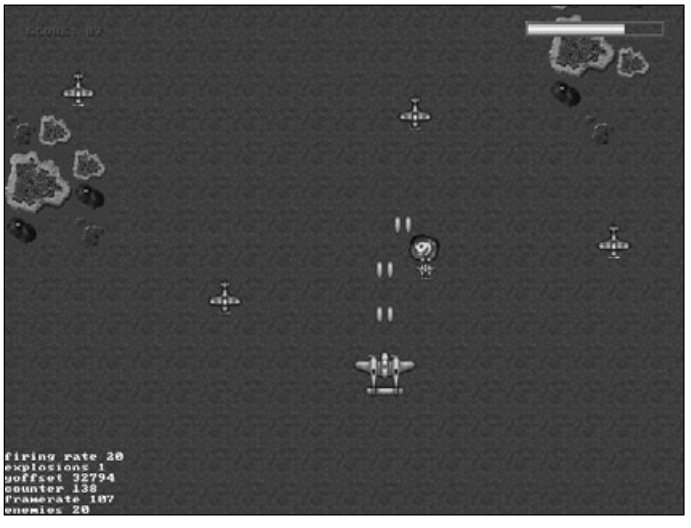


Figure 13.8 The firing rate of the player's P-38 fighter plane is improved with power-ups.

Another cool aspect of the game, thanks to Allegro's awesome sprite handling, is that explosions can overlap power-ups and other bullet sprites due to internal transparency within the sprites (see Figure 13.9). Note also the numerous debug-style messages in the bottom-left corner of the screen. While developing a game, it is extremely helpful to see status values that describe what is going on in order to tweak gameplay. I have modified many aspects of the game thanks to these messages.

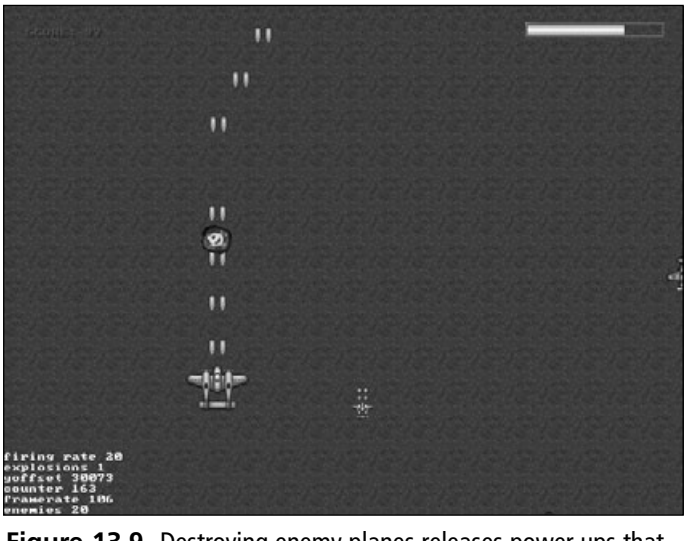


Figure 13.9 Destroying enemy planes releases power-ups that will improve the player's P-38 fighter.

Of course, what would the game be like without any challenge? Although this very early alpha version of *Warbirds Pacifica* does not have the code to allow enemy planes to fire at the player, it does detect collisions with enemy planes, which cause the player's P-38 to explode. (Although gameplay continues, the life meter at the top drops.) One of the first things you will want to do to enhance the game is add enemy firepower (see Figure 13.10).

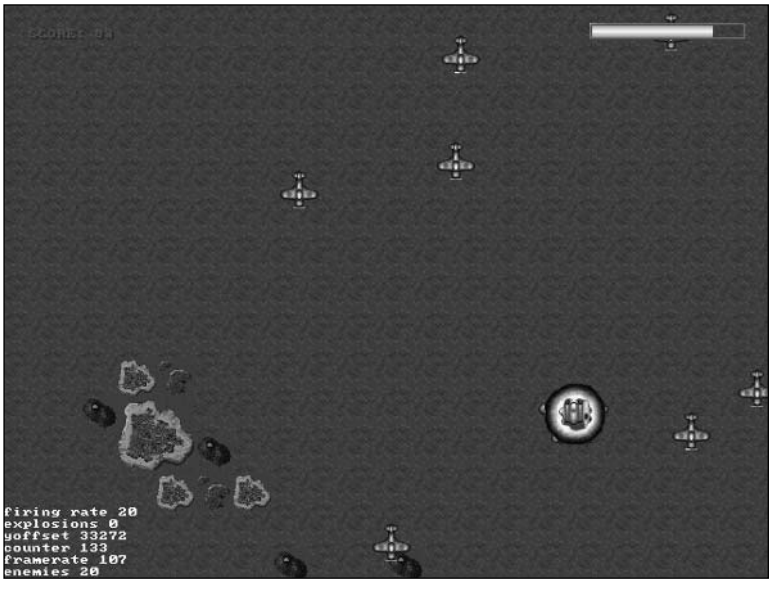


Figure 13.10 The enemy planes might not have much firepower, but they are still capable of Kamikaze attacks!

The Game's Artwork

This game is absolutely loaded with potential! There is so much that could be done with it that I really had to hold myself back when putting the game together as a technology demo for this chapter. It was so much fun adding just a single power-up that I came very close to adding all the rest of the power-ups to the game, including multi-shots! Why such enthusiasm? Because the artwork is already available for building an entire game, thanks to the generosity of Ari Feldman. The artwork featured in this game is a significant part of Ari's SpriteLib.

Let me show you some examples of the additional sprites available that you could use to quickly enhance this game. Figure 13.11 shows a set of enemy bomber sprites. The next image, Figure 13.12, shows a collection of enemy fighter planes that could be used in the game. Notice the different angles. Most shooters will launch squadrons of enemies at the player in formation, which is how these sprites might be used.

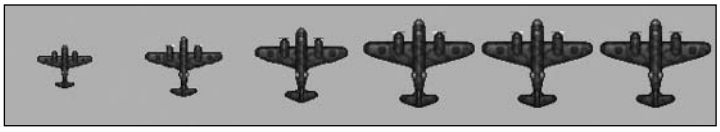


Figure 13.11 A set of enemy bomber sprites. Courtesy of Ari Feldman.

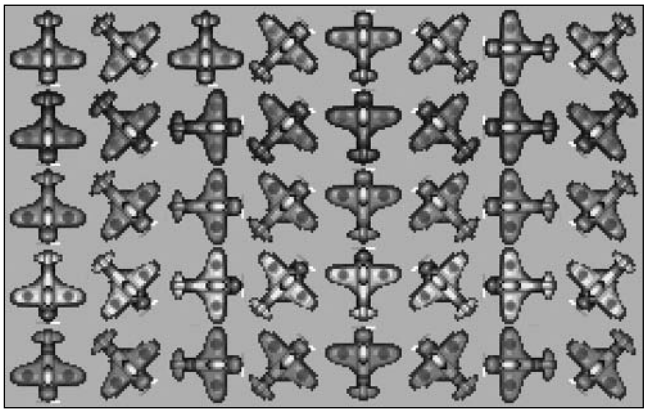


Figure 13.12 A collection of enemy fighter planes. Courtesy of Ari Feldman.

The next image, Figure 13.13, is an animated enemy submarine that comes up out of the water to shoot at the player. This would be a great addition to the game!

Yet another source of sprites for this game is shown in Figure 13.14—an enemy battleship with rotating gun turrets! The next image, Figure 13.15, shows a number of high-quality power-up sprites and bullet sprites. I used the shot power-up in the game as an example so that you can add more power-ups to the game.

Of course, a high-quality arcade game needs a high-quality font that looks really great on the screen. The default font with Allegro looks terrible and should not be used in a game like *Warbirds Pacifica*. Take a look at Figure 13.16 for a sample of the font available for the game with SpriteLib. You can use the existing menus and messages or construct your own using the provided alphabet.

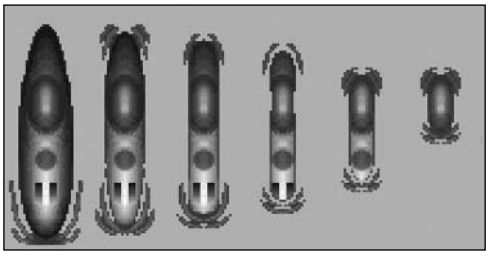


Figure 13.13 An enemy submarine sprite. Courtesy of Ari Feldman.

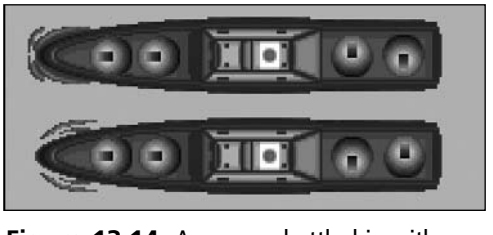


Figure 13.14 An enemy battleship with rotating gun turrets. Courtesy of Ari Feldman.

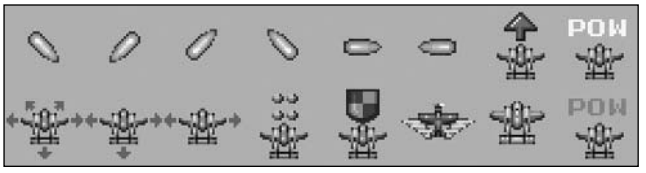


Figure 13.15 A collection of high-quality power-ups and bullets. Courtesy of Ari Feldman.

Figure 13.16 A high-quality font suitable for a scrolling shooter, such as *Warbirds Pacifica.* Courtesy of Ari Feldman.

Writing the Source Code

The source code for *Warbirds Pacifica* is designed to be easy to enhance because my intent was to provide you with a template, something to which you can apply your imagination to complete. The game has all the basic functionality and just needs to be well-rounded and, well, finished.

I recommend you use the *VerticalScroller* program as a basis because it already includes the two support files from the MappyAL library (mappyal.c and mappyal.h). If you are creating a new project from scratch, simply copy these two files to your new project folder and add them to the project by right-clicking on the project name and selecting Add Files to Project.

All the artwork for this game is located on the CD-ROM under \chapter13\Warbirds. You can open the project directly if you are not inclined to type in the source code; however, the more code you type in, the better programmer you will become. In my experience, just the act of typing in a game from a source code listing is a great learning experience. I see aspects of the game—and how it was coded—that are not apparent from simply paging

through the code listing. It helps you to become more intimate and familiar with the source code. This is an absolute must if you intend to learn how the game works in order to enhance or finish it.

warbirds.h

All of the struct and variable definitions are located in the warbirds.h file. You should add a new file to the project (File, New, C/C++ Header File) and give it this name.

```
#ifndef _WARBIRDS_H
#define _WARBIRDS_H

#include "allegro.h"
#include "mappyal.h"

//this must run at 640x480
//#define MODE GFX_AUTODETECT_FULLSCREEN
#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480

#define WHITE makecol(255,255,255)
#define GRAY makecol(60,60,60)
#define RED makecol(200,0,0)

#define MAX_ENEMIES 20
#define MAX_BULLETS 20
#define MAX_EXPLOSIONS 10
#define BOTTOM 48000 - HEIGHT

//define the sprite structure
typedef struct SPRITE
{
    int dir, alive;
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

//y offset in pixels
```

```
int yoffset = BOTTOM;

//player variables
int firecount = 0;
int firedelay = 60;
int health = 25;
int score = 0;

//timer variables
volatile int counter;
volatile int ticks;
volatile int framerate;

//bitmaps and sprites
BITMAP *buffer;
BITMAP *temp;
BITMAP *explosion_images[6];
SPRITE *explosions[MAX_EXPLOSIONS];
BITMAP *bigexp_images[7];
SPRITE *bigexp;
BITMAP *player_images[3];
SPRITE *player;
BITMAP *bullet_images[2];
SPRITE *bullets[MAX_BULLETS];
BITMAP *enemy_plane_images[3];
SPRITE *enemy_planes[MAX_ENEMIES];
BITMAP *progress, *bar;
BITMAP *bonus_shot_image;
SPRITE *bonus_shot;

#endif
```

main.c

Now for the main source code file. The main.c file will contain all of the source code for the *Warbirds Pacifica* template game. Remember, this game is not 100-percent functional for a reason—it was not designed to be a polished, complete game; rather, it was designed to be a template. To make this a complete game, you will want to create additional levels with Mappy; add some code to handle the loading of a new level when the player reaches the end of the first level; and add the additional enemy planes, ships, and so on, as described earlier. Then this game will rock! Furthermore, you will learn how to add sound effects to the game in Chapter 15, "Mastering the Audible Realm: Allegro's Sound Support," which will truly round out this game!

```
#include "warbirds.h"

//reuse our friendly tile grabber from chapter 9
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);

    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;

    blit(source,temp,x,y,0,0,width,height);

    return temp;
}

void loadsprites(void)
{
    int n;

    //load progress bar
    temp = load_bitmap("progress.bmp", NULL);
    progress = grabframe(temp,130,14,0,0,1,0);
    bar = grabframe(temp,6,10,130,2,1,0);
    destroy_bitmap(temp);

    //load bonus shot
    bonus_shot_image = load_bitmap("bonusshot.bmp", NULL);
    bonus_shot = malloc(sizeof(SPRITE));
    bonus_shot->alive=0;
    bonus_shot->x = 0;
    bonus_shot->y = 0;
    bonus_shot->width = bonus_shot_image->w;
    bonus_shot->height = bonus_shot_image->h;
    bonus_shot->xdelay = 0;
    bonus_shot->ydelay = 2;
    bonus_shot->xcount = 0;
    bonus_shot->ycount = 0;
    bonus_shot->xspeed = 0;
    bonus_shot->yspeed = 1;
    bonus_shot->curframe = 0;
```

```
bonus_shot->maxframe = 0;
bonus_shot->framecount = 0;
bonus_shot->framedelay = 0;


//load player airplane sprite
temp = load_bitmap("p38.bmp", NULL);
for (n=0; n<3; n++)
    player_images[n] = grabframe(temp,64,64,0,0,3,n);
destroy_bitmap(temp);

//initialize the player's sprite
player = malloc(sizeof(SPRITE));
player->x = 320-32;
player->y = 400;
player->width = player_images[0]->w;
player->height = player_images[0]->h;
player->xdelay = 1;
player->ydelay = 0;
player->xcount = 0;
player->ycount = 0;
player->xspeed = 0;
player->yspeed = 0;
player->curframe = 0;
player->maxframe = 2;
player->framecount = 0;
player->framedelay = 10;
player->animdir = 1;

//load bullet images
bullet_images[0] = load_bitmap("bullets.bmp", NULL);

//initialize the bullet sprites
for (n=0; n<MAX_BULLETS; n++)
{
    bullets[n] = malloc(sizeof(SPRITE));
    bullets[n]->alive = 0;
    bullets[n]->x = 0;
    bullets[n]->y = 0;
    bullets[n]->width = bullet_images[0]->w;
    bullets[n]->height = bullet_images[0]->h;
    bullets[n]->xdelay = 0;
    bullets[n]->ydelay = 0;
```

```
    bullets[n]->xcount = 0;
    bullets[n]->ycount = 0;
    bullets[n]->xspeed = 0;
    bullets[n]->yspeed = -2;
    bullets[n]->curframe = 0;
    bullets[n]->maxframe = 0;
    bullets[n]->framecount = 0;
    bullets[n]->framedelay = 0;
    bullets[n]->animdir = 0;
}

//load enemy plane sprites
temp = load_bitmap("enemyplane1.bmp", NULL);
for (n=0; n<3; n++)
    enemy_plane_images[n] = grabframe(temp,32,32,0,0,3,n);
destroy_bitmap(temp);

//initialize the enemy planes
for (n=0; n<MAX_ENEMIES; n++)
{
    enemy_planes[n] = malloc(sizeof(SPRITE));
    enemy_planes[n]->alive = 0;
    enemy_planes[n]->x = rand() % 100 + 50;
    enemy_planes[n]->y = 0;
    enemy_planes[n]->width = enemy_plane_images[0]->w;
    enemy_planes[n]->height = enemy_plane_images[0]->h;
    enemy_planes[n]->xdelay = 4;
    enemy_planes[n]->ydelay = 4;
    enemy_planes[n]->xcount = 0;
    enemy_planes[n]->ycount = 0;
    enemy_planes[n]->xspeed = (rand() % 2 - 3);
    enemy_planes[n]->yspeed = 1;
    enemy_planes[n]->curframe = 0;
    enemy_planes[n]->maxframe = 2;
    enemy_planes[n]->framecount = 0;
    enemy_planes[n]->framedelay = 10;
    enemy_planes[n]->animdir = 1;
}

//load explosion sprites
temp = load_bitmap("explosion.bmp", NULL);
for (n=0; n<6; n++)
    explosion_images[n] = grabframe(temp,32,32,0,0,6,n);
```

```
destroy_bitmap(temp);

//initialize the sprites
for (n=0; n<MAX_EXPLOSIONS; n++)
{
    explosions[n] = malloc(sizeof(SPRITE));
    explosions[n]->alive = 0;
    explosions[n]->x = 0;
    explosions[n]->y = 0;
    explosions[n]->width = explosion_images[0]->w;
    explosions[n]->height = explosion_images[0]->h;
    explosions[n]->xdelay = 0;
    explosions[n]->ydelay = 8;
    explosions[n]->xcount = 0;
    explosions[n]->ycount = 0;
    explosions[n]->xspeed = 0;
    explosions[n]->yspeed = -1;
    explosions[n]->curframe = 0;
    explosions[n]->maxframe = 5;
    explosions[n]->framecount = 0;
    explosions[n]->framedelay = 15;
    explosions[n]->animdir = 1;
}

//load explosion sprites
temp = load_bitmap("bigexplosion.bmp", NULL);
for (n=0; n<8; n++)
    bigexp_images[n] = grabframe(temp,64,64,0,0,7,n);
destroy_bitmap(temp);

//initialize the sprites
bigexp = malloc(sizeof(SPRITE));
bigexp->alive = 0;
bigexp->x = 0;
bigexp->y = 0;
bigexp->width = bigexp_images[0]->w;
bigexp->height = bigexp_images[0]->h;
bigexp->xdelay = 0;
bigexp->ydelay = 8;
bigexp->xcount = 0;
bigexp->ycount = 0;
bigexp->xspeed = 0;
bigexp->yspeed = -1;
```

```
    bigexp->curframe = 0;
    bigexp->maxframe = 6;
    bigexp->framecount = 0;
    bigexp->framedelay = 10;
    bigexp->animdir = 1;

}

int inside(int x,int y,int left,int top,int right,int bottom)
{
    if (x > left && x < right && y > top && y < bottom)
        return 1;
    else
        return 0;
}

void updatesprite(SPRITE *spr)
{
    //update x position
    if (++spr->xcount > spr->xdelay)
    {
        spr->xcount = 0;
        spr->x += spr->xspeed;
    }

    //update y position
    if (++spr->ycount > spr->ydelay)
    {
        spr->ycount = 0;
        spr->y += spr->yspeed;
    }

    //update frame based on animdir
    if (++spr->framecount > spr->framedelay)
    {
        spr->framecount = 0;
        if (spr->animdir == -1)
        {
            if (--spr->curframe < 0)
                spr->curframe = spr->maxframe;
        }
        else if (spr->animdir == 1)
        {
```

```
            if (++spr->curframe > spr->maxframe)
                spr->curframe = 0;
        }
    }
}

void startexplosion(int x, int y)
{
    int n;
    for (n=0; n<MAX_EXPLOSIONS; n++)
    {
        if (!explosions[n]->alive)
        {
            explosions[n]->alive++;
            explosions[n]->x = x;
            explosions[n]->y = y;
            break;
        }
    }

    //launch bonus shot if ready
    if (!bonus_shot->alive)
    {
        bonus_shot->alive++;
        bonus_shot->x = x;
        bonus_shot->y = y;
    }
}

void updateexplosions()
{
    int n, c=0;

    for (n=0; n<MAX_EXPLOSIONS; n++)
    {
        if (explosions[n]->alive)
        {
            c++;
            updatesprite(explosions[n]);
            draw_sprite(buffer, explosion_images[explosions[n]->curframe],
                explosions[n]->x, explosions[n]->y);
```

```
            if (explosions[n]->curframe >= explosions[n]->maxframe)
            {
                explosions[n]->curframe=0;
                explosions[n]->alive=0;
            }
        }
    }
    textprintf(buffer,font,0,430,WHITE,"explosions %d", c);

    //update the big "player" explosion if needed
    if (bigexp->alive)
    {
        updatesprite(bigexp);
        draw_sprite(buffer, bigexp_images[bigexp->curframe],
            bigexp->x, bigexp->y);
        if (bigexp->curframe >= bigexp->maxframe)
        {
            bigexp->curframe=0;
            bigexp->alive=0;
        }
    }
}

void updatebonuses()
{
    int x,y,x1,y1,x2,y2;

    //add more bonuses here

    //update bonus shot if alive
    if (bonus_shot->alive)
    {
        updatesprite(bonus_shot);
        draw_sprite(buffer, bonus_shot_image, bonus_shot->x, bonus_shot->y);
        if (bonus_shot->y > HEIGHT)
            bonus_shot->alive=0;

        //see if player got the bonus
        x = bonus_shot->x + bonus_shot->width/2;
        y = bonus_shot->y + bonus_shot->height/2;
        x1 = player->x;
        y1 = player->y;
```

```
            x2 = x1 + player->width;
            y2 = y1 + player->height;

            if (inside(x,y,x1,y1,x2,y2))
            {
                //increase firing rate
                if (firedelay>20) firedelay-=2;

                bonus_shot->alive=0;
            }
        }

}

void updatebullet(SPRITE *spr)
{
    int n,x,y;
    int x1,y1,x2,y2;

    //move the bullet
    updatesprite(spr);

    //check bounds
    if (spr->y < 0)
    {
        spr->alive = 0;
        return;
    }

    for (n=0; n<MAX_ENEMIES; n++)
    {
        if (enemy_planes[n]->alive)
        {
            //find center of bullet
            x = spr->x + spr->width/2;
            y = spr->y + spr->height/2;

            //get enemy plane bounding rectangle
            x1 = enemy_planes[n]->x;
            y1 = enemy_planes[n]->y - yoffset;
            x2 = x1 + enemy_planes[n]->width;
            y2 = y1 + enemy_planes[n]->height;
```

```
            //check for collisions
            if (inside(x, y, x1, y1, x2, y2))
            {
                enemy_planes[n]->alive=0;
                spr->alive=0;
                startexplosion(spr->x+16, spr->y);
                score+=2;
                break;
            }
        }
    }
}

void updatebullets()
{
    int n;
    //update/draw bullets
    for (n=0; n<MAX_BULLETS; n++)
        if (bullets[n]->alive)
        {
            updatebullet(bullets[n]);
            draw_sprite(buffer,bullet_images[0], bullets[n]->x, bullets[n]->y);
        }
}

void bouncex_warpy(SPRITE *spr)
{
    //bounces x off bounds
    if (spr->x < 0 - spr->width)
    {
        spr->x = 0 - spr->width + 1;
        spr->xspeed *= -1;
    }

    else if (spr->x > SCREEN_W)
    {
        spr->x = SCREEN_W - spr->xspeed;
        spr->xspeed *= -1;
    }

    //warps y if plane has passed the player
    if (spr->y > yoffset + 2000)
```

```
    {
        //respawn enemy plane
        spr->y = yoffset - 1000 - rand() % 1000;
        spr->alive++;
        spr->x = rand() % WIDTH;
    }

    //warps y from bottom to top of level
    if (spr->y < 0)
    {
        spr->y = 0;
    }

    else if (spr->y > 48000)
    {
        spr->y = 0;
    }
}


void fireatenemy()
{
    int n;
    for (n=0; n<MAX_BULLETS; n++)
    {
        if (!bullets[n]->alive)
        {
            bullets[n]->alive++;
            bullets[n]->x = player->x;
            bullets[n]->y = player->y;
            return;
        }
    }
}

void displayprogress(int life)
{
    int n;
    draw_sprite(buffer,progress,490,15);

    for (n=0; n<life; n++)
        draw_sprite(buffer,bar,492+n*5,17);
}
```

```
void updateenemyplanes()
{
    int n, c=0;

    //update/draw enemy planes
    for (n=0; n<MAX_ENEMIES; n++)
    {
        if (enemy_planes[n]->alive)
        {
            c++;
            updatesprite(enemy_planes[n]);
            bouncex_warpy(enemy_planes[n]);

            //is plane visible on screen?
            if (enemy_planes[n]->y > yoffset-32 && enemy_planes[n]->y <
                yoffset + HEIGHT+32)
            {
                //draw enemy plane
                draw_sprite(buffer, enemy_plane_images[enemy_planes[n]->curframe],
                    enemy_planes[n]->x, enemy_planes[n]->y - yoffset);
            }
        }
        //reset plane
        else
        {
            enemy_planes[n]->alive++;
            enemy_planes[n]->x = rand() % 100 + 50;
            enemy_planes[n]->y = yoffset - 2000 + rand() % 2000;
        }
    }
    textprintf(buffer,font,0,470,WHITE,"enemies %d", c);
}

void updatescroller()
{
    //make sure it doesn't scroll beyond map edge
    if (yoffset < 5)
    {
        //level is over
        yoffset = 5;
        textout_centre(buffer, font, "END OF LEVEL", SCREEN_W/2,
            SCREEN_H/2, WHITE);
    }
```

```
    if (yoffset > BOTTOM) yoffset = BOTTOM;


    //scroll map up 1 pixel
    yoffset-=1;

    //draw map with single layer
    MapDrawBG(buffer, 0, yoffset, 0, 0, SCREEN_W-1, SCREEN_H-1);
}

void updateplayer()
{
    int n,x,y,x1,y1,x2,y2;

    //update/draw player sprite
    updatesprite(player);
    draw_sprite(buffer, player_images[player->curframe],
        player->x, player->y);

    //check for collision with enemy planes
    x = player->x + player->width/2;
    y = player->y + player->height/2;
    for (n=0; n<MAX_ENEMIES; n++)
    {
        if (enemy_planes[n]->alive)
        {
            x1 = enemy_planes[n]->x;
            y1 = enemy_planes[n]->y - yoffset;
            x2 = x1 + enemy_planes[n]->width;
            y2 = y1 + enemy_planes[n]->height;
            if (inside(x,y,x1,y1,x2,y2))
            {
                enemy_planes[n]->alive=0;
                if (health > 0) health--;
                bigexp->alive++;
                bigexp->x = player->x;
                bigexp->y = player->y;
                score++;
            }
        }
    }
}
```

```
void displaystats()
{
    //display some status information
    textprintf(buffer,font,0,420,WHITE,"firing rate %d", firedelay);
    textprintf(buffer,font,0,440,WHITE,"yoffset %d",yoffset);
    textprintf(buffer,font,0,450,WHITE,"counter %d", counter);
    textprintf(buffer,font,0,460,WHITE,"framerate %d", framerate);

    //display score
    textprintf(buffer,font,22,22,GRAY,"SCORE: %d", score);
    textprintf(buffer,font,20,20,RED,"SCORE: %d", score);
}

void checkinput()
{
    //check for keyboard input
    if (key[KEY_UP])
    {
        player->y -= 1;
        if (player->y < 100)
            player->y = 100;
    }
    if (key[KEY_DOWN])
    {
        player->y += 1;
        if (player->y > HEIGHT-65)
            player->y = HEIGHT-65;
    }
    if (key[KEY_LEFT])
    {
        player->x -= 1;
        if (player->x < 0)
            player->x = 0;
    }
    if (key[KEY_RIGHT])
    {
        player->x += 1;
        if (player->x > WIDTH-65)
            player->x = WIDTH-65;
    }

    if (key[KEY_SPACE])
    {
```

```
        if (firecount > firedelay)
        {
            firecount = 0;
            fireatenemy();
        }
    }
}

//calculate framerate every second
void timer1(void)
{
    counter++;
    framerate = ticks;
    ticks=0;
    rest(2);
}
END_OF_FUNCTION(timer1)

void initialize()
{
    //initialize program
       allegro_init();
       install_timer();
       install_keyboard();
    set_color_depth(16);
       set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    text_mode(-1);
    srand(time(NULL));

    //create the double buffer and clear it
       buffer = create_bitmap(SCREEN_W, SCREEN_H);
       if (buffer==NULL)
    {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("Error creating double buffer");
        return;
    }
       clear(buffer);

    //load the Mappy file
       if (MapLoad("level1.fmp"))
    {
```

```
         set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
      allegro_message ("Can't find level1.fmp");
      return;
   }

    //set palette
    MapSetPal8();

    //identify variables used by interrupt function
    LOCK_VARIABLE(counter);
    LOCK_VARIABLE(framerate);
    LOCK_VARIABLE(ticks);
    LOCK_FUNCTION(timer1);

    //create new interrupt handler
    install_int(timer1, 1000);
}

void main (void)
{
    int n;

    //init game
    initialize();
    loadsprites();

    //main loop
    while (!key[KEY_ESC])
       {
        checkinput();

        updatescroller();

        updateplayer();
        updateenemyplanes();

        updatebullets();
        updateexplosions();
        updatebonuses();

        displayprogress(health);
        displaystats();
```

```
        //blit the double buffer
        acquire_screen();
              blit (buffer, screen, 0, 0, 0, 0, SCREEN_W-1, SCREEN_H-1);
        release_screen();

        ticks++;
        firecount++;
       }

    //delete the Mappy level
       MapFreeMem();

    //delete bitmaps
    destroy_bitmap(buffer);
    destroy_bitmap(progress);
    destroy_bitmap(bar);

    for (n=0; n<6; n++)
        destroy_bitmap(explosion_images[n]);

    for (n=0; n<3; n++)
    {
        destroy_bitmap(player_images[n]);
        destroy_bitmap(bullet_images[n]);
        destroy_bitmap(enemy_plane_images[n]);
    }

    //delete sprites
    free(player);
    for (n=0; n<MAX_EXPLOSIONS; n++)
        free(explosions[n]);
    for (n=0; n<MAX_BULLETS; n++)
        free(bullets[n]);
    for (n=0; n<MAX_ENEMIES; n++)
        free(enemy_planes[n]);

       allegro_exit();
       return;
}

END_OF_MAIN()
```

Summary

Vertical scrolling shooters were once the mainstay of the 1980s and 1990s video arcade, but have not been as prevalent in recent years due to the invasion of 3D, so to speak. Still, the scrolling shooter as a genre has a large and loyal fan following, so it will continue to be popular for years to come. This chapter explored the techniques involved in creating vertical scrollers and produced a sample template game called *Warbirds Pacifica* using the vertical scroller engine (which is really powered by the MappyAL library). I hope you enjoyed this chapter because this is not the end of the scroller! The next chapter takes a turn—a 90-degree turn, as a matter of fact—and covers the horizontal scroller.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. In which game genre does the vertical shooter belong?
   A. Shoot-em-up
   B. Platform
   C. Fighting
   D. Real-time strategy

2. What is the name of the support library used as the vertical scroller engine?
   A. ScrollerEngine
   B. VerticalScroller
   C. MappyAL
   D. AllegroScroller

3. What are the virtual pixel dimensions of the levels in *Warbirds Pacifica*?
   A. 640×480
   B. 48,000×640
   C. 20×1500
   D. 640×48,000

4. What is the name of the level-editing program used to create the first level of *Warbirds Pacifica*?
   A. Happy
   B. Mappy
   C. Snappy
   D. Frappy

5. How many tiles comprise a level in *Warbirds Pacifica*?
   A. 30,000
   B. 1,500
   C. 48,000
   D. 32,768

6. Which of the following games is a vertical scrolling shooter?
   A. *R-Type*
   B. *Mars Matrix*
   C. *Contra*
   D. *Castlevania*

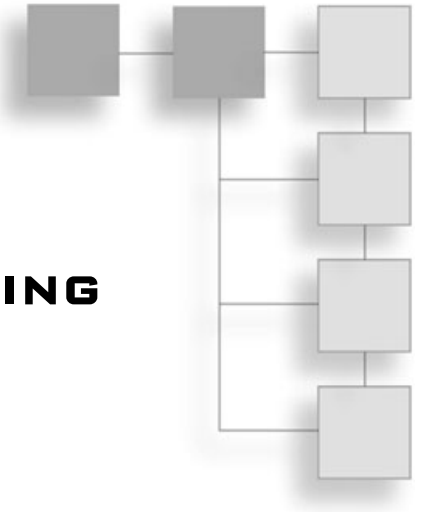
7. Who created the artwork featured in this chapter?
   A. Ray Kurzweil
   B. Clifford Stoll
   C. Ari Feldman
   D. Nicholas Negroponte

8. Which MappyAL function loads a map file?
   A. `LoadMap`
   B. `MapLoad`
   C. `LoadMappy`
   D. `ReadLevel`

9. Which MappyAL function removes a map from memory?
   A. `destroy_map`
   B. `free_mappy`
   C. `DeleteMap`
   D. `MapFreeMem`

10. Which classic arcade game inspired *Warbirds Pacifica*?
    A. *Pac-Man*
    B. *Mars Matrix*
    C. *1942*
    D. *Street Fighter II*

CHAPTER 14



Horizontal Scrolling Platform Games

Everyone has his own opinion of the greatest games ever made. Many games are found on bestseller lists or gamer polls, but there are only a few games that stand the test of time, capable of drawing you in again from a mere glance. One such game is *Super Mario World*, originally released as the launch title for the SNES and now available for the Game Boy Advance. This game is considered by many to be the greatest platformer ever made—if not the best game of all time in any genre. What is it about *Super Mario World* that is so appealing? Aside from the beautiful 2D graphics, charming soundtrack, and likable characters, this game features perhaps the best gameplay ever devised, with levels that are strikingly creative and challenging. The blend of difficulty and reward along with boss characters that go from tough to tougher only scratch the surface of this game's appeal.

Super Mario World is a horizontal scrolling platform game that takes place entirely from the side view (with the exception of the world view). That is the focus of this chapter; it is an introduction to platform games with an emphasis on how to handle tile collisions. Strictly speaking, platform games do not make up the entirety of the horizontal scroller genre; there are perhaps more shoot-em-ups (such as *R-Type* and *Gradius*) in this orientation than there are platformers. I am as big a fan of shooters as I am of platformers; however, because the last chapter focused on a shooter, this chapter will take on the subject of platform game programming.

Using a special feature of Mappy, I'll show you how to design a platform game level that requires very little source code to implement. By the time you have finished this chapter, you will know what it takes to create a platform game and you will have written a sample game that you can tweak and modify to suit your own platform game creations. Here is a list of the major topics in this chapter:

- Understanding horizontal scrolling games
- Developing a scrolling platform game

Understanding Horizontal Scrolling Games

I'm sure you have played many shoot-em-up and platform games in your life, but I will provide you with a brief overview anyway. Although it's tough to beat the gameplay of a vertical scrolling shooter, there is an equal amount of fun to be had with a horizontal scrolling game. The traditional shooters in this genre (*R-Type*, *Gradius*, and so on) have had long and successful runs, with new versions of these classic games released regularly. *R-Type* for Game Boy Color was followed a few years later by *R-Type Advance*, and this is a regular occurrence for a popular game series such as this one.

The other sub-genre of the horizontal scrolling game is the platformer—games such as *Super Mario World* and a vast number of other games of this type. *Kien* is a recent Game Boy Advance platform game with RPG elements. Another old favorite is *Ghosts 'n Goblins*. Have you ever wondered how these games are developed? Such games differ greatly from their horizontal shoot-em-up cousins because platformers by their very nature have the simulated effect of gravity that draws the player down. The goal of the game is to navigate a series of levels comprised of block tiles of various shapes and sizes, such as the game shown in Figure 14.1.

Developing a Platform Scroller

Although it would seem logical to modify the vertical scroller engine from the last chapter to adapt it to the horizontal direction, that only solves the simple problem of how to get tiles on the screen, which is relatively easy to do. The majority of the source code for *Warbirds Pacifica* in the last chapter handled animating the airplanes, bullets, and explosions. Likewise, the real challenge to a platform game is not getting the level to scroll horizontally, but designing the level so that solid tiles can be used as obstacles by the player without requiring a lot of custom code (or worse, a separate data file describing the tiles stored in the map file). In other words, you really want to do most of the work in Mappy, and then perform a few simple function calls in the game to determine when a collision has occurred.

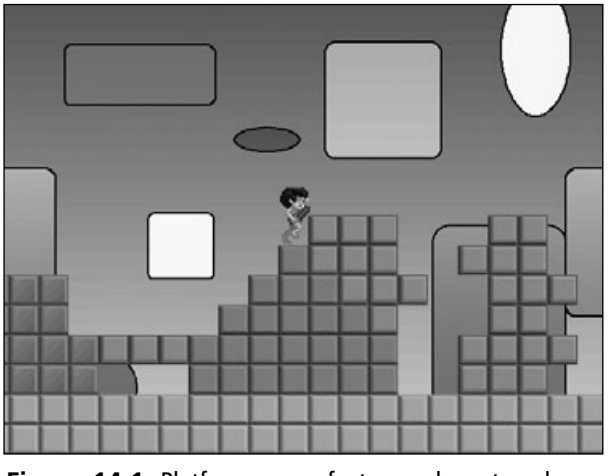


Figure 14.1 Platform games feature a character who walks and jumps.

Some code is required to cause a sprite to interact with tiles in a level, such as when you are blocking the player's movement, allowing the player to jump up on a solid tile, and so on. As you will soon see, the logic for accomplishing this key ingredient of platform gameplay is relatively easy to understand because it uses a simple collision detection routine that is based on the properties of the tiles stored in the Mappy-generated level file.

Creating Horizontal Platform Levels with Mappy

There are many ways to write a platform game. You might store map values in an array in your source code, containing the tile numbers for the map as well as solid block information used for collision detection. This is definitely an option, especially if you are writing a simple platform game. However, why do something the hard way when there is a better way to do it? As you saw in the last two chapters, Mappy is a powerful level-editing program used to create map files (with the .fmp extension). These map files can contain multiple layers for each map and can include animated tiles as well.

In Chapter 10, I explained how to develop a simple scrolling engine using a single large bitmap. (This engine was put to use to enhance the *Tank War* game.) Later, in Chapter 12, I introduced you to Mappy and explained how to walk the level (or preview it with source code). Now that you are using the MappyAL library, introduced in the previous chapter on vertical scrolling, there is no longer any need to work with the map directly. You have seen and experienced a logical progression from simple to advanced, while the difficulty has been reduced in each new chapter. This chapter is even simpler than the last one, and I will demonstrate with a sample program shortly.

Before you can delve into the source code for a platform game, I need to show you some new tricks in Mappy because you need to create a level with two types of blocks—background and foreground. Try not to confuse block type with layering. Mappy supports multiple layers, but I am not using layers to accomplish platform-style gameplay. Instead, the background tiles are static and repeated across the entire level, whereas the foreground tiles are used basically to support the player. Take a look at Figure 14.2 for an example. You can see the player standing on a ledge, which is how this template game looks at startup.

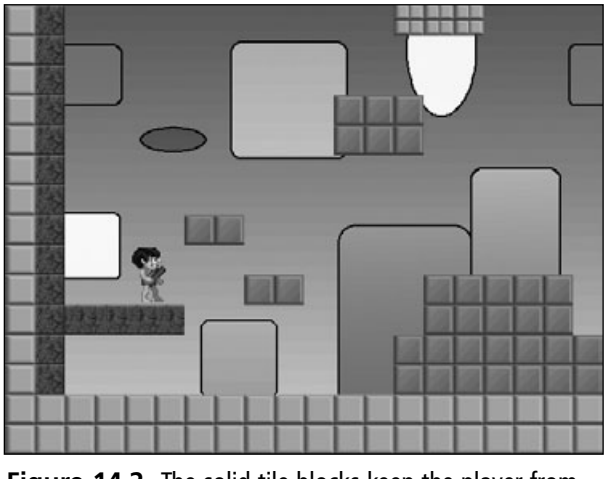


Figure 14.2 The solid tile blocks keep the player from falling through the bottom of the screen.

In the background you see a colorful image containing various shapes, while the foreground contains solid tiles. However, as far as Mappy is concerned, this map is made up of a single layer of tiles.

Allow me to explain. There are basically two ways to add a background to a Mappy level. You can simply insert generic neutral tiles in the empty spaces or you can insert a bitmap image. You might be wondering how to do that. Mappy includes a feature that can divide a solid bitmap into tiles and then construct a map out of it. The key is making sure your starting level size has the same dimensions as the source bitmap.

Run Mappy, open the File menu, and select New Map. Set each tile to 32×32 and set the map size to 20×15 tiles. The result of these dimensions is a 640×480-pixel map. Also, you will be working with true color (16-bit or higher color depth) in this chapter (see Figure 14.3).

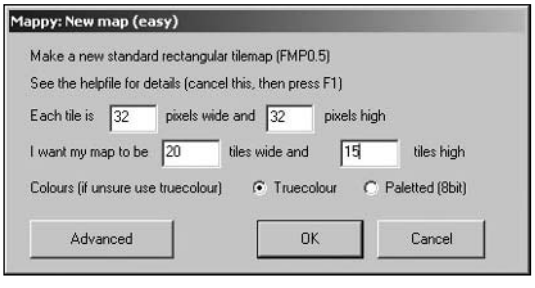


Figure 14.3 The New Map dialog box in Mappy

Now, use your favorite graphic editor to create a 640×480 bitmap image or use one of your favorite bitmaps resized to these dimensions. Normally at this point, you would use Import to load a tile map into Mappy, but the process for converting a solid image into tiles is a little different. Open the MapTools menu. Select the Useful Functions menu item and select Create Map from Big Picture, as shown in Figure 14.4.

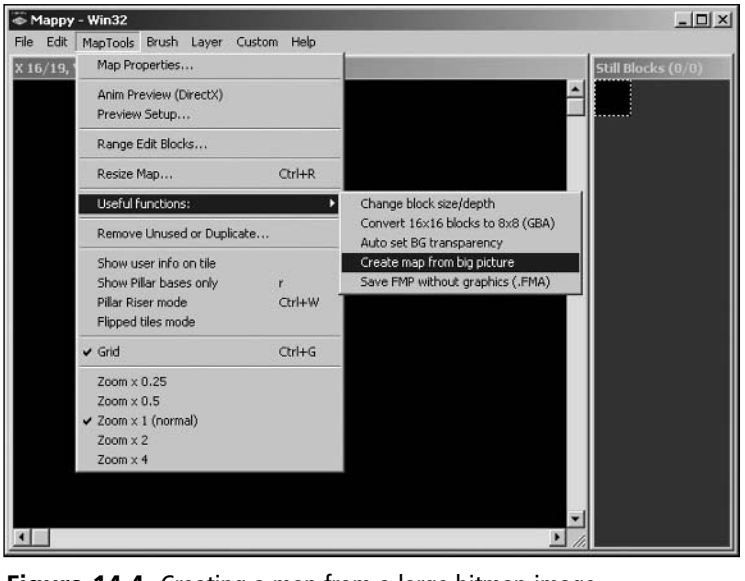


Figure 14.4 Creating a map from a large bitmap image

To demonstrate, I created a colorful bitmap image and used it as the basis for a new map in Mappy using this special feature. But before you create a new map, let me give you a little pointer. The background tiles must be stored with the foreground tiles. You'll want to create a new source bitmap that has room for your background image and the tiles used in the game. Paste your background image into the new bitmap at the top, with the game tiles located underneath. Also be sure to leave some extra space at the bottom so it is easier to add new tiles as you are developing the game (see Figure 14.5).

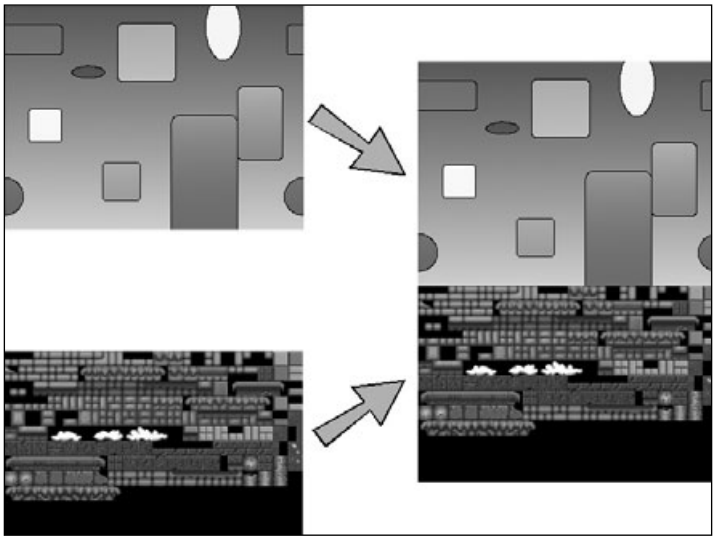


Figure 14.5 The background image and game tiles are stored in the same bitmap image and imported into Mappy.

Using this combined source bitmap, go into Mappy and, after having created the 640×480 map (20 tiles across, 15 tiles down, 32×32 pixels per tile), select Useful Functions, Create Map from Big Picture. The resulting map should look similar to the one shown in Figure 14.6. If you scroll down in the tile palette, you should see the foreground tiles below the background image tiles. See how Mappy has divided the image into a set of tiles? Naturally, you could do this sort of thing with source code by blitting a transparent tile map over a background image, but doing this in Mappy is more interesting (and saves you time writing source code).

You might be wondering, "What next? Am I creating a scrolling game out of a 640×480 tile map?" Not at all; this is only the first step. You must use a tile map that is exactly the same size as the background image in your source bitmap, or the background tiles will be tweaked. Once the background has been generated, you can resize the map.

Open the MapTools menu and select Resize Map to bring up the Resize Map Array dialog box shown in Figure 14.7.

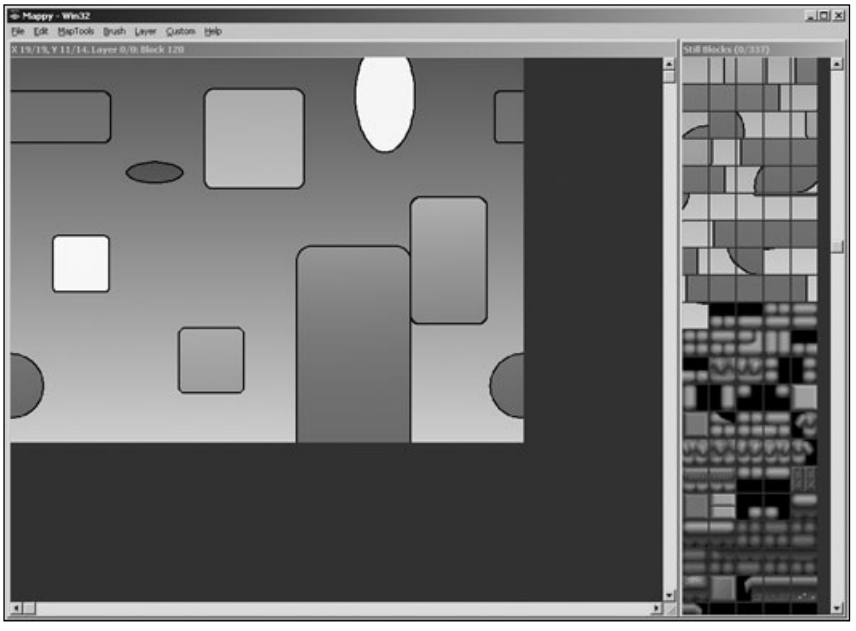


Figure 14.6 A new tile map has been generated based on the source bitmap image.

Press the button labeled 4 to instruct the resize routine to preserve the existing tiles during the resize. The new map can be any size you want, but I normally choose the largest map size possible until I've designed the level, to provide a lot of work space. Besides, it's more fun to include large levels in your games than smaller ones. Just keep in mind that Mappy supports a maximum of 30,000 tiles. If you want your game to scroll upward (as the player is jumping on tiles), keep that in mind. Fifteen tiles deep equates to 480 pixels. You can enter 20 for the height if you want. That is probably a better idea after all, to allow some room for jumping.

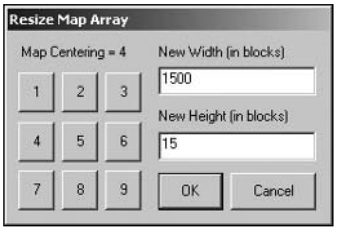


Figure 14.7 The Resize Map Array dialog box

Next, you can experiment with the Brush menu to duplicate the background tiles across the entire level, unless you intend to vary the background. I created a background that meshes well from either side to provide a seamless image when scrolling left or right. Basically, you can choose Grab New Brush, then use the mouse to select a rectangular set of tiles with which to create the brush, and then give the new brush a name. From then on, anywhere you click will duplicate that section of tiles. I used this method to fill the entire level with the same small background tiles. The beautiful thing about this is you end up with a very small memory footprint for such an apparently huge background image.

After resizing and filling the map with the background tiles, the result might look something like Figure 14.8.

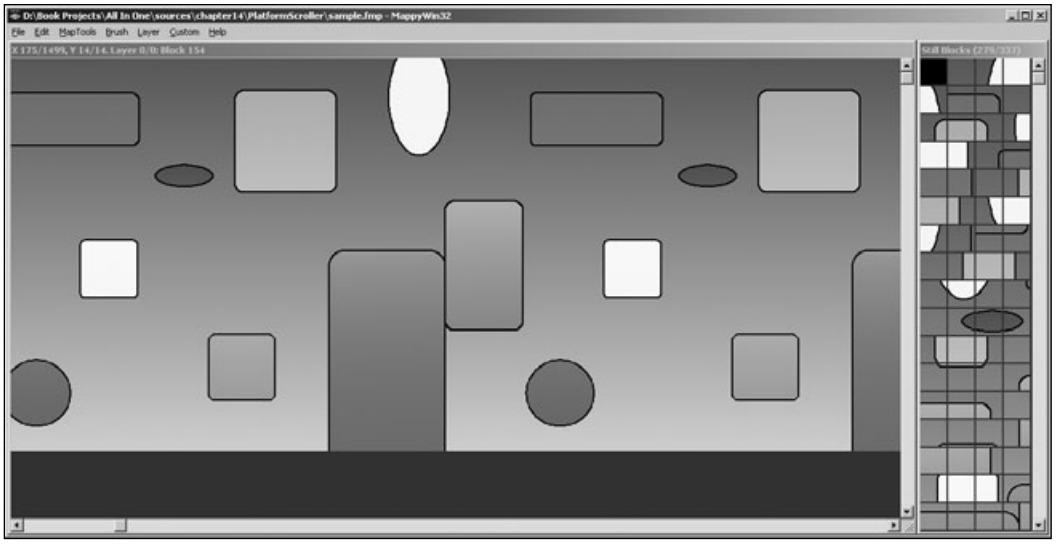


Figure 14.8 A very large horizontally oriented level in Mappy with a bitmap background image

Separating the Foreground Tiles

After you have filled the level with the background tiles, it's time to get started designing the level. But first, you need to make a change to the underlying structure of the foreground tiles, setting them to the FG1 property to differentiate them from the background tiles. This will allow you to identify these tiles in the game to facilitate collision detection on the edges of the tiles.

If you decided to skip over the step earlier in which I suggested adding tiles below the bitmap image, you will need to complete it at this time because the background tiles are not suitable for creating a game level.

The tiles provided on the CD-ROM in the \chapter14\PlatformScroller project folder will suffice if you want to simply copy the file off the CD-ROM. I have called the original tile image blocks1.bmp and the combined image blocks2.bmp. (This second one will be used in the *PlatformScroller* demo shortly.)

Throughout this discussion, I want to encourage you to use your own artwork in the game. Create your own funky background image as I have done for the *PlatformScroller* program that is coming up. As for the tiles, that is a more difficult matter because there is no easy way to draw attractive tiles. As expected, I am using a tileset from Ari Feldman's SpriteLib in this chapter as well. (See http://www.arifeldman.com for more information.)

SpriteLib is a good place to start when you need sprites and tiles with which to develop your game, although it is not a replacement for your own commissioned artwork. Contact Ari to find out how to order a custom sprite set.

Assuming you are using the blocks2.bmp file I created and stored in the project folder for this chapter, you'll want to scroll down in the tile palette to tile 156, the first foreground tile in the tile set (see Figure 14.9).

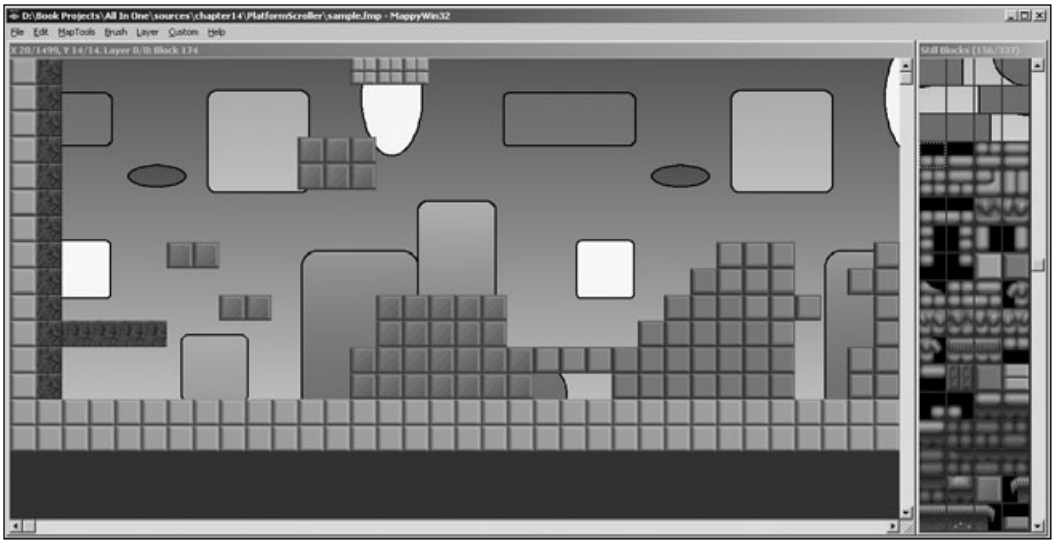


Figure 14.9 Highlighting the first foreground tile in Mappy (right side of the screen)

After you have identified the first foreground tile, you can use this number in the next step. What you are going to do is change the property of the tiles. Double-click on tile #156 to bring up the tile editor. By default, tiles that have been added to the map are assigned to the background, which is the standard level used in simple games (see Figure 14.10).

Do you see the four small boxes on the bottom-left of the Block Properties dialog box? These represent the tile image used for each level (BG, FG1, FG2, FG3). Click on the BG box to bring up the Pick Block Graphic dialog box. Scroll up to the very first tile, which is blank, and select it, and then close the dialog box (see Figure 14.11).

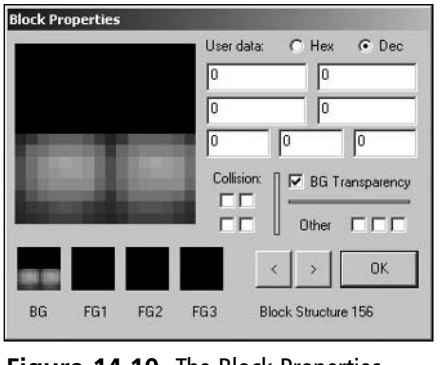


Figure 14.10 The Block Properties dialog box provides an interface for changing the properties of the tiles.

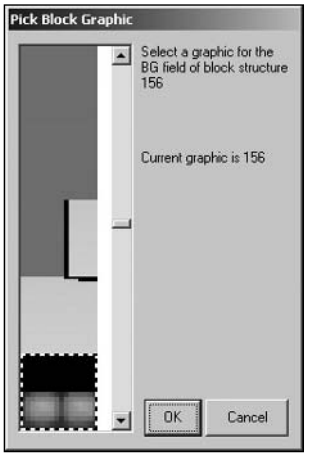


Figure 14.11 The Pick Block Graphic dialog box is used to select a tile for each of the four scroll layers.

Next, click on the FG1 map layer box and locate the tile image you just removed from BG. If you have a hard time locating tiles, I recommend first selecting FG1 before you remove the BG tile. After you have selected the correct tile, you have essentially moved the tile from BG to FG1. In a moment, I will show you a method to quickly make this change on a range of tiles.

The next property to change on the foreground tiles is the collision. If you look for the Collision boxes near the middle of the Block Properties dialog box, you'll see four check boxes. Check all of them so the tile properties look like Figure 14.12.

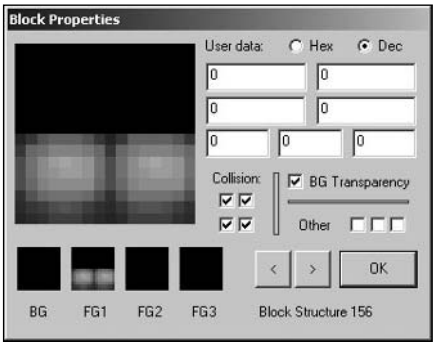


Figure 14.12 Changing the collision properties of the tile

Have you noticed that the Block Properties dialog box has many options that don't immediately seem useful? Mappy is actually capable of storing quite a bit of information for each tile. Imagine being able to set the collision property while also having access to seven numeric values and three Booleans. This is more than enough information for even a highly complex RPG, which typically has more complicated maps than other games. You can set these values in Mappy for use in the game, and you can also read or set the values in your program using the various properties and arrays in MappyAL. For reference, open the mappyal.h file, which contains all the definitions. You can also examine some of the sample programs that come with MappyAL (included on the CD-ROM under \mappy\mappyal).

For the purpose of creating a simple platform game, you only need to set the four collision boxes. (Note that you can fine-tune the collision results in your game by setting only certain collision boxes here.)

Performing a Range Block Edit

Open the MapEdit menu and select Range Edit Blocks to bring up the Range Alter Block Properties dialog box shown in Figure 14.13.

In the From field, enter the number of the first foreground tile. If you are using the blocks2.bmp file for this chapter project, the tile number is 156.

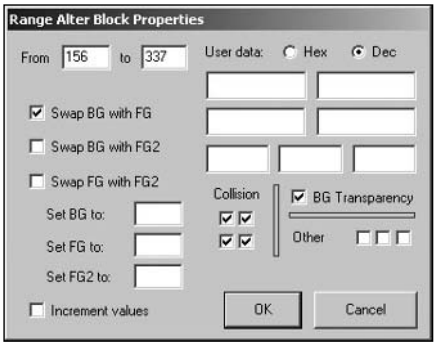


Figure 14.13 The Range Alter Block Properties dialog box

In the To field, enter the number of the last tile in the foreground tile set, which is 337 in this case.

You now have an opportunity to set any of the property values for the range of blocks. Make sure all four collision boxes are fully checked.

The most important thing to do with this range edit is swap the BG for the FG1 layer. This will have the same effect as the manual edit you performed earlier, and it will affect all of the tiles in one fell swoop.

After clicking on OK to perform the action, you can save the map file and move on to the next section. You might want to double-click on one of the tiles to ensure that the change from BG to FG1 has been made.

If you have not added any tiles to your map, you must do that before you continue. As a general rule, the edges of the map should be walled, and a floor should be across the bottom, or at least insert a platform for the start position if your level design does not include a floor. You might want to let the player "fall" as part of the challenge for a level, in which case you'll need to check the Y position of the player's sprite to determine when the player has dropped below the floor. Just be careful to design your level so that there is room for the player to fall. The *PlatformScroller* program to follow does not account for sprites going out of range, but normally when the player falls below the bottom edge of the screen, he has lost a life and must restart the level.

Developing a Scrolling Platform Game

The *PlatformScroller* program included on the CD-ROM is all ready to run, but I will go over the construction of this program and the artwork used by it. You already created the map in the last section, but you can also use the provided map file (sample.fmp) if you want.

Describing the Game

The *PlatformScroller* demo features an animated player character who can run left or right (controlled by the arrow keys) and jump (controlled by the spacebar). The map is quite large, 1,500 tiles across (48,000 pixels) by 15 tiles down (480 pixels). The *PlatformScroller* engine is capable of handling up and down scroll directions, so you can design maps that go up, for instance, by allowing the player to jump from ledge to ledge, by flying, or by some other means. Figure 14.14 shows the player jumping. It is up to the level designer to ensure that the player has a path on which to walk, and it is up to the programmer to handle cases in which the player falls off the screen (and usually dies).

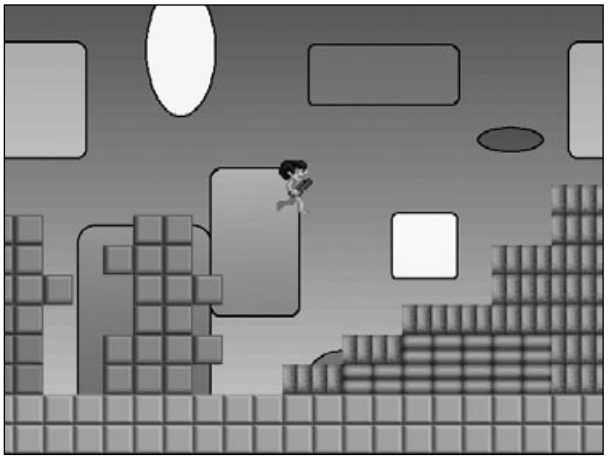


Figure 14.14 The *PlatformScroller* program demonstrates how the player's sprite can interact with tiles using the collision properties set within Mappy.

The background image is an example; you should design your own background imagery, as described earlier in this chapter. Although I have not gotten into the subject in this book, you can also feature parallax scrolling using MappyAL by creating additional layers in the map file. MappyAL has the code to draw parallax layers. Of course, you can draw multiple layers yourself using the standard Allegro `blit` function.

The Game Artwork

The artwork for the *PlatformScroller* demo is primarily comprised of the background image and foreground tiles you have already seen. For reference, the tiles are shown in Figure 14.15.

Figure 14.15 The source tiles used in *PlatformScroller* (which you may use to modify the level)

The only animated artwork in the game is the player character that moves around the level, running and jumping (see Figure 14.16). This character is represented by a sprite with eight frames of animation. Four additional animation frames are provided in the guy.bmp file that you can use for a jumping animation. I have not used these frames to keep the source code listing relatively short (in contrast to the long listing for *Warbirds Pacifica* in the previous chapter).

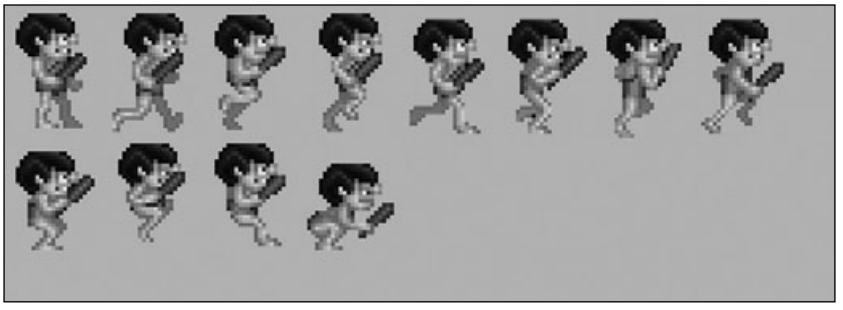


Figure 14.16 The source image containing the animated player character in the *PlatformScroller* demo

Using the Platform Scroller

Most of the source code for the *PlatformScroller* demo is familiar from previous chapters, including the `SPRITE` struct and so on. The new information that might need clarification has to do with tile collision.

You might recall from the Block Properties dialog box in Mappy that you set four collision boxes. These values are stored in a struct called `BLKSTR`.

```
//structure for data blocks
typedef struct {
long int bgoff, fgoff;     //offsets from start of graphic blocks
long int fgoff2, fgoff3;   //more overlay blocks
unsigned long int user1, user2;     //user long data
unsigned short int user3, user4;    //user short data
unsigned char user5, user6, user7;  //user byte data
unsigned char tl : 1;       //bits for collision detection
unsigned char tr : 1;
unsigned char bl : 1;
unsigned char br : 1;
unsigned char trigger : 1;  //bits to trigger an event
unsigned char unused1 : 1;
unsigned char unused2 : 1;
unsigned char unused3 : 1;
} BLKSTR;
```

You might be able to identify the members of the struct after seeing them represented in the Block Properties dialog box. You might notice the seven integer values (`user1` to `user7`) and the three values (`unused1`, `unused2`, `unused3`).

The values you need for collision detection with tiles are called `tl` and `tr` (for top-left and top-right) and `bl` and `br` (you guessed it, for bottom-left and bottom-right). What is needed to

determine when a collision takes place? It's remarkably easy thanks to MappyAL. You can retrieve the block number from an (x,y) position (presumably, the player's sprite location), and then simply return a value specifying whether that tile has one or more of the collision values (`tl`, `tr`, `bl`, `br`) set to 1 or 0. Simply returning the result is enough to pass a true or false response from a collision function. So here you have it:

```
int collided(int x, int y)
{
    BLKSTR *blockdata;
    blockdata = MapGetBlock(x/mapblockwidth, y/mapblockheight);
    return blockdata->tl;
}
```

The `MapGetBlock` function accepts a (row,column) value pair and simply returns a pointer to the block located in that position of the map. This is extremely handy, isn't it?

Writing the Source Code

Because the collision and ability to retrieve a specific tile from the map are so easy to handle, the source code for the *PlatformScroller* program is equally manageable. There is some code to manage the player's position, but a small amount of study reveals the simplicity of this code. The player's position is tracked as player->x and player->y and is compared to the collision values to determine when the sprite should stop moving (left, right, or down). There is currently no facility for handling the bottom edge of tiles; the sprite can jump through a tile from below, but not from above (see Figure 14.17). This might be a feature you will need, depending on the requirements of your own games.

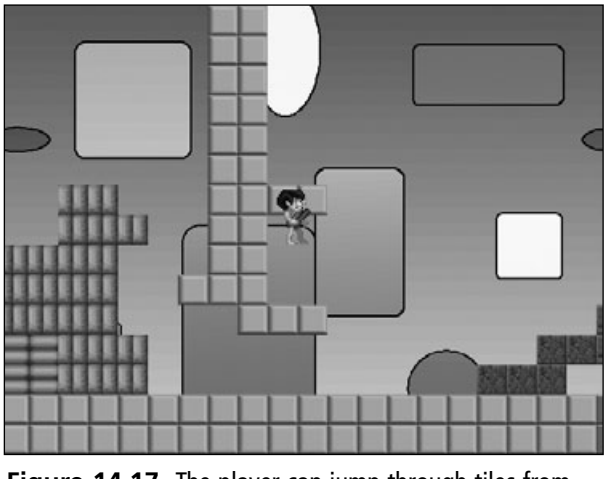


Figure 14.17 The player can jump through tiles from below, but will stop when landing on top of a tile.

The source code for the *Platform Scroller* demo follows. As was the case with the projects in the last chapter, you will need to include the mappyal.h and mappyal.c files (which make up the MappyAL library) and include a linker reference to alleg.lib as usual (or -lalleg, depending on your compiler). I have highlighted in bold significant sections of new code that contribute to the logic of the game or require special attention.

```
#include <stdio.h>
#include <allegro.h>
#include "mappyal.h"

#define MODE GFX_AUTODETECT_FULLSCREEN
#define WIDTH 640
#define HEIGHT 480
#define JUMPIT 1600

//define the sprite structure
typedef struct SPRITE
{
    int dir, alive;
    int x,y;
    int width,height;
    int xspeed,yspeed;
    int xdelay,ydelay;
    int xcount,ycount;
    int curframe,maxframe,animdir;
    int framecount,framedelay;
}SPRITE;

//declare the bitmaps and sprites
BITMAP *player_image[8];
SPRITE *player;
BITMAP *buffer;
BITMAP *temp;

//tile grabber
BITMAP *grabframe(BITMAP *source,
                  int width, int height,
                  int startx, int starty,
                  int columns, int frame)
{
    BITMAP *temp = create_bitmap(width,height);
    int x = startx + (frame % columns) * width;
    int y = starty + (frame / columns) * height;
    blit(source,temp,x,y,0,0,width,height);
    return temp;
}

int collided(int x, int y)
{
```

```
    BLKSTR *blockdata;
    blockdata = MapGetBlock(x/mapblockwidth, y/mapblockheight);
    return blockdata->tl;
}

int main (void)
{
    int mapxoff, mapyoff;
    int oldpy, oldpx;
    int facing = 0;
    int jump = JUMPIT;
    int n;

    allegro_init();
    install_timer();
    install_keyboard();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);

    //load the player sprite
    temp = load_bitmap("guy.bmp", NULL);
    for (n=0; n<8; n++)
        player_image[n] = grabframe(temp,50,64,0,0,8,n);
    destroy_bitmap(temp);

    //initialize the sprite
    player = malloc(sizeof(SPRITE));
    player->x = 80;
    player->y = 100;
    player->curframe=0;
    player->framecount=0;
    player->framedelay=6;
    player->maxframe=7;
    player->width=player_image[0]->w;
    player->height=player_image[0]->h;

    //load the map
    if (MapLoad("sample.fmp")) exit(0);

    //create the double buffer
    buffer = create_bitmap (WIDTH, HEIGHT);
    clear(buffer);
```

```
//main loop
while (!key[KEY_ESC])
{
    oldpy = player->y;
    oldpx = player->x;

    if (key[KEY_RIGHT])
    {
        facing = 1;
        player->x+=2;
        if (++player->framecount > player->framedelay)
        {
            player->framecount=0;
            if (++player->curframe > player->maxframe)
                player->curframe=1;
        }
    }
    else if (key[KEY_LEFT])
    {
        facing = 0;
        player->x-=2;
        if (++player->framecount > player->framedelay)
        {
            player->framecount=0;
            if (++player->curframe > player->maxframe)
                player->curframe=1;
        }
    }
    else player->curframe=0;

    //handle jumping
    if (jump==JUMPIT)
    {
        if (!collided(player->x + player->width/2,
            player->y + player->height + 5))
            jump = 0;

        if (key[KEY_SPACE])
            jump = 30;
    }
    else
    {
```

```
    player->y -= jump/3;
    jump--;
}

if (jump<0)
{
    if (collided(player->x + player->width/2,
        player->y + player->height))
    {
        jump = JUMPIT;
        while (collided(player->x + player->width/2,
            player->y + player->height))
            player->y -= 2;
    }
}

//check for collision with foreground tiles
if (!facing)
{
    if (collided(player->x, player->y + player->height))
        player->x = oldpx;
}
else
{
    if (collided(player->x + player->width,
        player->y + player->height))
        player->x = oldpx;
}

//update the map scroll position
mapxoff = player->x + player->width/2 - WIDTH/2 + 10;
mapyoff = player->y + player->height/2 - HEIGHT/2 + 10;

//avoid moving beyond the map edge
if (mapxoff < 0) mapxoff = 0;
if (mapxoff > (mapwidth * mapblockwidth - WIDTH))
    mapxoff = mapwidth * mapblockwidth - WIDTH;
if (mapyoff < 0)
    mapyoff = 0;
if (mapyoff > (mapheight * mapblockheight - HEIGHT))
    mapyoff = mapheight * mapblockheight - HEIGHT;
```

```
        //draw the background tiles
        MapDrawBG(buffer, mapxoff, mapyoff, 0, 0, WIDTH-1, HEIGHT-1);

        //draw foreground tiles
        MapDrawFG(buffer, mapxoff, mapyoff, 0, 0, WIDTH-1, HEIGHT-1, 0);

        //draw the player's sprite
        if (facing)
            draw_sprite(buffer, player_image[player->curframe],
                (player->x-mapxoff), (player->y-mapyoff));
        else
            draw_sprite_h_flip(buffer, player_image[player->curframe],
                (player->x-mapxoff), (player->y-mapyoff));

        //blit the double buffer
        vsync();
        acquire_screen();
        blit(buffer, screen, 0, 0, 0, 0, WIDTH-1, HEIGHT-1);
        release_screen();
       } //while

    //clean up
    for (n=0; n<9; n++)
        destroy_bitmap(player_image[n]);
    free(player);
    destroy_bitmap(buffer);
    MapFreeMem();
    allegro_exit();
}
END_OF_MAIN();
```

Summary

This chapter provided an introduction to horizontal scrolling platform games, explained how to create platform levels with Mappy, and demonstrated how to put platforming into practice with a sample demonstration program that you could use as a template for any number of platform games. This subject might seem dated to some, but when does great gameplay ever get old? If you take a look at the many Game Boy Advance titles being released this year, you'll notice that most of them are scrolling arcade-style games or platformers! The market for such games has not waned in the two decades since the inception of this genre and it does not look like it will let up any time soon. So have fun and create the next *Super Mario World*, and I guarantee you, someone will publish your game.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Which term is often used to describe a horizontal-scrolling game with a walking character?
   A. Shooter
   B. Platform
   C. RPG
   D. Walker

2. What is the name of the map-editing tool you have used in the last several chapters?
   A. Mappy
   B. Map Editor
   C. Mapper
   D. Tile Editor

3. What is the identifier for the Mappy block property representing the background?
   A. BG1
   B. BACK
   C. BG
   D. BGND

4. What is the identifier for the Mappy block property representing the first foreground layer?
   A. FG1
   B. FORE1
   C. FG
   D. LV1

5. Which dialog box allows the editing of tile properties in Mappy?
   A. Tile Properties
   B. Map Tile Editor
   C. Map Block Editor
   D. Block Properties

6. Which menu item brings up the Range Alter Block Properties dialog?
   A. Range Alter Block Properties
   B. Range Edit Blocks
   C. Range Edit Tile Properties
   D. Range Block Edit

7. What is the name of the MappyAL struct that contains information about tile blocks?
   A. `BLOCKS`
   B. `TILEBLOCK`
   C. `BLKSTR`
   D. `BLKINFO`

8. What MappyAL function returns a pointer to a block specified by the (x,y) parameters?
   A. `MapGetBlock`
   B. `GetDataBlock`
   C. `GetTileAt`
   D. `MapGetTile`

9. What is the name of the function that draws the map's background?
   A. `MapDrawBG`
   B. `DrawBackground`
   C. `DrawMapBack`
   D. `DrawMapBG`

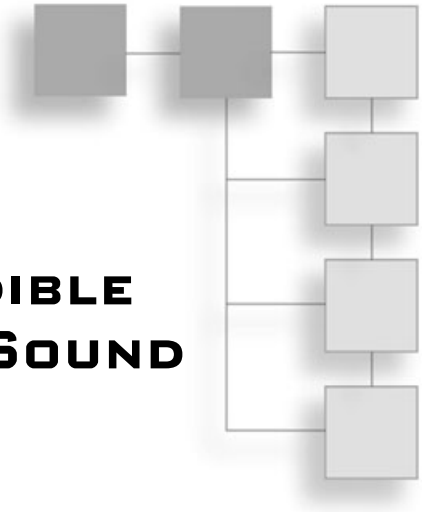
10. Which MappyAL block struct member was used to detect collisions in the sample program?
    A. `bl`
    B. `br`
    C. `tl`
    D. `tr`

PART III

Taking it to the Next Level

Welcome to Part III of *Game Programming All in One, 2nd Edition*. Part III includes six chapters that push the boundaries of your game development skills to the limit. You will find coverage of sound mixing and sample playback, storing game resources in datafiles, and playing FLIC movies before you delve into the complex subjects of artificial intelligence and mathematics. The book ends with a chapter about how to get your games published.

CHAPTER 15



Mastering the Audible Realm: Allegro's Sound Support

Most game programmers are interested in pushing graphics to the limit, first and foremost, and few of us really get enthusiastic about the sound effects and music in a game. That is natural, since the graphics system is the most critical aspect of the game. Sound can be an equal partner with the graphics to provide a memorable, challenging, and satisfying game experience far beyond pretty graphics alone. Indeed, the sound effects and music are often what gamers love most about a game.

This chapter provides an introduction to the sound support that comes with Allegro, and Allegro is significantly loaded with features! Allegro provides an interface to the underlying sound system available on any particular computer system first, and if some features are not available, Allegro will emulate them if necessary. For instance, a basic digital sound mixer is often the first request of a game designer considering the sound support for a game because this is the core of a sound engine. Allegro will interface with DirectSound on Windows systems to provide the mixer and many more features and will take advantage of any similar standardized library support in other operating systems to provide a consistent level of performance and function in a game on any system.

Here is a breakdown of the major topics in this chapter:

- Understanding sound initialization routines
- Working with standard sample playback routines
- Using low-level sample playback routines

The PlayWave Program

I want to get started right away with a sample program to demonstrate how to load and play a WAV file through the sound system because this is the usual beginning of a more complex sound system in a game. Figure 15.1 shows the output from the *PlayWave* program. As with all the other support functions in Allegro, you only need to link to the Allegro library file (alleg.lib or liballeg.a) and include allegro.h in your program—no other special requirements are needed. Essentially, you have a built-in sound system along with everything else in Allegro. Go ahead and try out this program; I will explain how it works later in this chapter. All you need to run it is a sample WAV file, which you can usually find in abundance on the Web in public domain sound libraries. I have included a sample clapping.wav file in the project folder for this program on the CD-ROM; it is in \chapter15\PlayWave.

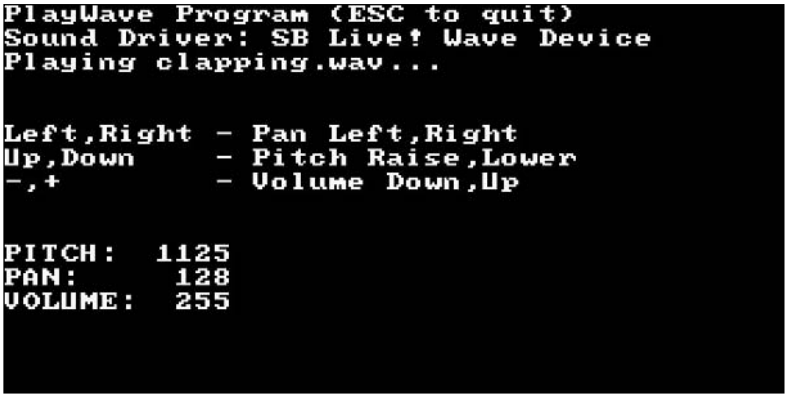


Figure 15.1 The *PlayWave* program demonstrates how to initialize the sound system and play a WAV file.

```
#include <allegro.h>

#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define WHITE makecol(255,255,255)

void main(void)
{
    SAMPLE *sample;
    int panning = 128;
    int pitch = 1000;
    int volume = 128;
```

```
//initialize the program
allegro_init();
install_keyboard();
install_timer();
set_color_depth(16);
set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
text_mode(0);

//install a digital sound driver
if (install_sound(DIGI_AUTODETECT, MIDI_NONE, "") != 0)
{
    allegro_message("Error initializing sound system");
    return;
}

//display program information
textout(screen,font,"PlayWave Program (ESC to quit)",0,0,WHITE);
textprintf(screen,font,0,10,WHITE,"Sound Driver: %s",digi_driver->name);
textout(screen,font,"Playing clapping.wav...",0,20,WHITE);
textout(screen,font,"Left,Right - Pan Left,Right",0,50,WHITE);
textout(screen,font,"Up,Down    - Pitch Raise,Lower",0,60,WHITE);
textout(screen,font,"-,+        - Volume Down,Up",0,70,WHITE);

//load the wave file
sample = load_sample("clapping.wav");
if (!sample)
{
    allegro_message("Error reading wave file");
    return;
}

//play the sample with looping
play_sample(sample, volume, pan, pitch, TRUE);

//main loop
while (!key[KEY_ESC])
{
    //change the panning
    if ((key[KEY_LEFT]) && (panning > 0))
        panning--;
    else if ((key[KEY_RIGHT]) && (panning < 255))
        panning++;
```

```
        //change the pitch (rounding at 512)
        if ((key[KEY_UP]) && (pitch < 16384))
            pitch = ((pitch * 513) / 512) + 1;
        else if ((key[KEY_DOWN]) && (pitch > 64))
            pitch = ((pitch * 511) / 512) - 1;

        //change the volume
        if (key[KEY_EQUALS] && volume < 255)
            volume++;
        else if (key[KEY_MINUS] && volume > 0)
            volume--;

        //adjust the sample
        adjust_sample(sample, volume, pan, pitch, TRUE);

        //pause
        rest(5);

        //display status
        textprintf(screen,font,0,100,WHITE,"PITCH: %5d", pitch);
        textprintf(screen,font,0,110,WHITE,"PAN:   %5d", panning);
        textprintf(screen,font,0,120,WHITE,"VOLUME:%5d", volume);
    }

    //destroy the sample
    destroy_sample(sample);

    //remove the sound driver
    remove_sound();

    return;
}

END_OF_MAIN();
```

Now I want go over some of the functions in the *PlayWave* program and more Allegro sound routines that you'll need. This gives you a preview of what is possible with Allegro, but don't limit your imagination to this meager example because much more is possible.

Sound Initialization Routines

As with the graphics system, you must initialize the sound system before you use the sound routines. Why is that? Allegro runs as lean as possible and only allocates memory

when it is needed. It would be a shame if every Allegro feature were allocated and initialized automatically with even the smallest of programs (such as a command-line utility).

Now I'll go over some of the sound initialization routines you'll be using most often. If you require more advanced features, you can refer to the Allegro documentation, header files, and online sources for information on topics such as sound recording, MIDI, and streaming. I will not cover those features here because they are not normally needed in a game.

Detecting the Digital Sound Driver

The `detect_digi_driver` function determines whether the specified digital sound device is available. It returns the maximum number of voices that the driver can provide or zero if the device is not available. This function must be called before `install_sound`.

```
int detect_digi_driver(int driver_id);
```

Reserving Voices

The `reserve_voices` function is used to specify the number of voices that are to be used by the digital and MIDI sound drivers, respectively. This must be called before `install_sound`. If you reserve too many voices, subsequent calls to `install_sound` will fail. The actual number of voices available depends on the driver, and in some cases you will actually get more than you reserve. To restore the voice setting to the default, you can pass –1 to the function. Be aware that sound quality might drop if too many voices are in use.

```
void reserve_voices(int digi_voices, int midi_voices);
```

Setting an Individual Voice Volume

The `set_volume_per_voice` function is used to adjust the volume of each voice to compensate for mixer output being too loud or too quiet, depending on the number of samples being mixed (because Allegro lowers the volume each time a voice is added to help reduce distortion). This must be called before calling `install_sound`. To play a sample at the maximum volume without distortion, use 0; otherwise, you should call this function with 1 when panning will be used. It is important to understand that each time you increase the parameter by one, the volume of each voice will be halved. So if you pass 2, you can play up to eight samples at maximum volume without distortion (as long as panning is not used). If all else fails, you can pass –1 to restore the volumes to the default levels. Table 15.1 provides a guide.

Here is the definition of the function:

```
void set_volume_per_voice(int scale);
```

Table 15.1 Channel Volume Parameters

Number of Voices	Recommended Parameters
1–8 voices	`set_volume_per_voice(2)`
16 voices	`set_volume_per_voice(3)`
32 voices	`set_volume_per_voice(4)`
64 voices	`set_volume_per_voice(5)`

Initializing the Sound Driver

After you have configured the sound system to meet your needs with the functions just covered, you can call `install_sound` to initialize the sound driver. The default parameters are `DIGI_AUTODETECT` and `MIDI_AUTODETECT`, which instruct Allegro to read hardware settings from a configuration file (which was a significant issue under MS-DOS and is no longer needed with the sound drivers of modern operating systems).

```
int install_sound(int digi, int midi, const char *cfg_path);
```

tip

The third parameter of `install_sound` generally is not needed any longer with modern operating systems that use a sound card device driver model.

Removing the Sound Driver

The `remove_sound` function removes the sound driver and can be called when you no longer need to use the sound routines.

```
void remove_sound();
```

Changing the Volume

The `set_volume` function is used to change the overall volume of the sound system (both digital and MIDI), with a range of 0 to 255. To leave one parameter unchanged while updating the other, pass –1. Most systems with sound cards will have hardware mixers, but Allegro will create a software sound mixer if necessary.

```
void set_volume(int digi_volume, int midi_volume);
```

Standard Sample Playback Routines

The digital sample playback routines can be rather daunting because there are so many of them, but many of these routines are holdovers from when Allegro was developed for MS-DOS. I will cover the most important and useful sample playback routines. Because sound mixers are common in the sound card now, many of the support functions are no longer needed; it is usually enough for any game that a sound mixer is working and sound effects can be played simultaneously.

If some of this listing seems like a header file dump, it is because there are so many sound routines provided by Allegro to manipulate samples and voice channels that a code example for each one would be too difficult (and time consuming). Suffice it to say, many of the seldom-used functions are included here for your reference.

Loading a Sample File

The `load_sample` function will load a .wav or .voc file. The .voc file format was created by Creative Labs for the first Sound Blaster sound card, and this format was very popular with MS-DOS games. It is nice to have the ability to load either file format with this routine because .voc might still be a better format for some older systems.

```
SAMPLE *load_sample(const char *filename);
```

Loading a WAV File

The `load_wav` function will load a standard Windows or OS/2 RIFF WAV file. This function is called by `load_sample` based on the file extension.

```
SAMPLE *load_wav(const char *filename);
```

Loading a VOC File

The `load_voc` function will load a Creative Labs VOC file. This function is called by `load_sample` based on the file extension.

```
SAMPLE *load_voc(const char *filename);
```

Playing a Sample

The `play_sample` function starts playback of a sample using the provided parameters to set the properties of the sample prior to playback. The available parameters are volume, panning, frequency (pitch), and a Boolean value for looping the sample.

The volume and pan range from 0 to 255. Frequency is relative rather than absolute—1000 represents the frequency at which the sample was recorded, 2000 is twice this, and so

on. If the loop flag is set, the sample will repeat until you call `stop_sample` and can be manipulated during playback with `adjust_sample`. This function returns the voice number that was allocated for the sample (or –1 if it failed).

```
int play_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);
```

Altering a Sample's Properties

The `adjust_sample` function alters the properties of a sample during playback. (This is usually only useful for looping samples.) The parameters are volume, panning, frequency, and looping. If there is more than one copy of the same sample playing (as in a repeatable sound, such as an explosion), this will adjust the first one. If the sample is not playing it has no effect.

```
void adjust_sample(const SAMPLE *spl, int vol, int pan, int freq, int loop);
```

Stopping a Sample

The `stop_sample` function stops playback and is often needed for samples that are looping in playback. If more than one copy of the sample is playing (such as an explosion sound), this function will stop all of them.

```
void stop_sample(const SAMPLE *spl);
```

Creating a New Sample

The `create_sample` function creates a new sample with the specified bits (sampling rate), stereo flag, frequency, and length. The returned `SAMPLE` pointer is then treated like any other sample.

```
SAMPLE *create_sample(int bits, int stereo, int freq, int len);
```

Destroying a Sample

The `destroy_sample` function is used to remove a sample from memory. You can call this function even when the sample is playing because Allegro will first stop playback.

```
void destroy_sample(SAMPLE *spl);
```

Low-Level Sample Playback Routines

If you need more detailed control over how samples are played, you can use the lower-level voice functions as an option rather than using the sample routines. The voice routines require more work because you must allocate and free voice data in memory rather than letting Allegro handle such details, but you do gain more control over the mixer and playback functionality.

Allocating a Voice

The `allocate_voice` function allocates memory for a sample in the mixer with default parameters for volume, centered pan, standard frequency, and no looping. After voice playback has finished, it must be removed using `deallocate_voice`. This function returns the voice number or –1 on error.

```
int allocate_voice(const SAMPLE *spl);
```

Removing a Voice

The `deallocate_voice` function removes a voice from the mixer after stopping playback and releases any resources it was using.

```
void deallocate_voice(int voice);
```

Reallocating a Voice

The `reallocate_voice` function changes the sample for an existing voice, which is equivalent to deallocating the voice and then reallocating it again using the new sample.

```
void reallocate_voice(int voice, const SAMPLE *spl);
```

Releasing a Voice

The `release_voice` function releases a voice and allows it to play through to completion without any further manipulation. After playback has finished, the voice is automatically removed. This is equivalent to deallocating the voice at the end of playback.

```
void release_voice(int voice);
```

Activating a Voice

The `voice_start` function activates a voice using the properties configured for the voice.

```
void voice_start(int voice);
```

Stopping a Voice

The `voice_stop` function stops (or rather, pauses) a voice at the current playback position, after which playback can be resumed with a call to `voice_start`.

```
void voice_stop(int voice);
```

Setting Voice Priority

The `voice_set_priority` function sets the priority of the sample in the mixer with a priority range of 0 to 255. Lower-priority voices are cropped when the mixer becomes filled.

```
void voice_set_priority(int voice, int priority);
```

Checking the Status of a Voice

The `voice_check` function determines whether a voice has been allocated, returning a copy of the sample if it is allocated or NULL if the sample is not present.

```
SAMPLE *voice_check(int voice);
```

Returning the Position of a Voice

The `voice_get_position` function returns the current position of playback for that voice or –1 if playback has finished.

```
int voice_get_position(int voice);
```

Setting the Position of a Voice

The `voice_set_position` function sets the playback position of a voice in sample units.

```
void voice_set_position(int voice, int position);
```

Altering the Playback Mode of a Voice

The `voice_set_playmode` function adjusts the loop status of a voice and can be called even while a voice is engaged in playback.

```
void voice_set_playmode(int voice, int playmode);
```

The playmode parameters listed in Table 15.2 can be passed to this function.

Table 15.2 Play Mode Parameters

Play Mode Parameter	Description
PLAYMODE_PLAY	Plays the sample once; this is the default without looping.
PLAYMODE_LOOP	Loops repeatedly through the sample.
PLAYMODE_FORWARD	Plays the sample from start to end; supports looping.
PLAYMODE_BACKWARD	Plays the sample in reverse from end to start; supports looping.
PLAYMODE_BIDIR	Plays the sample forward and backward, reversing direction each time the start or end position is reached during playback.

Returning the Volume of a Voice

The `voice_get_volume` function returns the current volume of a voice in the range of 0 to 255.

```
int voice_get_volume(int voice);
```

Setting the Volume of a Voice

The `voice_set_volume` function sets the volume of a voice in the range of 0 to 255.

```
void voice_set_volume(int voice, int volume);
```

Ramping the Volume of a Voice

The `voice_ramp_volume` functions starts a volume ramp up (crescendo) or down (diminuendo) from the current volume to the specified volume for a specified number of milliseconds.

```
void voice_ramp_volume(int voice, int time, int endvol);
```

Stopping a Volume Ramp

The `voice_stop_volumeramp` function interrupts a volume ramp that was previously started with `voice_ramp_volume`.

```
void voice_stop_volumeramp(int voice);
```

Returning the Pitch of a Voice

The `voice_get_frequency` function returns the current pitch of the voice in Hertz (Hz).

```
int voice_get_frequency(int voice);
```

Setting the Pitch of a Voice

The `voice_set_frequency` function sets the pitch of a voice in Hertz (Hz).

```
void voice_set_frequency(int voice, int frequency);
```

Performing a Frequency Sweep of a Voice

The `voice_sweep_frequency` function performs a frequency sweep (glissando) from the current frequency (or pitch) to the specified ending frequency, lasting for the specified number of milliseconds.

```
void voice_sweep_frequency(int voice, int time, int endfreq);
```

Stopping a Frequency Sweep

The `voice_stop_frequency_sweep` function interrupts a frequency sweep that was previously started with `voice_sweep_frequency`.

```
void voice_stop_frequency_sweep(int voice);
```

Returning the Pan Value of a Voice

The `voice_get_pan` function returns the current panning value from 0 (left speaker) to 255 (right speaker).

```
int voice_get_pan(int voice);
```

Setting the Pan Value of a Voice

The `voice_set_pan` function sets the panning position of a voice with a range of 0 (left speaker) to 255 (right speaker).

```
void voice_set_pan(int voice, int pan);
```

Performing a Sweeping Pan on a Voice

The `voice_sweep_pan` function performs a sweeping pan from left to right (or vice versa) from the current panning value to the specified ending value with a duration in milliseconds.

```
void voice_sweep_pan(int voice, int time, int endpan);
```

Stopping a Sweeping Pan

The `voice_stop_pan_sweep` function interrupts a panning sweep operation that was previously started with the `voice_sweep_pan` function.

```
void voice_stop_pan_sweep(int voice);
```

The SampleMixer Program

I think you will be pleasantly surprised by the simplicity of the next demonstration program in this chapter. *SampleMixer* is a short program that shows you how easy it is to feature multi-channel digital sample playback in your own games (and any other programs) using Allegro's digital sound mixer. Figure 15.2 shows the output from the program. As you can see, there is only a simple interface with no bells or whistles.

The WAV files used in this sample program are included on the CD-ROM in the \chapter15\SampleMixer folder.

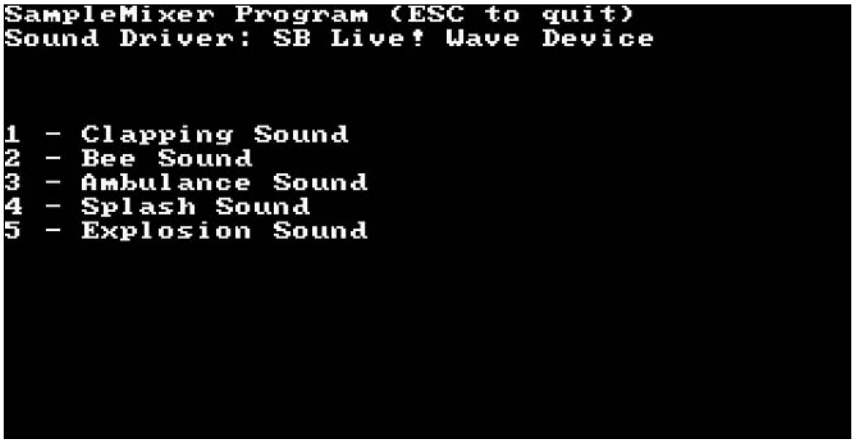


Figure 15.2 The *SampleMixer* program demonstrates the sound mixer provided by Allegro.

```
#include <allegro.h>

#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define WHITE makecol(255,255,255)

void main(void)
{
    SAMPLE *samples[5];
    int volume = 128;
    int pan = 128;
    int pitch = 1000;
    int n;

    //initialize the program
    allegro_init();
    install_keyboard();
    install_timer();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    text_mode(0);

    //install a digital sound driver
    if (install_sound(DIGI_AUTODETECT, MIDI_NONE, "") != 0)
    {
```

```
    allegro_message("Error initializing the sound system");
    return;
}

//display program information
textout(screen,font,"SampleMixer Program (ESC to quit)",0,0,WHITE);
textprintf(screen,font,0,10,WHITE,"Sound Driver: %s", digi_driver->name);

//display simple menu
textout(screen,font,"1 - Clapping Sound",0,50,WHITE);
textout(screen,font,"2 - Bee Sound",0,60,WHITE);
textout(screen,font,"3 - Ambulance Sound",0,70,WHITE);
textout(screen,font,"4 - Splash Sound",0,80,WHITE);
textout(screen,font,"5 - Explosion Sound",0,90,WHITE);

//load the wave file
//normally you would want to include error checking here
samples[0] = load_sample("clapping.wav");
samples[1] = load_sample("bee.wav");
samples[2] = load_sample("ambulance.wav");
samples[3] = load_sample("splash.wav");
samples[4] = load_sample("explode.wav");

//main loop
while (!key[KEY_ESC])
{
    if (key[KEY_1])
        play_sample(samples[0], volume, pan, pitch, FALSE);
    if (key[KEY_2])
        play_sample(samples[1], volume, pan, pitch, FALSE);
    if (key[KEY_3])
        play_sample(samples[2], volume, pan, pitch, FALSE);
    if (key[KEY_4])
        play_sample(samples[3], volume, pan, pitch, FALSE);
    if (key[KEY_5])
        play_sample(samples[4], volume, pan, pitch, FALSE);

    //block fast key repeats
    rest(50);
}

//destroy the samples
for (n=0; n<5; n++)
```

```
        destroy_sample(samples[n]);

    //remove the sound driver
    remove_sound();

    return;
}
END_OF_MAIN();
```

Enhancing Tank War

This chapter will see the final enhancement to *Tank War*! It's been a long journey for this game, from a meager vector-based demo, through the various stages to bitmaps, sprites, scrolling backgrounds, and animation. The final revision to the game (the ninth) will add sound effects. In addition, since this is the last update that will be made to *Tank War*, I have decided to throw in a few extras for good measure. Back in Chapter 5, it was premature to add joystick support to *Tank War*. But much time has passed, and you have learned a great deal in the intervening 10 chapters, so now you'll finally have the opportunity to add joystick support to the game. Along the way, I'll show you how to limit the input routines a little to make the tanks move more realistically.

By the time you have finished this section, *Tank War* will have sound effects, joystick support, and improved gameplay. All that will remain is for you to create some new map files using Mappy to see how far you can take the game! I also suggest you play with the techniques from Chapter 14 for testing collisions with Mappy tiles to add solid blocks to *Tank War*. Because that is beyond the scope of this chapter, I leave the challenge to you. Now let's get started on the changes to the game.

Modifying the Game

The last revision to the game was back in Chapter 12, when you added Mappy support to it. Now you can work on adding sound effects and joystick support, and tweaking the gameplay a little. If you haven't already, open the *Tank War* project from Chapter 12 to make the proposed changes. You can also open the completed project in \chapter15 \tankwar if you want. At the very least, you need to copy the wave files out of the folder and into the project folder on your hard drive. Here is a list of the files you need for this enhancement:

- ammo.wav
- fire.wav
- goopy.wav
- harp.wav

- hit1.wav
- hit2.wav
- ohhh.wav
- scream.wav

Modifying tankwar.h

The first change occurs in tankwar.h because there are some variables needed for this enhancement, as well as a new function prototype. Scroll down in tankwar.h to the variables section and add the lines noted in bold.

```
//variables used for sound effects
#define PAN 128
#define PITCH 1000
#define VOLUME 128
#define NUM_SOUNDS 8
#define AMMO 0
#define HIT1 1
#define HIT2 2
#define FIRE 3
#define GOOPY 4
#define HARP 5
#define SCREAM 6
#define OHHH 7
SAMPLE *sounds[NUM_SOUNDS];

//some variables used to slow down keyboard input
int key_count[2];
int key_delay[2];

//function prototypes
void loadsounds();
void readjoysticks();
void animatetank(int num);
void updateexplosion(int num);
```

Modifying setup.c

Now open the setup.c source code file. Add the new `loadsounds` function to the top of the file. This function loads all the new sound effects that will be used in *Tank War*.

```
void loadsounds()
{
    //install a digital sound driver
```

```
if (install_sound(DIGI_AUTODETECT, MIDI_NONE, "") != 0)
{
    allegro_message("Error initializing sound system");
    return;
}

//load the ammo sound
sounds[AMMO] = load_sample("ammo.wav");
if (!sounds[AMMO])
{
    allegro_message("Error reading ammo.wav");
    return;
}

//load the hit1 sound
sounds[HIT1] = load_sample("hit1.wav");
if (!sounds[HIT1])
{
    allegro_message("Error reading hit1.wav");
    return;
}
//load the hit2 sound
sounds[HIT2] = load_sample("hit2.wav");
if (!sounds[HIT2])
{
    allegro_message("Error reading hit2.wav");
    return;
}
//load the fire sound
sounds[FIRE] = load_sample("fire.wav");
if (!sounds[FIRE])
{
    allegro_message("Error reading fire.wav");
    return;
}
//load the goopy sound
sounds[GOOPY] = load_sample("goopy.wav");
if (!sounds[GOOPY])
{
    allegro_message("Error reading goopy.wav");
    return;
}
```

```
    //load the harp sound
    sounds[HARP] = load_sample("harp.wav");
    if (!sounds[HARP])
    {
        allegro_message("Error reading harp.wav");
        return;
    }
    //load the scream sound
    sounds[SCREAM] = load_sample("scream.wav");
    if (!sounds[SCREAM])
    {
        allegro_message("Error reading scream.wav");
        return;
    }
    //load the ohhh sound
    sounds[OHHH] = load_sample("ohhh.wav");
    if (!sounds[OHHH])
    {
        allegro_message("Error reading ohhh.wav");
        return;
    }

    //cannons are reloading
    play_sample(sounds[0], VOLUME, PAN, PITCH, FALSE);
}
```

Modifying bullet.c

Now open the bullet.c file to add some function calls to play sounds at various points in the game (for instance, during an explosion). The first function in this file is `updateexplosion`. Down at the bottom of this function is an `else` statement. Add the `play_sample` line as shown.

```
    }
    else
    {
        //play "end of explosion" sound
        play_sample(sounds[HARP], VOLUME, PAN, PITCH, FALSE);

        explosions[num]->alive = 0;
        explosions[num]->curframe = 0;
    }
}
```

Now scroll down a little to the `explosion` function. Add the new lines of code as shown. You might be wondering why there are three sounds being played at the start of an explosion. It's for variety! The three sounds together add a distinctive explosion sound, along with a light comical twist. Remember that Allegro mixes sounds, so these are all played at basically the same time.

```
void explode(int num, int x, int y)
{
    //initialize the explosion sprite
    explosions[num]->alive = 1;
    explosions[num]->x = x;
    explosions[num]->y = y;
    explosions[num]->curframe = 0;
    explosions[num]->maxframe = 20;

    //play explosion sounds
    play_sample(sounds[GOOPY], VOLUME, PAN, PITCH, FALSE);
    play_sample(sounds[HIT1], VOLUME, PAN, PITCH, FALSE);
    play_sample(sounds[HIT2], VOLUME, PAN, PITCH, FALSE);
}
```

Now scroll down to the `movebullet` function. You'll make a ton of changes to this function, basically to add more humorous elements to the game. Whenever a bullet hits the edge of the map, a reload sound is played (ammo.wav), which tells the player that he can fire again. Remember that bullets will keep going until they strike the enemy tank or the edge of the map. The next change to this function is quite funny, in my opinion. Whenever there is a near miss of a bullet close to your tank, one of two samples is played. If it's player 1, the scream.wav sample is played, while ohhh.wav is played for a near miss with player 2. This really adds a nice touch to the game, as you'll see when you play it. Now, just make all the changes noted in bold.

```
void movebullet(int num)
{
    int x, y, tx, ty;

    x = bullets[num]->x;
    y = bullets[num]->y;

    //is the bullet active?
    if (!bullets[num]->alive) return;

    //move bullet
    bullets[num]->x += bullets[num]->xspeed;
    bullets[num]->y += bullets[num]->yspeed;
```

```
x = bullets[num]->x;
y = bullets[num]->y;

//stay within the virtual screen
if (x < 0 || x > MAPW-6 || y < 0 || y > MAPH-6)
{
    //play the ammo sound
    play_sample(sounds[AMMO], VOLUME, PAN, PITCH, FALSE);

    bullets[num]->alive = 0;
    return;
}

//look for a direct hit using basic collision
tx = scrollx[!num] + SCROLLW/2;
ty = scrolly[!num] + SCROLLH/2;
if (inside(x,y,tx-15,ty-15,tx+15,ty+15))
{
    //kill the bullet
    bullets[num]->alive = 0;

    //blow up the tank
    x = scrollx[!num] + SCROLLW/2;
    y = scrolly[!num] + SCROLLH/2;

    //draw explosion in enemy window
    explode(num, tanks[!num]->x, tanks[!num]->y);
    scores[num]++;

    //kill any "near miss" sounds
    if (num)
        stop_sample(sounds[SCREAM]);
    else
        stop_sample(sounds[OHHH]);
}

else if (inside(x,y,tx-30,ty-30,tx+30,ty+30))
{
    //it's a near miss!
    if (num)
        //player 1 screams
        play_sample(sounds[SCREAM], VOLUME, PAN, PITCH, FALSE);
    else
```

```
            //player 2 ohhhs
            play_sample(sounds[OHHH], VOLUME, PAN, PITCH, FALSE);
    }
}
```

Now, scroll down a little more to the `fireweapon` function. I have added a single `play_sample` function call that plays a sound whenever a player fires a bullet. This is the basic fire sound. Add the line shown in bold.

```
void fireweapon(int num)
{
    int x = scrollx[num] + SCROLLW/2;
    int y = scrolly[num] + SCROLLH/2;

    //ready to fire again?
    if (!bullets[num]->alive)
    {
        //play fire sound
        play_sample(sounds[FIRE], VOLUME, PAN, PITCH, FALSE);

        bullets[num]->alive = 1;
```

Modifying input.c

Next, open the input.c file. The first thing you must do is add a new function called `readjoysticks`. This function first verifies that a joystick is connected, and then tries to scan the input of one or two joysticks, if present. If you have two joysticks or gamepads, try plugging them into your PC to see how much fun *Tank War* can be when played like a console game! Add the new `readjoysticks` function to the top of input.c.

```
void readjoysticks()
{
    int b, n;

    if (num_joysticks)
    {
        //read the joystick
        poll_joystick();

        for (n=0; n<2; n++)
        {
            //left stick
            if (joy[n].stick[0].axis[0].d1)
                turnleft(n);
```

```
            //right stick
            if (joy[n].stick[0].axis[0].d2)
                turnright(n);

            //forward stick
            if (joy[n].stick[0].axis[1].d1)
                forward(n);

            //backward stick
            if (joy[n].stick[0].axis[1].d2)
                backward(n);

            //any button will do
            for (b=0; b<joy[n].num_buttons; b++)
                if (joy[n].button[b].b)
                {
                    fireweapon(n);
                    break;
                }
        }
    }
}
```

Next, you need to make some modifications to the `forward`, `backward`, `turnleft`, and `turnright` functions. These changes help slow down the device input so it's easier to control the tanks. (Previously, you might recall, the tanks would turn far too fast.) This also makes the tank movement feel more realistic because you must speed up gradually, rather than going from 0 to 60 in 0.5 seconds, as the game was played before. Note the changes in bold.

```
void forward(int num)
{
    if (key_count[num]++ > key_delay[num])
    {
        key_count[num] = 0;

        tanks[num]->xspeed++;
        if (tanks[num]->xspeed > MAXSPEED)
            tanks[num]->xspeed = MAXSPEED;
    }
}

void backward(int num)
{
    if (key_count[num]++ > key_delay[num])
```

```
    {
        key_count[num] = 0;

        tanks[num]->xspeed--;
        if (tanks[num]->xspeed < -MAXSPEED)
            tanks[num]->xspeed = -MAXSPEED;
    }
}

void turnleft(int num)
{
    if (key_count[num]++ > key_delay[num])
    {
        key_count[num] = 0;

        tanks[num]->dir--;
        if (tanks[num]->dir < 0)
            tanks[num]->dir = 7;
    }
}

void turnright(int num)
{
    if (key_count[num]++ > key_delay[num])
    {
        key_count[num] = 0;

        tanks[num]->dir++;
        if (tanks[num]->dir > 7)
            tanks[num]->dir = 0;
    }
}
```

The last change you'll make is to the `getinput` function. There has been a `rest` function call in here since the first version of the game, while the timing of the game belongs in the main loop. Simply delete the line indicated in bold (and commented out).

```
void getinput()
{
    //hit ESC to quit
    if (key[KEY_ESC])   gameover = 1;

    //WASD - SPACE keys control tank 1
    if (key[KEY_W])     forward(0);
```

```
    if (key[KEY_D])     turnright(0);
    if (key[KEY_A])     turnleft(0);
    if (key[KEY_S])     backward(0);
    if (key[KEY_SPACE]) fireweapon(0);

    //arrow - ENTER keys control tank 2
    if (key[KEY_UP])    forward(1);
    if (key[KEY_RIGHT]) turnright(1);
    if (key[KEY_DOWN])  backward(1);
    if (key[KEY_LEFT])  turnleft(1);
    if (key[KEY_ENTER]) fireweapon(1);

    //short delay after keypress
    //rest(20);
}
```

Modifying main.c

Next up is the main.c file, the primary source code file for *Tank War*, which contains (among other things) that game loop. Scroll down to `main` and add the call to `loadsounds`, as indicated in bold.

```
//main function
void main(void)
{
    int anim;

    //initialize the game
    allegro_init();
    install_keyboard();
    install_timer();
    srand(time(NULL));
    setupscreen();
    setuptanks();
    loadsprites();
    loadsounds();
```

Next, scroll down a little bit past the section of code that loads the Mappy file and add the new code shown in bold. This code initializes the joystick(s) and sets the input delay variables.

```
    //load the Mappy file
    if (MapLoad("map3.fmp"))
    {
        allegro_message ("Can't find map3.fmp");
```

```
    return;
}

//set palette
MapSetPal8();

//install the joystick handler
install_joystick(JOY_TYPE_AUTODETECT);
poll_joystick();

//setup input delays
key_count[0] = 0;
key_delay[0] = 2;
key_count[1] = 0;
key_delay[1] = 2;
```

Now, scroll down to the end of the game loop and insert or change the following lines of code after the call to `getinput`, as shown in bold. You'll insert a call to `readjoysticks` and modify the `rest` function call to increase the delay a bit (because the delay in `getinput` was removed).

```
    //check for keypresses
    if (keypressed())
        getinput();

    readjoysticks();

    //slow the game down
    rest(30);
}
```

Now let's clean up the memory that was used by these new changes. Scroll down a little bit more and insert the following code after the call to `MapFreeMem`, as shown in bold.

```
//free the MappyAL memory
MapFreeMem();

//free the samples
for (n=0; n<NUM_SOUNDS; n++)
    destroy_sample(sounds[n]);

//remove the sound driver
remove_sound();

//remove the joystick driver
```

```
    remove_joystick();

    return;
}
END_OF_MAIN();
```

Final Comments about Tank War

Figure 15.3 shows the final version of *Tank War*. It's been a long haul, and you've seen the game grow from a meager vector game to the current incarnation with animated sprites and scrolling backgrounds. Here's a list of the features of the final version of the game:

- Two-player split-screen gameplay
- A scrolling battlefield
- Support for new maps created with Mappy
- Advanced update code to show all the action in both windows
- Keyboard and dual joystick support
- Sixty-four animated frames for each tank
- Support for stereo sound cards
- Numerous sound effects to enhance gameplay
- Support for maps with up to 30,000 tiles
- A battlefield that can be up to 5,500×5,500 pixels in size
- Ability to run on Windows, Linux, Mac OS X, and many other systems

Figure 15.3 The final version of *Tank War*

Well, what are you waiting for? Go ahead and get started on the tenth revision to *Tank War*!

Summary

This chapter provided an introduction to the sound support routines provided by Allegro for including sound effects in a game. Allegro provides an interface to the underlying operating system (with support for DirectSound) that, along with a software sound mixer, provides a consistent level of functionality and performance from one computer system to another. In this chapter, you learned how to initialize the sound system, load a WAV file, and play back the WAV file with or without looping. You were also provided with an example that demonstrated Allegro's automatic mixing of samples that are played. In a nutshell, it requires very little effort to play a sound effect, and mixing is handled automatically, allowing you to focus on gameplay rather than the mechanics of an advanced sound system.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the name of the function that initializes the Allegro sound system?
   A. `install_sound`
   B. `init_sound`
   C. `initialize_sound_system`
   D. `init_snd`

2. Which function can you use to play a sound effect in your own games?
   A. `start_playback`
   B. `play_sound`
   C. `play_sample`
   D. `digi_snd_play`

3. What is the name of the function that specifically loads a RIFF WAV file?
   A. `load_riff`
   B. `load_wav`
   C. `load_wave`
   D. `load_riff_wav`

4. Which function can be used to change the frequency, volume, panning, and looping properties of a sample?
   A. `modify_sample`
   B. `change_sample`
   C. `alter_sample`
   D. `adjust_sample`

5. What function would you use to shut down the Allegro sound system?
   A. `uninstall_sound`
   B. `remove_sound`
   C. `close_sound`
   D. `close_sound_system`

6. Which function provides the ability to change the overall volume of sound output?
   A. `set_volume`
   B. `change_volume`
   C. `fix_volume`
   D. `set_vol`

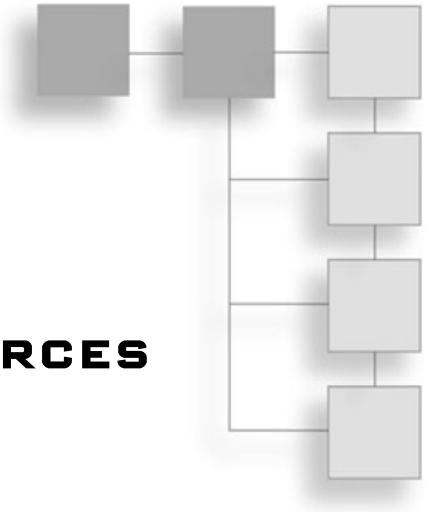
7. What is the name of the function used to stop playback of a sample?
   A. `stop_playback`
   B. `stop_playing`
   C. `halt_playback`
   D. `stop_sample`

8. Within what range must a panning value remain?
   A. –32,768 to 32,767
   B. 0 to 65,536
   C. 1 to 100
   D. 0 to 255

9. What parameter should you pass to `install_sound` to initialize the standard digital sound driver?
   A. `SND_AUTODETECT`
   B. `SND_AUTODETECT_DIGITAL`
   C. `DIGI_AUTODETECT`
   D. `DIGI_AUTODETECT_SOUND`

10. What is the name of the function that plays a sample through the sound mixer?
    A. `start_playback`
    B. `play_sample`
    C. `play_sample_mix`
    D. `start_mix_playback`

CHAPTER 16



Using Datafiles to Store Game Resources

Suppose you have written one of the greatest new games to come out of the indie market in years, and you are chomping at the bit to get the game out into the world. You poured your blood, sweat, and tears into the game and it has cost you every moment of your free time for three years. Your friends and relatives have abandoned you, and you haven't emerged from your room in months, focused on and dedicated to one goal—making this the most unbelievable game ever. You have come up with a new gaming technology that you think will create a whole new genre. It's the next *Doom* or *Warcraft*.

There's just one problem. You have spent so much time getting the game running and polished that you have paid no attention to the game's resources. Now, faced with distribution, you are struggling to come up with a plan for protecting your game's resources—all the amazing artwork (that you had commissioned from a professional artist), sound effects and music (commissioned from a sound studio), and professional voice acting at various parts throughout the game. You have valuable assets to protect. You have thought about finding a ZIP decompression library, but are not looking forward to the problems associated with temp files.

Luckily for you, you planned ahead and developed your new cutting-edge game with Allegro. And it just so happens that Allegro has support for data files to store all of your game resources with encryption and compression. Best of all, it's extremely easy to use, and you don't need to deal with temp files. Ready to learn how to do this?

Here is a breakdown of the major topics in this chapter:

- Understanding Allegro datafiles
- Creating Allegro datafiles
- Using Allegro datafiles
- Testing Allegro datafiles

Understanding Allegro Datafiles

Allegro data files (*datafiles*) are similar to ZIP archive files in that they can contain multiple files of different types and sizes, with support for encryption and compression. However, Allegro datafiles differ from the files of general-purpose archival programs in that Allegro is geared entirely to store game resources. Allegro datafiles use the LZSS compression algorithm when compression is used.

Datafiles are created by the Allegro Datafile archiving utility and have a .dat extension. They can contain bitmaps, sounds, FLI animations, Mappy levels, text files, and any other type of file or binary data that your game will need. You can distribute your game with a single executable and a single datafile.

One of the best things about datafiles is that, because they are so easy to create and use, you can use a datafile for any program you write, not just games. This really adds a strong degree of appeal to Allegro even for general-purpose programming, developing command-line utilities and support programs in addition to full-blown game editors and similar programs. Instead of distributing a background image, sprite image, and sound file in a small game, simply bundle it all together in a datafile and send it off to your friends with the program file. In other words, you don't have to reserve datafile use only for big projects; you can feel free to use datafiles frequently even on non-game projects.

Datafiles use a struct to keep track of their resources. The struct looks like this:

```
typedef struct DATAFILE
{
   void *dat;   //pointer to the actual data
   int type;    //type of the data
   long size;   //size of the data in bytes
   void *prop;  //list of object properties
} DATAFILE;
```

When you refer to an object in a datafile, you must use a `DATAFILE` struct to get at the resource. Usually you will not need to be concerned with anything other than the `dat` member variable, which is a `void` pointer to the object in the file (or in memory after the datafile has been loaded).

Would you like a quick example? Although I haven't covered the `load_datafile` function yet, here is an example of how you might load a datafile into memory and then grab a sprite directly out of the datafile:

```
DATAFILE *data = load_datafile("game.dat");
draw_sprite(screen, data[PLAYER_SPRITE].dat, x, y);
```

If you want to identify a resource by type (for instance, to verify that the resource is a valid type), you can use the `type` member variable of the `DATAFILE` struct. Table 16.1 provides a list of the various types of objects that can be stored in a datafile.

Table 16.1 Datafile Object Types and Formats

Data Type	Format	Description
DAT_FILE	"FILE"	Nested data file
DAT_DATA	"DATA"	Block of binary data (miscellaneous)
DAT_FONT	"FONT"	Font object
DAT_SAMPLE	"SAMP"	Sound sample structure
DAT_MIDI	"MIDI"	MIDI file
DAT_PATCH	"PAT"	GUS patch file
DAT_FLI	"FLIC"	FLI animation
DAT_BITMAP	"BMP"	BITMAP structure
DAT_RLE_SPRITE	"RLE"	RLE_SPRITE structure
DAT_C_SPRITE	"CMP"	Linear compiled sprite
DAT_XC_SPRITE	"XCMP"	Mode-X compiled sprite
DAT_PALETTE	"PAL"	Array of 256 RGB structures
DAT_END	N/A	Special flag to mark the end of the data list

Creating Allegro Datafiles

Before you can practice using datafiles, you need to learn how to create and manage them. This will also give you a heads-up on extracting the game resources from other people's games that use Allegro datafiles! Not that I would condone the theft of artwork…but it is interesting to see how some people develop their artwork.

Allegro comes with a command-line utility program called dat.exe that you will use to create and manage your datafiles. The Allegro utilities are located in the tools folder inside the root Allegro folder, wherever you installed it (based on the sources). If you have extracted Allegro to your root drive folder, then the dat.exe program is likely to be found in \allegro\tools. You will need to open a command prompt or shell and change to that folder to run the program. Alternatively, you might want to just add \allegro\tools to your system path. In Windows, you would do that by typing

```
path=C:\allegro\tools;%path%
```

After you do this, you will be able to maintain your datafiles from any folder on the hard drive because dat.exe will be included in the path. Here is the output from dat.exe if you run it with no parameters:

```
Datafile archiving utility for Allegro 4.0.3, MSVC.s
By Shawn Hargreaves, 2003

Usage: dat [options] filename.dat [names]

Options:
    '-a' adds the named files to the datafile
    '-bpp colordepth' grabs bitmaps in the specified format
    '-c0' no compression
    '-c1' compress objects individually
    '-c2' global compression on the entire datafile
    '-d' deletes the named objects from the datafile
    '-dither' dithers when reducing color depths
    '-e' extracts the named objects from the datafile
    '-g x y w h' grabs bitmap data from a specific grid location
    '-h outputfile.h' sets the output header file
    '-k' keeps the original file names when grabbing objects
    '-l' lists the contents of the datafile
    '-m dependencyfile' outputs makefile dependencies
    '-o output' sets the output file or directory when extracting data
    '-p prefixstring' sets the prefix for the output header file
    '-pal objectname' specifies which palette to use
    '-s0' no strip: save everything
    '-s1' strip grabber specific information from the file
    '-s2' strip all object properties and names from the file
    '-t type' sets the object type when adding files
    '-transparency' preserves transparency through color conversion
    '-u' updates the contents of the datafile
    '-v' selects verbose mode
    '-w' always updates the entire contents of the datafile
    '-007 password' sets the file encryption key
    'PROP=value' sets object properties
```

I'm not going over all these options; consider it your homework for the day. The really important thing to know about the dat.exe syntax is the usage.

```
Usage: dat [options] filename.dat [names]
```

When you run dat.exe, first you must include any options, then the name of the datafile, followed by the files you want to add to (or extract from) the datafile. Looking through the options, I see that `-a` is the parameter that adds files to a datafile. But you must also use the `-t` option to tell `dat` what kind of file you are adding. Go ahead and try it. Locate a bitmap file, change to that directory from the command prompt (or shell), and adapt the following command to suit the bitmap file you intend to add to the datafile.

```
dat -a -t BMP -bpp 16 test.dat back.bmp
```

Do you see the `-bpp 16` parameter? You must specify the color depth of the bitmaps you are adding to the datafile or it will treat them as 8-bit images (one byte per pixel). I have used the `-bpp 16` parameter to instruct the `dat` program to store the file as a 16-bit bitmap. The output from `dat` should look something like this:

```
test.dat not found: creating new datafile
Inserting back.bmp -> BACK_BMP
Writing test.dat
```

Now you can find out whether the bitmap image is actually stored inside the test.dat file.

```
dat -l test.dat
```

You should see a result that looks something like this:

```
Reading test.dat
- BMP  - BACK_BMP                - bitmap (640x480, 16 bit)
```

Great, it worked! Now there's just one problem. I see from the options list that I can add compression to the datafile using the `-c2` option, so I'd like to reduce the size of the file. Here is the command to do that:

```
dat -c2 test.dat
```

The output looks like this:

```
Reading test.dat
Writing test.dat
```

I see that the file has been reduced from 900 KB to about 100 KB. Perfect!

Now I want to another file (a sprite), and then I'll demonstrate how to get to these objects from an Allegro program.

```
dat -a -t BMP -bpp 16 test.dat ship.bmp
```

results in this output, so I know it's good:

```
Reading test.dat
Inserting ship.bmp -> SHIP_BMP
Writing test.dat
```

Now that you have added two files to the datafile, take a peek inside:

```
dat -l test.dat
```

produces this output:

```
Reading test.dat
```

```
- BMP  - BACK_BMP                - bitmap (640x480, 16 bit)
- BMP  - SHIP_BMP                - bitmap (111x96, 16 bit)
```

If you take a look at the file size, you'll see that it is still compressed. Trying to compress it again results in the same file size, so it's apparent that once `-c2` has been applied to a datafile, compression is then applied to any new files added to it.

I should also point out that you should reference the objects in the file in the order they are displayed using `dat -l test.dat`. You can reference the back.bmp file using array index 0, explode.wav using array index 1, and so on.

The `dat` tool is able to generate a header file containing the datafile definition of values using the `-h` option.

```
dat test.dat -h defines.h
```

produces a file that looks like this:

```
/* Allegro datafile object indexes, produced by dat v4.0.3, MSVC.s */
/* Datafile: test.dat */
/* Date: Thu Apr 15 20:49:59 2004 */
/* Do not hand edit! */

#define BACK_BMP                    0         /* BMP  */
#define SHIP_BMP                    1         /* BMP  */
```

It is best to include this header file directly in your project and not edit it manually (although for the simple demonstration program later in the chapter, I have simply pasted the defines into the program).

Using Allegro Datafiles

You have learned some details about what datafiles are made of and how to create and update them. Now it's time to put them to the test in a real Allegro program that will load the datafile and retrieve game objects directly out of the datafile. First you need to go over the datafile functions to learn how to manipulate a datafile with source code.

Loading a Datafile

The `load_datafile` function loads a datafile into memory and returns a pointer to it or NULL. If the datafile has been encrypted, you must first use the `packfile_password` function to set the appropriate key. See grabber.txt for more information. If the datafile contains true color graphics, you must set the video mode or call `set_color_conversion()` before loading the datafile.

```
DATAFILE *load_datafile(const char *filename);
```

note

If you are programming in C++, you will get an error unless you include a cast for the type of object being referenced in the datafile. Here is an example:

```
draw_sprite(screen, (BITMAP *)data[SPRITE].dat, x, y);
```

Unloading a Datafile

The `unload_datafile` function frees all the objects in a datafile and removes the datafile from memory.

```
void unload_datafile(DATAFILE *dat);
```

Loading a Datafile Object

The `load_datafile_object` will load a specific object from a datafile, returning the object as a single `DATAFILE *` pointer (instead of the usual array).

```
DATAFILE *load_datafile_object(const char *filename, const char *objectname);
```

Here is an example:

```
sprite = load_datafile_object("datafile.dat", "SPRITE_BMP");
```

Unloading a Datafile Object

The `unload_datafile_object` function will free an object that was loaded with the `load_datafile_object` function.

```
void unload_datafile_object(DATAFILE *dat);
```

Finding a Datafile Object

The `find_datafile_object` function searches an opened datafile for an object with the specified name, returning a pointer to the object or NULL.

```
DATAFILE *find_datafile_object(const DATAFILE *dat, const char *objectname);
```

Testing Allegro Datafiles

Now that you have a basic understanding of how datafiles are created and what the data inside a datafile looks like, it's time to learn how to read a datafile in an Allegro program. I have written a short program that loads the test.dat file you created earlier in this chapter and displays the back.bmp and ship.bmp files stored in the datafile. You should be able

to use this basic example (along with the list of data file object types) to use any other type of file in your programs (such as samples or Mappy files). Figure 16.1 shows the output of the *TestDat* program.

Figure 16.1 The *TestDat* program demonstrates how to read bitmaps from an Allegro datafile.

```
#include <allegro.h>

#define MODE GFX_AUTODETECT_WINDOWED
#define WIDTH 640
#define HEIGHT 480
#define WHITE makecol(255,255,255)

//define objects in datafile
#define BACK_BMP 0
#define SHIP_BMP 1


void main(void)
{
    DATAFILE *data;
    BITMAP *sprite;
```

```
    //initialize the program
    allegro_init();
    install_keyboard();
    install_timer();
    set_color_depth(16);
    set_gfx_mode(MODE, WIDTH, HEIGHT, 0, 0);
    text_mode(-1);

    //load the datafile
    data = load_datafile("test.dat");

    //blit the background image using datafile directly
    blit(data[BACK_BMP].dat, screen, 0, 0, 0, 0, WIDTH-1, HEIGHT-1);

    //grab sprite and store in separate BITMAP
    sprite = (BITMAP *)data[SHIP_BMP].dat;
    draw_sprite(screen, sprite, WIDTH/2-sprite->w/2,
        HEIGHT/2-sprite->h/2);

    //display title
    textout(screen,font,"TestDat Program (ESC to quit)",0,0,WHITE);

    //pause
    while(!key[KEY_ESC]) { }

    //remove datafile from memory
    unload_datafile(data);

    allegro_exit();
}
END_OF_MAIN();
```

Summary

This chapter provided an introduction to Allegro datafiles and showed you how to create them, modify them, and read them into an Allegro program or game. Datafiles make it much easier to distribute your games to others because you need only include the datafile and executable program file. Datafiles can contain any type of file, but some items are predefined so they are recognized and handled properly by Allegro.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the shorthand term for an Allegro data file?
   A. datafile
   B. datfile
   C. data file
   D. ADF

2. What compression algorithm does Allegro use for compressed datafiles?
   A. LZSS
   B. LZH
   C. ZIP
   D. RAR

3. What is the command-line program that is used to manage Allegro datafiles?
   A. data.exe
   B. datafile.exe
   C. datafile.exe
   D. dat.exe

4. What is the Allegro datafile object struct called?
   A. `DATA_FILE`
   B. `DATAFILE`
   C. `DAT_FILE`
   D. `AL_DATFILE`

5. What function is used to load a datafile into memory?
   A. `open_data_file`
   B. `load_dat`
   C. `load_datfile`
   D. `load_datafile`

6. What is the data type format shortcut string for bitmap files?
   A. `BITMAP_IMAGE`
   B. `BITMAP`
   C. `BMP`
   D. `DATA_BITMAP`

7. What is the data type constant for wave files, defined by Allegro for use in reading datafiles?
   A. `DAT_RIFF_WAV`
   B. `DAT_WAVE`
   C. `DAT_SAMPLE`
   D. `DAT_SOUND`

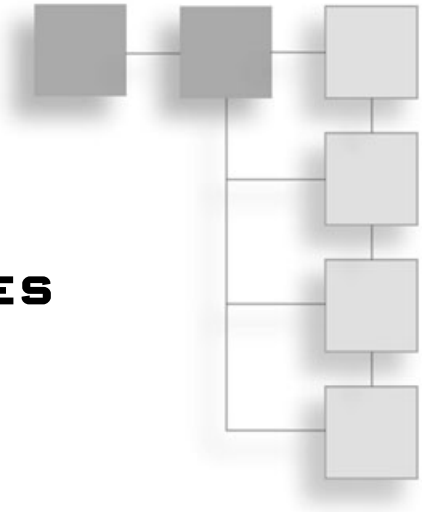
8. What is the `dat` option to specify the type of file being added to the datafile?
   A. `-t <type>`
   B. `-a <type>`
   C. `-d <type>`
   D. `-s <type>`

9. What is the `dat` option to specify the color depth of a bitmap file being added to the datafile?
   A. `-c <depth>`
   B. `-d <depth>`
   C. `-bpp <depth>`
   D. `-color <depth>`

10. Which function loads an individual object from a datafile?
    A. `load_data_object`
    B. `load_object_file`
    C. `load_datafile`
    D. `load_datafile_object`

This page intentionally left blank

CHAPTER 17



Playing FLIC Movies

FLI is an animation format developed by Autodesk for creating and playing computer-generated animations at high resolutions using Autodesk Animator, while the FLC format was the standard format used in Autodesk Animator Pro. These two formats (FLI and FLC) are both referred to as the FLIC format. The original FLI format was limited to a resolution of 320×200, while FLC provided higher resolutions and file compression. This chapter focuses on the functions built into Allegro for reading and playing FLIC movies, which are especially useful as cut-scenes within a game or as the opening video often presented as a game begins.

Here is a breakdown of the major topics in this chapter:

- Playing FLI animation files
- Loading FLIs into memory

Playing FLI Animation Files

Animated or rendered movies are often used in games to fill in a cut-scene at a specified point in the game or to tell a story as the game starts. Of course, you can use an animation for any purpose within a game using Allegro's built-in support for FLI loading and playback (both from memory and from disk file). The only limitation is that you can only play one FLI at a time. If you need multiple animations to run at the same time, I recommend converting the FLI file to one or more bitmap images and treating the movie as an animated sprite—although I'll leave implementation of that concept up to you. (First you would need to convert the FLI to individual bitmap images.)

The easiest way to play an FLI animation file with Allegro is by using the `play_fli` function, which simply plays an FLI or FLC file directly to the screen or to another destination bitmap.

```
int play_fli(const char *filename, BITMAP *bmp, int loop,
        int (*callback)());
```

The first parameter is the FLI/FLC file to play; the second parameter is the destination bitmap where you would like the animation to play; and the third parameter, `loop`, determines whether the animation is looped at the end (1 is looped, 0 is not). In practice, however, you will want to intercept playback in the callback function and pass a return value of 1 from the callback to stop playback.

As you can see from the function definition, `play_fli` supports a callback function. The purpose for this is so that your game can continue running while the FLI is played; otherwise, playback would run without interruption. The callback function is very simple—it returns an `int` but accepts no parameters.

When you are playing back an animation file, keep in mind that `play_fli` draws each frame at the upper-left corner of the destination bitmap (which is usually the screen). If you want more control over the playback of an FLI, you have two options. First, you can tell `play_fli` to draw the frames on a memory bitmap and then draw that bitmap to the screen yourself. (See the following section on using the callback function.)

The FLI Callback Function

The callback function makes it possible to do other things inside your program after each frame of the animation is displayed. Note that you should return from the callback function as quickly as possible or the playback timing will be off. When you want to use a callback function, simply declare a function like this:

```
int fli_callback(void)
{
}
```

You can then use `play_fli` to start playback of an FLI file, including the `fli_callback` function.

```
play_fli("particles.fli", screen, 1, fli_callback);
```

The PlayFlick Program

The `play_fli` function is not really very useful if you don't also use the callback function. I have written a test program called *PlayFlick* that demonstrates how to use `play_fli` along with the callback to play an animation with logistical information printed after each frame of the FLI is displayed on the screen. Figure 17.1 shows the output from the *PlayFlick* program.

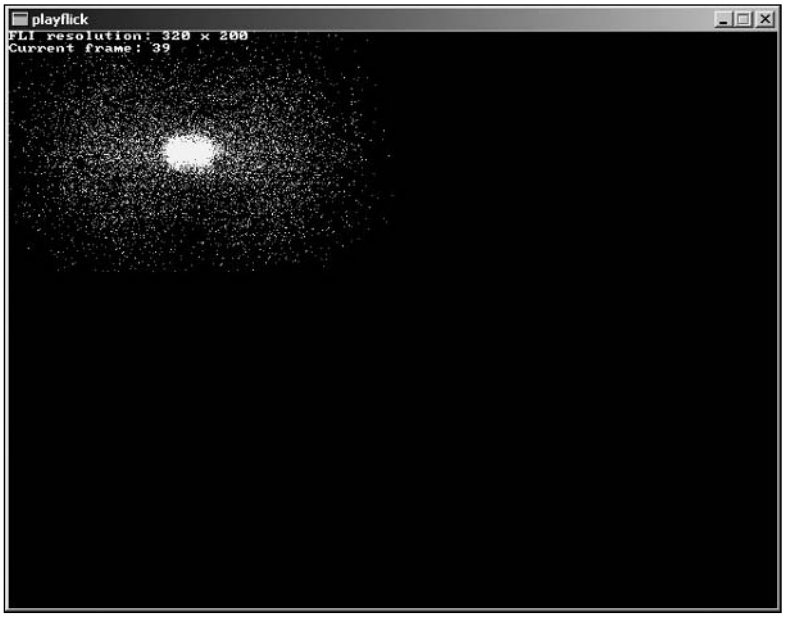


Figure 17.1 The *PlayFlick* program demonstrates how to play an Autodesk Animator FLI/FLC file.

If you are writing this program from scratch (as follows), you will of course need an FLI file to use for testing. You can copy one of the FLI files off the CD-ROM from the folder for this chapter and project, \chapter17\playflick. The sample file is called particles.fli, and there are several other sample FLI files in other project folders for this chapter.

```
#include <stdio.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

int ret;

int fli_callback(void)
{
    //display some info after each frame
    textprintf(screen, font, 0, 0, WHITE,
        "FLI resolution: %d x %d", fli_bitmap->w, fli_bitmap->h);
    textprintf(screen, font, 0, 10, WHITE,
        "Current frame: %2d", fli_frame);
```

```
    //ESC key stops animation
    if (key[KEY_ESC])
        return 1;
    else
        return 0;
}

void main(void)
{
    //initialize Allegro
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_timer();
    install_keyboard();

    //play fli with callback
    play_fli("particles.fli", screen, 1, fli_callback);

    //time to leave
    allegro_exit();
}
END_OF_MAIN();
```

Playing an FLI from a Memory Block

Allegro provides you with a way to play a raw FLI file that has been mass copied from disk into memory with header and all. The `play_memory_fli` function will play a memory FLI as if it were a disk file. The FLI routines must still work with only one file at a time, even if that file was loaded into a memory block (which you must create with `malloc` and read into memory using your own file input code). You would also use this function when you have stored an FLI inside a datafile. (For more information about datafiles, refer to Chapter 16.)

```
int play_memory_fli(const void *fli_data, BITMAP *bmp,
        int loop, int (*callback)());
```

Loading FLIs into Memory

The two functions covered thus far were designed for simple FLI playback with little to no control over the frames inside the animation. Fortunately, Allegro provides a low-level interface for FLI playback, allowing you to read an FLI file and manipulate it frame by frame, adjusting the palette and blitting the frame to the screen manually.

Opening and Closing FLI Files

To open an FLI file for low-level playback, you'll use the `open_fli` function.

```
int open_fli(const char *filename);
```

If you are using a datafile (or you have loaded an entire FLI file into memory byte for byte), you'll use the `open_memory_fli` function to open it for low-level access.

```
int open_memory_fli(const void *fli_data);
```

If the file was opened successfully, a value of `FLI_OK` will be returned; otherwise, `FLI_ERROR` will be returned by these functions. Information about the current FLI is held in global variables, so you can only have one animation open at a time.

note

The FLI routines make use of interrupts, so you must install the timer by calling `install_timer` at the start of the program.

After you have finished playing an FLI animation, you can close the file by calling `close_fli`.

```
void close_fli();
```

Processing Each Frame of the Animation

After you have opened the FLI file, you are ready to begin handling the low-level processing of the animation playback. Allegro provides a number of functions and global variables for dealing with each animation frame; you'll see that they are easy to use in practice.

For starters, take a look at the `next_fli_frame` function.

```
int next_fli_frame(int loop);
```

This function reads the next frame of the current animation file. If `loop` is set, the player will cycle when playback reaches the end of the file; otherwise, the function will return `FLI_EOF`. If no error occurs, this function will return `FLI_OK`, but if an error has occurred, it will return `FLI_ERROR` or `FLI_NOT_OPEN`. One useful return value is `FLI_EOF`, which tells you that the playback has reached the last frame of the file.

What about drawing each frame image? The frame is read into the global variables `fli_bitmap` (which contains the current frame image) and `fli_palette` (which contains the current frame's palette).

```
extern BITMAP *fli_bitmap;
extern PALETTE fli_palette;
```

Even if you are running a program in a high-color or true-color video mode, you will need to set the current palette to render the animation frames properly. (This at least applies to 8-bit FLI files; FLC files might not need a palette.)

After each call to `next_fli_frame`, Allegro sets a global variable indicating the current frame in the animation sequence of the FLI file, called `fli_frame`.

```
extern int fli_frame;
```

The current frame is helpful to know, but it doesn't help with timing, which will differ from one FLI file to another. Allegro takes care of the problem by automatically incrementing a global variable called `fli_timer` whenever a new frame should be displayed. This works regardless of the computer's speed because it is handled by an interrupt. It is important to pay attention to timing unless you are only concerned with the image of each frame and not playback speed.

```
extern volatile int fli_timer;
```

Each time you call `next_fli_frame`, the `fli_timer` variable is decremented, so if playback is in sync with timing, this variable will always be 0 unless a new frame is ready to be displayed. This makes it easy to determine when each frame should be drawn.

The LoadFlick Program

To demonstrate the low-level FLI animation routines, I've written a short program called *LoadFlick*. The output from this program is shown in Figure 17.2. *LoadFlick* pretty much demonstrates everything you need to know about the low-level FLI routines, including how to load an FLI file, keep track of each frame, manage timing, and blit the image to the screen.

```
#include <stdio.h>
#include "allegro.h"

#define WHITE makecol(255,255,255)

int ret;

void main(void)
{
    //initialize Allegro
    allegro_init();
    set_color_depth(16);
    set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);
    install_timer();
    install_keyboard();
```

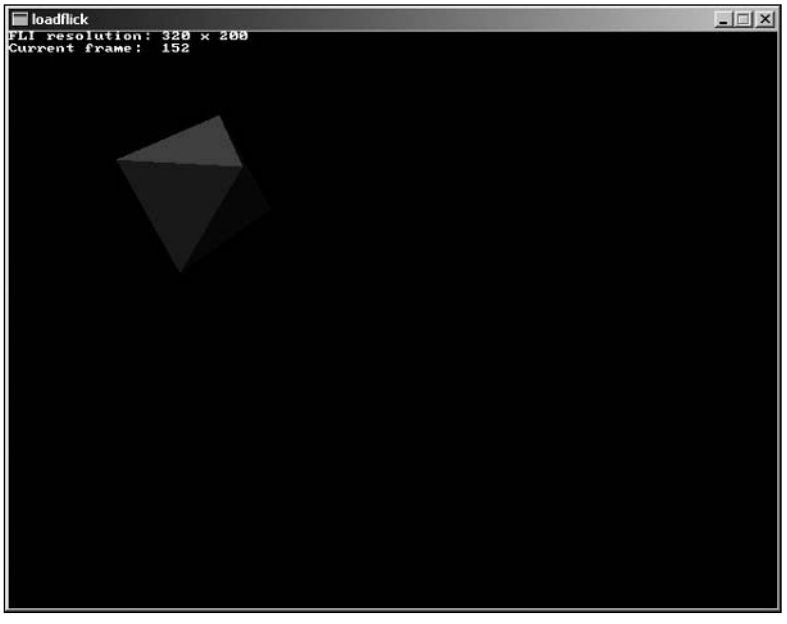


Figure 17.2 The *LoadFlick* program handles each frame of the FLI animation individually.

```
//load the fli movie file
ret = open_fli("octahedron.fli");
if (ret != FLI_OK)
{
    textout(screen, font, "Error loading octahedron.fli",
        0, 30, WHITE);
    readkey();
    return;
}

//display movie resolution
textprintf(screen, font, 0, 0, WHITE,
    "FLI resolution: %d x %d", fli_bitmap->w, fli_bitmap->h);

//main loop
while (!key[KEY_ESC])
{
    //is it time for the next frame?
    if (fli_timer)
    {
```

```
            //open the next frame
            next_fli_frame(1);

            //adjust the palette
            set_palette(fli_palette);

            //copy the FLI frame to the screen
            blit(fli_bitmap, screen, 0, 0, 0, 30,
                fli_bitmap->w, fli_bitmap->h);

            //display current frame
            textprintf(screen, font, 0, 10, WHITE,
                "Current frame: %4d", fli_frame);
        }
    }

    //remove fli from memory
    close_fli();

    //time to leave
    allegro_exit();
}
END_OF_MAIN();
```

The ResizeFlick Program

Let's do something fun just to see how useful the low-level FLI routines can be when you want full control over each frame in the animation. The *ResizeFlick* program is similar to *LoadFlick* in that it opens an FLI into memory before playback. The difference in this new program is that the resulting FLI frames are resized to fill the screen (using a proper ratio for the height). Note that the FLI file must be in landscape orientation—wider than it is tall—or the bottom of each frame image might be cropped. It's best to use FLI files with a resolution that is similar to one of the common screen resolutions, such as 320×240, 640×480, and so on.

Figure 17.3 shows the *ResizeFlick* program running with a short animation of a jet aircraft (the U.S. Air Force SR-71 Blackbird). Note the black area at the bottom of the screen—this is due to the fact that the original FLI animation was 320×200, so when it was scaled there were pixels left blank on the bottom. If you want to truly fill the entire screen, you can do away with the `width` and `height` variables and simply pass `SCREEN_W-1` and `SCREEN_H-1` as the last two parameters of `stretch_blit`, which will cause the FLI to be played back in true full-screen mode (although with image artifacts if the scaling is not a multiple of the original resolution).

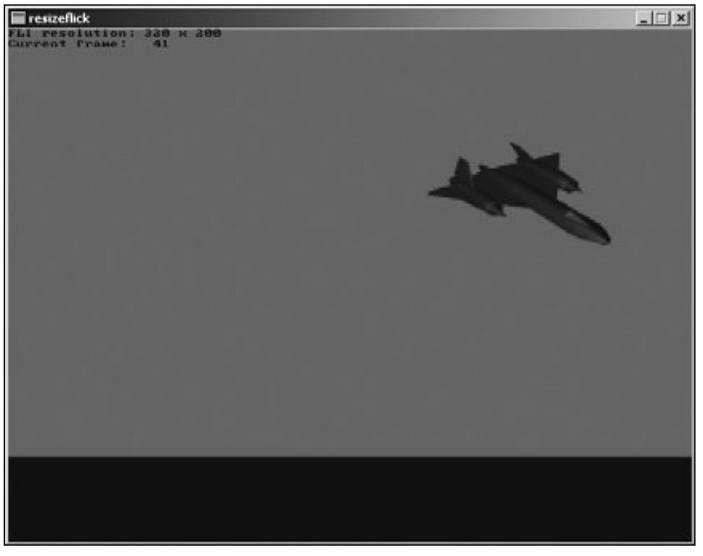


Figure 17.3 The *ResizeFlick* program shows how to play an FLI at any scaled resolution.

```
#include "allegro.h"

#define WHITE makecol(255,255,255)
#define BLACK makecol(0,0,0)

int ret,width,height;

void main(void)
{
     //initialize Allegro
     allegro_init();
     install_timer();
     install_keyboard();
     text_mode(-1);

     //set video mode--color depth defaults to 8-bit
     set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0);

     //load the fli movie file
     ret = open_fli("sr-71.fli");
     if (ret != FLI_OK)
     {
          textout(screen, font, "Error loading sr-71.fli",
```

```
            0, 30, WHITE);
        readkey();
        return;
    }

    //main loop
    while (!key[KEY_ESC])
    {
        //is it time for the next frame?
        if (fli_timer)
        {
            //open the next frame
            next_fli_frame(1);

            //adjust the palette
            set_palette(fli_palette);

            //calculate scale
            width = SCREEN_W;
            height = fli_bitmap->h * (SCREEN_W / fli_bitmap->w);

            //draw scaled FLI (note: screen must be in 8-bit mode)
            stretch_blit(fli_bitmap, screen, 0, 0, fli_bitmap->w,
                fli_bitmap->h, 0, 0, width, height);

            //display movie resolution
            textprintf(screen, font, 0, 0, BLACK,
                "FLI resolution: %d x %d", fli_bitmap->w, fli_bitmap->h);

            //display current frame
            textprintf(screen, font, 0, 10, BLACK,
                "Current frame: %4d", fli_frame);

        }
    }

    //remove fli from memory
    close_fli();

    //time to leave
    allegro_exit();
}
END_OF_MAIN();
```

Summary

This chapter provided an overview of the FLIC animation routines available with Allegro. You learned how to play an FLI/FLC file directly from disk as well as how to load an FLI/FLC file into memory and manipulate the animation frame by frame. There were three sample programs in this chapter to demonstrate the routines available for playback of an FLIC file, including a program at the end of the chapter that displayed a movie scaled to the entire screen.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Which company developed the FLI/FLC file format?
   A. Autodesk
   B. Borland
   C. Microsoft
   D. Bungie

2. Which product first used the FLI format?
   A. 3D Studio Max
   B. WordPerfect
   C. Animator
   D. PC Paintbrush

3. Which product premiered the more advanced FLC format?
   A. Animator Pro
   B. PC Animation
   C. Dr. Halo
   D. CorelDRAW

4. What is the common acronym used to describe both FLI and FLC files?
   A. FLICK
   B. FLICKS
   C. FLI/C
   D. FLIC

5. Which function plays an FLIC file directly?
   A. `play_fli`
   B. `direct_play`
   C. `play_animation`
   D. `play_flic`

6. How many FLIC files can be played back at a time by Allegro?
   A. 1
   B. 2
   C. 3
   D. 4

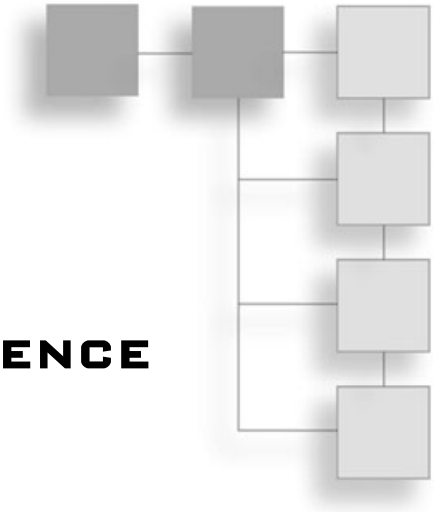
7. Which function loads an FLIC file for low-level playback?
   A. `load_fli`
   B. `read_fli`
   C. `open_fli`
   D. `shoo_fli`

8. Which function moves the animation to the next frame in an FLIC file?
   A. `next_fli_frame`
   B. `get_next_frame`
   C. `move_frame`
   D. `next_fli`

9. What is the name of the variables used to set the timing of FLIC playback?
   A. `flic_frames`
   B. `playback_timer`
   C. `fli_playback`
   D. `fli_timer`

10. What is the name of the variable that contains the bitmap of the current FLIC frame?
    A. `fli_frame`
    B. `fli_bitmap`
    C. `fli_image`
    D. `current_fli`

CHAPTER 18



Introduction to Artificial Intelligence

Probably the thing I dislike most about some games is how the computer cheats. I'm playing my strategy game and I have to spend 10 minutes finding their units while they automatically know where mine are, which type they are, their energies, and so on. It's not the fact that they cheat to make the game harder, it's the fact that they cheat because the artificial intelligence is very weak. The computer adversary should know just about the same information as the player. If you look at a unit, you don't see their health, their weapons, and their bullets. You just see a unit and, depending on your units, you respond to it. That's what the computer should do; that's what artificial intelligence is all about.

In this chapter I will first give you a quick overview of several types of artificial intelligence, and then you will see how you can apply one or two to games. In this chapter, I'm going to go against the norm for this book and explain the concepts with little snippets of code instead of complete programs. The reason I'm doing this is because the implementation of each field of artificial intelligence is very specific, and where is the fun in watching a graph give you the percentage of the decisions if you can't actually see the bad guy hiding and cornering you? Complete examples would basically require a complete game! For this reason, I will go over several concrete artificial intelligence examples, giving only the theory and some basic code for the implementation, and it will be up to you to choose the best implementation for what you want to do.

Here is a breakdown of the major topics in this chapter:

- Understanding the various fields of artificial intelligence
- Using deterministic algorithms
- Recognizing finite state machines
- Identifying fuzzy logic

- Understanding a simple method for memory
- Using artificial intelligence in games

The Fields of Artificial Intelligence

There are many fields of artificial intelligence; some are more game-oriented and others are more academic. Although it is possible to use almost any of them in games, there are a few that stand out, and they will be introduced and explained in this section.

Expert Systems

Expert systems solve problems that are usually solved by specialized humans. For example, if you go to a doctor, he will analyze you (either by asking you a set of questions or doing some analysis himself), and according to his knowledge, he will give you a diagnosis.

An expert system could be the doctor if it had a broad enough knowledge base. It would ask you a set of questions, and depending on your answers, it would consult its knowledge base and give you a diagnosis. The system checks each of your answers with the possible answers in its knowledge base, and depending on your answer, it asks you other questions until it can easily give you a diagnosis.

For a sample knowledge tree, take a look at Figure 18.1. As you can see, a few questions would be asked, and according to the answers, the system would follow the appropriate tree branch until it reached a leaf.

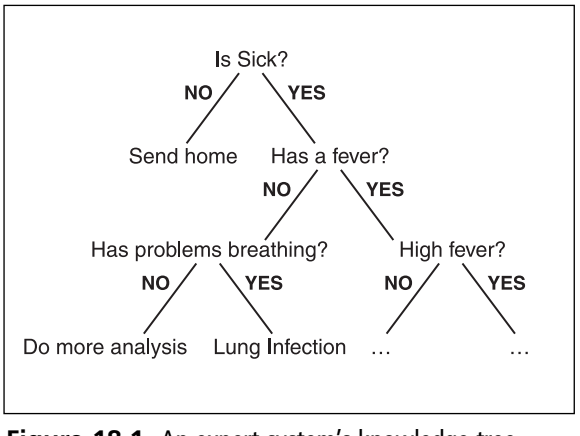


Figure 18.1 An expert system's knowledge tree

A very simple expert system for a doctor could be something like the following code. Note that this is all just pseudo-code, based on a fictional scripting language, and it *will not compile* in a compiler, such as Dev-C++ or Visual C++. This is not intended to be a functional example, just a glimpse at what an expert system's scripting language might look like.

```
Answer = AskQuestion ("Do you have a fever?");
if (Answer == YES)
  Answer = AskQuestion ("Is it a high fever (more than 105.8 F)?");
```

```
  if (Answer == YES)
      Solution = "Go to a hospital now!";
  end if
  Is Sick?
  NO YES
  Has a fever?
  NO YES
  Has problems breathing?
  NO YES
  High fever?
  NO YES
  Send home
  . . . Lung Infection Do more analysis . . .
else
  Answer = AskQuestion ("Do you feel tired?");
  if (Answer == YES)
    Solution = "You probably have a virus, rest a few days!";
  else
    Solution = "Knowledge base insufficient. Further diagnosis needed.";
  end if
else
  Answer = AskQuestion ("Do you have problems breathing?");
  if (Answer == YES)
    Solution = "Probably a lung infection, need to do exams."
  else
    Solution = "Knowledge base insufficient. Further diagnosis needed.";
  end if
end if
```

As you can see, the system follows a set of questions, and depending on the answers, either asks more questions or gives a solution.

note

For the rest of this chapter, you can assume that the strings work exactly like other variables, and you can use operators such as = and == to the same effect as in normal types of variables.

Fuzzy Logic

Fuzzy logic expands on the concept of an expert system. While an expert system can give values of either true (1) or false (0) for the solution, a fuzzy logic system can give values in between. For example, to know whether a person is tall, an expert system would do the following (again, this is fictional script):

```
Answer = AskQuestion ("Is the person's height more than 5' 7"?");
if (Answer == YES)
  Solution = "The person is tall.";
else
  Solution = "The person is not tall.";
end if
```

A fuzzy set would appear like so:

```
Answer = AskQuestion ("What is the person's height?");
if (Answer >= 5' 7")
  Solution = "The person is tall.";
end if
if ((Answer < 5' 7") && (Answer < 5' 3"))
  Solution = "The person is almost tall.";
end if
if ((Answer < 5' 3") && (Answer < 4' 11"))
  Solution = "The person isn't very tall.";
else
  Solution = "The person isn't tall.";
end if
```

The result would be fuzzy. Usually a fuzzy set returns values from 0 (false) to 1 (true), representing the membership of the problem. In the last example, a more realistic fuzzy system would use the graph shown in Figure 18.2 to return a result.

As you can see from the graph, for heights greater than 5' 7", the function returns 1; for heights less than 4' 11", the function returns 0; and for values in between, it returns the corresponding value between 5' 7" and 4' 11". You could get this value by subtracting the height from 5' 7" (the true statement) and dividing by 20 (5' 7"–4' 11", which is the variance in the graph). In code, this would be something like the following:

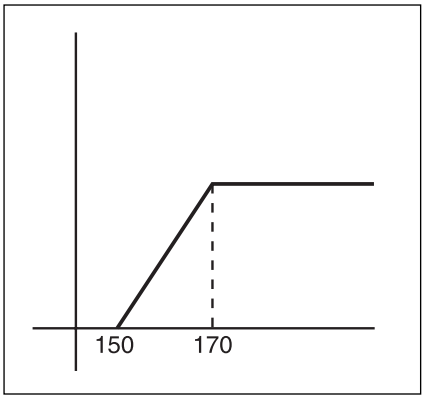


Figure 18.2 Fuzzy membership

```
Answer = AskQuestion ("What is the person's height?");
if (Answer >= 5' 7")
  Solution = 1
end if
if (Answer <= 4' 11")
  Solution = 0
else
  Solution = (Answer - 5' 7") / (5' 7" - 4' 11")
end if
```

You might be wondering why you don't simply use the equation only and discard the `if` clauses. The problem with doing so is that if the answer is more than 5' 7" or less than 4' 11", it will give values outside the 0 to 1 range, thus making the result invalid.

Fuzzy logic is extremely useful when you need reasoning in your game.

Genetic Algorithms

Using genetic algorithms is a method of computing solutions that relies on the concepts of real genetic concepts (such as evolution and hereditary logic). You might have had a biology class in high school that explained heredity, but in case you didn't, the field of biology studies the evolution of subjects when they reproduce. (Okay, maybe there is a little more to it than that, but you are only interested in this much.)

As you know, everyone has a blood type, with the possible types being A, B, AB, and O, and each of these types can be either positive or negative. When two people have a child, their types of blood will influence the type of blood the child has. All that you are is written in your DNA. Although the DNA is nothing more than a collection of bridges between four elements, it holds all the information about you, such as blood type, eye color, skin type, and so on. The little "creatures" that hold this information are called *genes*.

What you might not know is that although you have only one type of blood, you have two genes specifying which blood type you have. How can that be? If you have two genes describing two types of blood, how can you have only one type of blood?

Predominance! Certain genes' information is stronger (or more influential) than that of others, thus dictating the type of blood you have. What if the two genes' information is equally strong? You get a hybrid of the two. For the blood type example, both type A and type B are equally strong, which makes the subject have a blood type AB. Figure 18.3 shows all the possible combinations of the blood types. From this table, you can see that both the A and B types are predominant, and the O type isn't. You can also see that positive is the predominant type.

So, how does this apply to the computer? There are various implementations that range from solving mathematical equations to fully generating artificial creatures for scientific

research. Implementing a simple genetics algorithm in the computer isn't difficult. The necessary steps are described here:

1. Pick up a population and set up initial information values.
2. Order each of the information values to a flat bit vector.
3. Calculate the fitness of each member of the population.
4. Keep only the two with the highest fitness.
5. Mate the two to form a child.

Parent 1	Parent 2	Offspring
A	A	A
A	O	A
A	B	AB
B	B	B
B	O	B
B	A	AB
O	O	O

Figure 18.3 Gene blood type table

And thus you will have a child that is the product of the two best subjects in the population. Of course, to make a nice simulator you wouldn't use only two of the subjects—you would group various subjects in groups of two and mate them to form various children, or *offspring*. Now I'll explain each of the steps.

You first need to use the initial population (all the subjects, including creatures, structures, or mathematical variables) and set them up with their initial values. (These initial values can be universally known information, previous experiences of the subject, or completely random values.) Then you need to order the information to a bit vector, as shown in Figure 18.4.

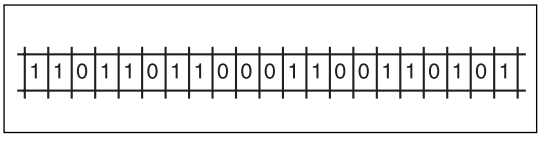


Figure 18.4 Bit vectors (or binary encoding) of information—the virtual DNA

Although some researchers say that an implementation of a genetic algorithm must be done with bit vectors, others say that the bit vectors can be replaced by a function or equation that will analyze each gene of the progenitors and generate the best one out of the two. To be consistent with the DNA discussion earlier, I will use bit vectors (see Figure 18.4).

You now have to calculate the fitness of each subject. The fitness value indicates whether you have a good subject (for a creature, this could be whether the creature was strong, smart, or fast, for example) or a bad subject. Calculating the fitness is completely dependent on the application, so you need to find some equation that will work for what you want to do.

After you calculate the fitness, get the two subjects with the highest fitness and mate them. You can do this by randomly selecting which gene comes from which progenitor or by intelligently selecting the best genes of each to form an even more perfect child. If you

want to bring mutation to the game, you can switch a bit here and there after you get the final offspring. That's it—you have your artificial offspring ready to use. This entire process is shown in Figure 18.5.

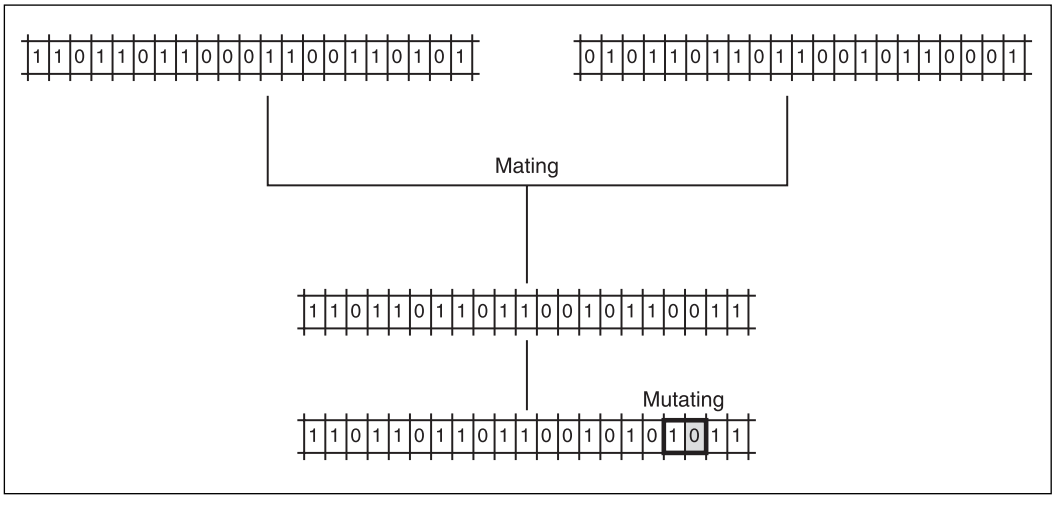


Figure 18.5 Mating and mutation of an offspring

A good use of this technology in games is to simulate artificial environments. Instead of keeping the same elements of the environment over and over, you could make elements (such as small programs) evolve to stronger, smarter, and faster elements (or objects) that can interact with the environment and you.

Neural Networks

Neural networks attempt to solve problems by imitating the workings of a brain. Researchers started trying to mimic animal learning by using a collection of idealized neurons and applying stimuli to them to change their behavior. Neural networks have evolved much in the past few years, mostly due to the discovery of various new learning algorithms, which made it possible to implement the idea of neural networks with success. Unfortunately, there still aren't major discoveries in this field that make it possible to simulate the human brain efficiently.

The human brain is made of around 50 billion neurons (give or take a few billion). Each neuron can compute or process information and send this information to other neurons. Trying to simulate 50 billion neurons in a computer would be disastrous. Each neuron takes various calculations to be simulated, which would lead to around 200 billion calculations. You can forget about modeling the brain fully, but you can use a limited set of neurons (the human brain only uses around 5 to 10 percent of its capacity) to mimic basic actions of humans.

In 1962, Rosenblatt created something called a *perceptron*, one of the earliest neural network models. A perceptron is an attempt to simulate a neuron by using a series of inputs, weighted by some factor, which will output a value of 1 if the sum of all the weighted inputs is greater than a threshold, or 0 if it isn't. Figure 18.6 shows the idea of a perceptron and its resemblance to a neuron.

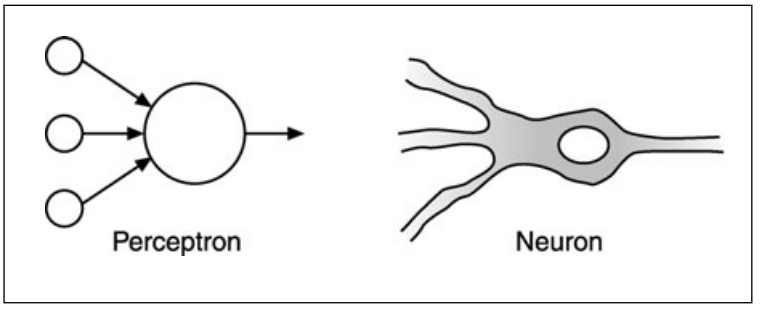


Figure 18.6 A perceptron and a neuron

While a perceptron is just a simple way to model a neuron, many other ideas evolved from this, such as using the same values for various inputs, adding a bias or memory term, and mixing various perceptrons using the output of one as input for others. All of this together formed the current neural networks used in research today.

There are several ways to apply neural networks to games, but probably the most predominant is by using neural networks to simulate memory and learning. This field of artificial intelligence is probably one of its most interesting parts, but unfortunately, the topic is too vast to give a proper explanation of it here. Fortunately, neural networks are becoming more and more popular these days, and numerous publications are available about the subject.

Deterministic Algorithms

Deterministic algorithms are more of a game technique than an artificial intelligence concept. *Deterministic algorithms* are predetermined behaviors of objects in relation to the universe problem. You will consider three deterministic algorithms in this section—random motion, tracking, and patterns. While some say that patterns aren't a deterministic algorithm, I've included them in this section because they are predefined behaviors.

note

The universe (or universe problem) is the current state of the game that influences the subject, and it can range from the subject's health to the terrain slope, number of bullets, number of adversaries, and so on.

Random Motion

The first, and probably simplest, deterministic algorithm is random motion. Although random motion can't really be considered intelligence (because it's random), there are a few things you can make to simulate some simple intelligence.

As an example, suppose you are driving on a road, you reach a fork, and you really don't know your way home. You would usually take a random direction (unless you are superstitious and always take the right road). This isn't very intelligent, but you can simulate it in your games like so:

```
NewDirection = rand() % 2;
```

This will give a random value that is either 0 or 1, which would be exactly the same thing as if you were driving. You can use this kind of algorithm in your games, but it isn't very much fun. However, there are things to improve here. Another example? Okay. Suppose you are watching some guard patrolling an area. Two things might happen: The guard could move in a logical way, perhaps a circle or straight line, but most of the time he will move randomly. He will move from point A to B, then to C, then go to B, then C again, then D, then back to A, and repeat this in a totally different form. Take a look at Figure 18.7 to see this idea in action.

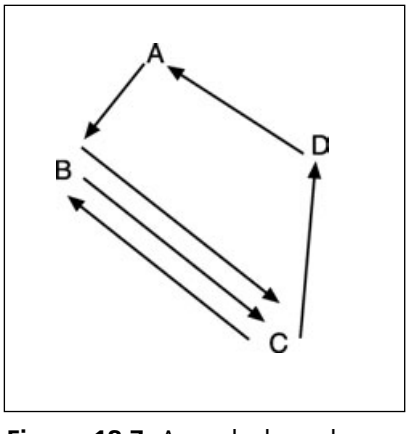


Figure 18.7 A very bad guard

His movement can be described in code something like this:

```
Vector2D kGuardVelocity;
Vector2D kGuardPosition;
int kGuardCycles;
/* Initialize random velocity and cycles */
kGuardVelocity[0] = rand () % 10 - 5;
```

```
kGuardVelocity[1] = rand () % 10 - 5;
kGuardCycles = rand () % 20;
while (GameIsRunning)
{
  // If we still have some cycles with the current movement
  while (kGuardCycles-- > 0)
  {
    A
    D
    C
    B
    kGuardPosition += kGuardVelocity;
  }
  // Change velocity and cycles
  kGuardVelocity [0] = rand () % 10 - 5;
  kGuardVelocity [1] = rand () % 10 - 5;
  kGuardCycles = rand () % 20;
}
```

And you have your guard. You might think this isn't very intelligent, but if you were only playing the game, you would simply see that the guard was patrolling the place, and you would think that he was being intelligent.

note

Some of the psuedo-code in this chapter is based on the code developed to represent vectors in Chapter 19, "The Mathematical Side of Games."

Tracking

When you are trying to catch someone, there are a few things you must do. First, move faster than him, or else you will never catch him, and move in the direction he is from you. There is no logic in running south if he is north of you.

To solve this problem and add a little more intelligence to your games, you can use a tracking algorithm. Suppose the guard spots an intruder. He would probably start running toward him. If you wanted to do this in your game, you would use the following code:

```
Vector2D kGuardVelocity;
Vector2D kGuardPosition;
Vector2D kIntruderPosition;
int iGuardSpeed;
// Intruder was spotted, run to him
Vector2D kDistance;
```

```
kDistance = kIntruderPosition - kGuardPosition;
kGuardVelocity = kDistance.Normalize();
kGuardVelocity *= iGuardSpeed;
kGuardPosition += kGuardVelocity;
```

This code gets the direction from the intruder to the guard (the normalized distance) and moves the guard to that direction by a speed factor. Of course, there are several improvements you could make to this algorithm, such as taking into account the intruder's velocity and maybe doing some reasoning about the best route to take.

The last thing to learn with regard to tracking algorithms is about anti-tracking algorithms. An *anti-tracking algorithm* uses the same concepts as the tracking algorithm, but instead of moving toward the target, it runs away from the target. In the previous guard example, if you wanted the intruder to run away from the guard, you could do something like this:

```
mrVector2D kGuardVelocity;
mrVector2D kGuardPosition;
mrVector2D kIntruderPosition;
mrUInt32 iGuardSpeed;
// Guard has spotted the intruder, intruder run away from him
mrVector2D kDistance;
kDistance = kGuardPosition - kIntruderPosition;
kGuardVelocity = -kDistance.Normalize();
kGuardVelocity *= iGuardSpeed;
kGuardPosition += kGuardVelocity;
```

As you can see, the only thing you need to do is negate the distance to the target (the distance from the guard to the intruder). You could also use the distance from the intruder to the guard and not negate it, because it would produce the same final direction.

Patterns

A *pattern*, as the name indicates, is a collection of actions. When those actions are performed in a determined sequence, a pattern (repetition) can be found. Take a look at my rice-cooking pattern, for example. There are several steps I take when I'm cooking rice:

1. Take the ingredients out of the cabinet.
2. Get the cooking pan from under the counter.
3. Add about two quarts of water to the pan.
4. Boil the water.
5. Add 250 grams of rice, a pinch of salt, and a little lemon juice.
6. Let the rice cook for 15 minutes.

And presto, I have rice ready to be eaten. (You don't mind if I eat while I write, do you?) Whenever I want to cook rice, I follow these steps or this pattern. In games, a pattern can be as simple as making an object move in a circle or as complicated as executing orders, such as attacking, defending, harvesting food, and so on. How is it possible to implement a pattern in a game? First you need to decide how a pattern is defined. For your small implementation, you can use a simple combination of two values—the action description and the action operator. The *action description* defines what the action does, and the *action operator* defines how it does it. The action operator can express the time to execute the action, how to execute it, or the target for the action, depending on what the action is.

Of course, your game might need a few more arguments to an action than only these two; you can simply add the necessary parameters. Take another look at the guard example. Remember that there were two things the guard might be doing if he was patrolling the area—moving randomly (as you saw before) or in a logical way. For this example, assume the guard is moving in a logical way—that he is performing a square-styled movement, as shown in Figure 18.8.

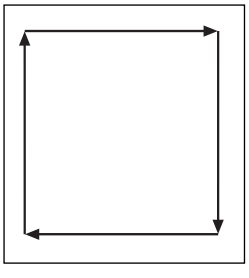


Figure 18.8 A good guard patrolling the area

As you can see, the guard moves around the area in a square-like pattern, which is more realistic than moving randomly. Now, doing this in code isn't difficult, but you first need to define how an action is represented. For simple systems like yours, you can define an action with a description and an operator. The description field describes the action (well, duh!), but the operator can have various meanings. It can be the time the action should be performed, the number of shots that should be fired, or anything else that relates to the action. For the guard example, the operator would be the number of feet to move. Although this system works for many actions, you might want to introduce more data to the pattern. Doing so is easy; you simply need to include more operators in the action definition. A simple example could be:

```
class Action
{
public:
string Description;
string Operator;
};
```

To make your guard pattern, you could do something like this:

```
Action GuardPattern [4];
GuardPattern[0].Description = "MoveUp";
GuardPattern[0].Operator = "10";
GuardPattern[1].Description = "MoveRight";
```

```
GuardPattern[1].Operator = "10";
GuardPattern[2].Description = "MoveDown";
GuardPattern[2].Operator = "10";
GuardPattern[3].Description = "MoveLeft";
GuardPattern[3].Operator = "10";
```

And your guard pattern would be defined. The last thing you need to do is the pattern processor. This isn't hard; you simply need to check the actual pattern description and, depending on the pattern description, perform the action like so:

```
mrUInt32 iNumberOfActions = 4;
mrUInt32 iCurrentAction;
for (iCurrentAction = 0; iCurrentAction < iNumberOfActions;
iCurrentAction++)
{
  if (GuardPattern [iCurrentAction].Description == "MoveUp";
  {
    kGuardPosition [1] += GuardPattern [iCurrentAction].Operator;
  }
  if (GuardPattern [iCurrentAction].Description == "MoveRight';
  {
    kGuardPosition [0] += GuardPattern [iCurrentAction].Operator;
  }
  if (GuardPattern [iCurrentAction].Description == "MoveDown";
  {
    kGuardPosition [1] -= GuardPattern [iCurrentAction].Operator;
  }
  if (GuardPattern [iCurrentAction].Description == "MoveUp";
  {
    kGuardPosition [0] -= GuardPattern [iCurrentAction].Operator;
  }
}
```

This would execute the pattern to make the guard move in a square. Of course, you might want to change this to only execute one action per frame or execute only part of the action per frame, but that's another story.

Finite State Machines

Random logic, tracking, and patterns should be enough to enable you to create some intelligent characters for your game, but they don't depend on the actual state of the problem to decide what to do. If for some reason a pattern tells the subject to fire the weapon, and there isn't any enemy near, then the pattern doesn't seem very intelligent, does it? That's where finite state machines (or software) enter.

A *finite state machine* has a finite number of states that can be as simple as a light switch (either on or off) or as complicated as a VCR (idle, playing, pausing, recording, and more, depending on how much you spend on it).

A *finite state software application* has a finite number of states. These states can be represented as the state of the playing world. Of course, you won't create a state for each difference in an object's health. (If the object had a health ranging from 0 to 1,000, and you had 10 objects, that would mean 100,010 different states, and I don't even want to think about that case!) However, you can use ranges, such as whether an object's health is below a number, and only use the object's health for objects that are near the problem you are considering. This would reduce the states from 100,010 to about four or five.

Let's resume the guard example. If an intruder were approaching the area, until now you would only make your guard run to him. But what if the intruder is too far? Or too near? And what if the guard had no bullets in his gun? You might want to make the guard act differently. For example, consider the following cases:

1. Intruder is in a range of 1000 feet: Just pay attention to the intruder.
2. Intruder is in a range of 500 feet: Run to him.
3. Intruder is in a range of 250 feet: Tell him to stop.
4. Intruder is in a range of 100 feet and has bullets: Shoot first, ask questions later.
5. Intruder is in a range of 100 feet and doesn't have bullets: Sound the alarm.

You have five scenarios, or more accurately, states. You could include more factors in the decision, such as whether there are any other guards in the vicinity, or you could get more complicated and use the guard's personality to decide. If the guard is too much of a coward, you probably never shoot, but just run away. The previous steps can be described in code like this:

```
// State 1
if ((DistanceToIntruder () > 500) && (DistanceToIntruder () < 1000))
{
    Guard.TakeAttention ();
}
// State 2
if ((DistanceToIntruder () > 250) && (DistanceToIntruder () < 500))
{
  Guard.RunToIntruder ();
}
// State 3
if ( (DistanceToIntruder () > 100) && (DistanceToIntruder () < 250))
{
  Guard.WarnIntruder ();
```

```
}
// State 4
if (DistanceToIntruder () < 100)
{
  if (Guard.HasBullets ())
  {
    Guard.ShootIntruder();
  }

  // State 5
  else
  {
    Guard.SoundAlarm();
  }
}
```

Not hard, was it? If you combine this with the deterministic algorithms you saw previously, you can make a very robust artificial intelligence system for your games.

Fuzzy Logic

I have already covered the basics of fuzzy logic, but this time I will go into several of the fuzzy logic techniques more deeply, and explain how to apply them to games.

Fuzzy Logic Basics

Fuzzy logic uses some mathematical sets theory, called *fuzzy set theory*, to work. If you're rusty with sets, check the mathematics chapter (Chapter 19, "The Mathematical Side of Games") before you continue. Fuzzy logic is based on the membership property of things. For example, while all drinks are included in the liquids group, they aren't the only things in the group; some detergents are liquids too, and you don't want to drink them, do you? The same way that drinks are a subgroup—or more accurately, a subset—of the liquids group, some drinks can also be subsets of other groups, such as wine and soft drinks. In the wine group, there are red and white varieties. In the soft drink group, there are carbonated and non-carbonated varieties.

All this talk about alcoholic and non-alcoholic drinks was for demonstration purposes only, so don't go out and drink alcohol just to see whether I'm right. Alcohol damages your brain and your capacity to code, so stay away from it (and drugs, too).

Okay, I'll stop being so paternal and get back to fuzzy logic. Grab a glass and fill it with some water (as much as you want). The glass can have various states—it can be empty, half full, or full (or anywhere in between). How do you know which state the glass is in? Take a look at Figure 18.9.

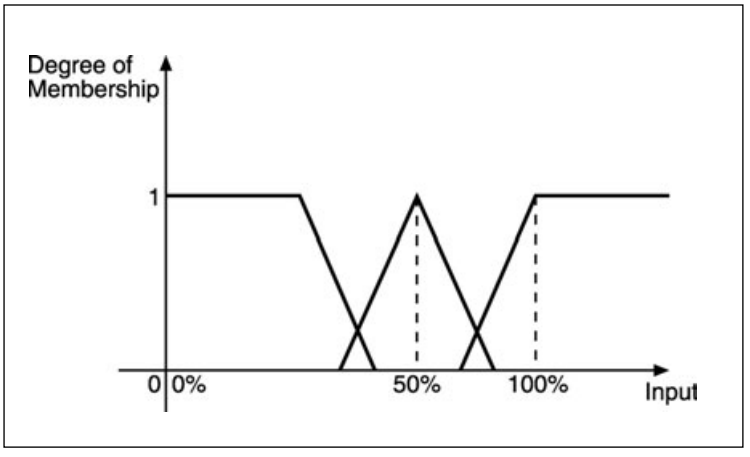


Figure 18.9 Group membership for a glass of water

As you can see, when the glass has 0 percent water, it is totally empty; when it has 50 percent water, it is half full (or half empty, if you prefer). When it has 100 percent of its size in water, then it is full. What if you only poured 30 percent of the water? Or 10 percent? Or 99 percent? As you can see from the graph, the glass will have a membership value for each group. If you want to know the membership values of whatever percentage of water you have, you will have to see where the input (the percentage) meets the membership's graphs to get the degree of membership of each, as shown in Figure 18.10.

Memberships graphs can be as simple as the ones in Figure 18.10, or they can be trapezoids, exponentials, or other equation-derived functions. For the rest of this section, you will only use normal triangle shapes to define memberships. As in Figure 18.10, you can see that the same percentage of water can be part of two or more groups, where the greater membership value will determine the value's final membership.

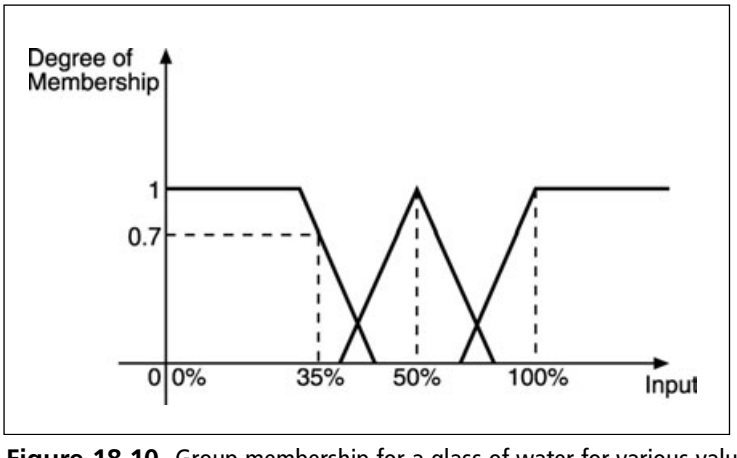


Figure 18.10 Group membership for a glass of water for various values

You can also see that the final group memberships will range from zero to one. This is one of the requirements for a consistent system. To calculate the membership value on a triangle membership function, assuming that the value is inside the membership value (if it isn't, the membership is just zero), you can use the following code:

```
float fCenterOfTriangle = (fMaximumRange - fMinimumRange) / 2;
/* Value is in the center of the range */
if (fValue == fCenterTriangle)
{
  fDegreeOfMembership = 1.0;
}
/* Value is in the first half of the range */
if (fValue < fCenterTriangle)
{
  fDegreeOfMembership = (fValue - fMinimumRange) /
    (fCenterTriangle - fMinimumRange);
}
/* Value is in the second half of the range */
if (fValue > fCenterTriangle)
{
  fDegreeOfMembership = ((fMaximumRange - fCenterTriangle) - (fValue -
    fCenterTriangle)) / (fMaximumRange - fCenterTriangle);
}
```

And you have the degree of membership. If you played close attention, what you did was use the appropriate line slope to check for the vertical intersection of `fValue` with the triangle.

Fuzzy Matrices

The last topic about fuzzy logic I want to cover is fuzzy matrices. This is what really makes you add intelligence to your games. First, I need to pick a game example to demonstrate this concept. Anyone like soccer?

You will be defining three states of the game.

1. The player has the ball.
2. The player's team has the ball.
3. The opposite team has the ball.

Although there are many other states, you will only be focusing on these three. For each of these states, there is a problem state for the player. You will be considering the following:

1. The player is clear.
2. The player is near an adversary.
3. The player is open for a goal.

Using these three states, as well as the previous three, you can define a matrix that will let you know which action the player should take when the two states overlap. Figure 18.11 shows the action matrix.

	Player has ball	Player team has ball	Adversaries have ball
Player is clear	Run for goal	Try to get a good position	Run to nearest adversary
Player is near adversary	Pass the ball	Try to get clear	Try to tackle the adversary
Player is open for goal	Shoot	Get a good position to shoot	Run to nearest adversary

Figure 18.11 The action matrix for a soccer player

Using this matrix would make the player react like a normal player would. If he is clear and doesn't have the ball, he will try to get in a favorable position for a goal. If he has the ball at a shooting position, he will try to score. You get the idea.

But how do you calculate which state is active? It's easy—you use the group membership of each state for both inputs, and multiply the input row by the column row to get the final result for each cell. (It's not matrix multiplication; you simply multiply each row position by the column position to get the row column value.) This will give you the best values from which to choose. For example, if one cell has a value of 0.34 and the other cell has a value of 0.50, then the best choice is probably to do what the cell with 0.50 says. Although this isn't an exact action, it is the best you can take. There are several ways to improve this matrix, such as using randomness, evaluating the matrix with another matrix (such as the personality of the player), and many more.

A Simple Method for Memory

Although programming a realistic model for memory and learning is hard, there is a method that I personally think is pretty simple to implement—you can store game states as memory patterns. This method will save the game state for each decision it makes (or for each few, depending on the complexity of the game) and the outcome of that decision; it will store the decision result in a value from zero to one (with zero being a very bad result and one being a very good result).

For example, consider a fighting game. After every move the subject makes, the game logs the result (for example, whether the subject hit the target, missed the target, caused much damage, or was hurt after the attack). Calculate the result and adjust the memory result for that attack. This will make the computer learn what is good (or not) against a certain player, especially if the player likes to follow the same techniques over and over again.

You can use this method for almost any game, from Tic-Tac-Toe, for which you would store the player's moves and decide which would be the best counter-play using the current state of the game and the memory, to racing games, for which you would store the movement of the cars from point to point and, depending on the result, choose a new way to get to the path. The possibilities are infinite, of course. This only simulates memory, and using only memory isn't the best thing to do—but it is usually best to act based on memory instead of only pure logic.

Artificial Intelligence and Games

There are various fields of artificial intelligence, and some are getting more advanced each day. The use of neural networks and genetic algorithms for learning is pretty normal in today's games. Even if all these techniques are being applied to games nowadays and all the hype is out, it doesn't mean you need to use it in your own games. If you need to model a fly, just make it move randomly. There is no need to apply the latest techniques in genetic algorithms to make the fly sound like a fly; random movement will do just as well (or better) than any other algorithm. There are a few rules I like to follow when I'm developing the artificial intelligence for a game.

1. If it looks intelligent, then your job is done.
2. Put yourself in the subject's place and code what you think you would do.
3. Sometimes the simpler technique is the needed one.
4. Always pre-design the artificial intelligence.
5. When nothing else works, use random logic.

Summary

This chapter has provided a small introduction to artificial intelligence. Such a broad topic could easily take a few sets of books to explain—and even then, many details would have to be left out. The use of artificial intelligence depends much on the type of game you are developing, so it is usually also very application-specific. While 3D engines can be used repeatedly, it is less likely that artificial intelligence code can. Although this chapter covered some of the basics of artificial intelligence, it was just a small subset of what you might use, so don't be afraid to experiment!

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. Which of the following is *not* one of the three deterministic algorithms covered in this chapter?
   A. Random logic
   B. Tracking
   C. Conditions
   D. Patterns

2. Can fuzzy matrices be used without multiplying the input memberships? Why or why not?
   A. No, it is absolutely necessary to multiply the input memberships.
   B. Yes, but only after negating the matrix.
   C. Yes, it is possible using AND and OR operators, and then randomly selecting action for the active cell.
   D. Yes, it is possible using XOR and NOT operators after multiplying the matrix.

3. Which type of system solves problems that are usually solved by specialized humans?
   A. Expert system
   B. Deterministic algorithm
   C. Conditional algorithm
   D. If-then-else

4. Which type of intelligence system is based on an expert system, but is capable of determining fractions of complete answers?
   A. Genetic algorithm
   B. Fuzzy logic
   C. Deterministic algorithm
   D. Expert system

5. Which type of intelligence system uses a method of computing solutions for a hereditary logic problem?
   A. Expert system
   B. Fuzzy logic
   C. Genetic algorithm
   D. Conditional logic

6. Which type of intelligence system solves problems by imitating the workings of a brain?
   A. State machine
   B. Genetic algorithm
   C. Fuzzy logic
   D. Neural network

7. Which of the following uses predetermined behaviors of objects in relation to the universe problem?
   A. Genetic algorithm
   B. Deterministic algorithm
   C. Fuzzy logic
   D. Neural network

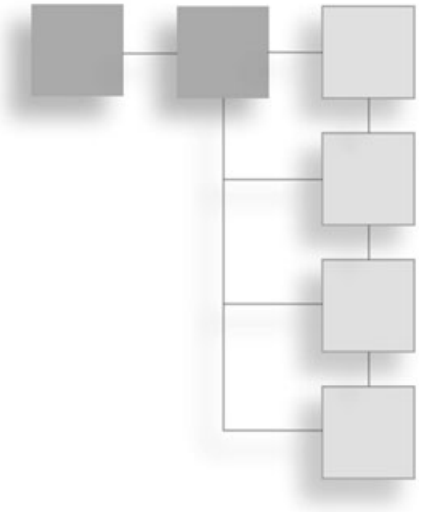
8. Which type of deterministic algorithm "fakes" intelligence?
   A. Patterns
   B. Tracking
   C. Random motion
   D. Logic

9. Which type of deterministic algorithm will cause one object to follow another?
   A. Tracking
   B. Conditional
   C. Patterns
   D. Random motion

10. Which type of deterministic algorithm follows preset templates?
    A. Tracking
    B. Random motion
    C. Genetic
    D. Patterns

This page intentionally left blank

CHAPTER 19



The Mathematical Side of Games

As you might already know, math is an extremely important subject in high-level computer programming, especially in game programming. Behind the scenes, in the graphics pipeline, and in the physics engine, heavy math is being processed by your computer, often with direct implementation in the silicon (as is the case with most graphics chips). While a huge amount of heavy math is needed to get a polygon on the screen with a software renderer, that is all handled by highly optimized (and fantastically complex) mathematics built into the latest graphics processors. Vectors, matrices, functions, and other math-related topics comprise an indispensable section in any game-programming curriculum. In this chapter, I will go over basic linear algebra, such as vector operations, matrices, and probability, with a bit of calculus when I get into the basics of functions. Please note that this is an extremely simple primer on basic algebra and calculus and should accomplish little more than whetting your appetite. For a really solid treatment of game mathematics, please refer to *Mathematics for Game Developers* (Course Technology PTR, 2004) by Christopher Tremblay, who has tackled the subject with a tenacity that is sure to enhance your math skills.

Here is a breakdown of the major topics in this chapter:

- Using trigonometry
- Understanding vectors
- Working with matrices
- Using probability
- Working with functions

Trigonometry

Trigonometry is the study of angles and their relationships to shapes and various other geometries. You will use some of the material covered here as support for some advanced operations you will build later.

Visual Representation and Laws

Before I go into the details of trigonometry, let me introduce a new concept—radians. A *radian* is a measurement of an angle, just like a degree. One radian is the angle formed in any circle where the length of the arc defined by the angle and the circle radius are of same length, as shown in Figure 19.1. You will use radians as your measurement because they are the units C++ math functions use for angles. Because you are probably accustomed to using degrees as your unit of measurement, you need to be able to convert from radians to degrees and vice versa. As you might know, π radians is the angle that contains half a circle, as you can see in Figure 19.2. And you probably know that 180 degrees is also the angle that contains half a circle. Knowing this, you can convert any radian unit to degrees, as shown in Equation 19.1, and vice versa using Equation 19.2.

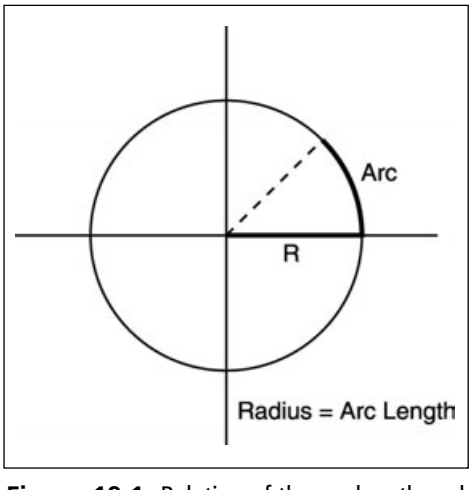


Figure 19.1 Relation of the arc length and radius of the circle

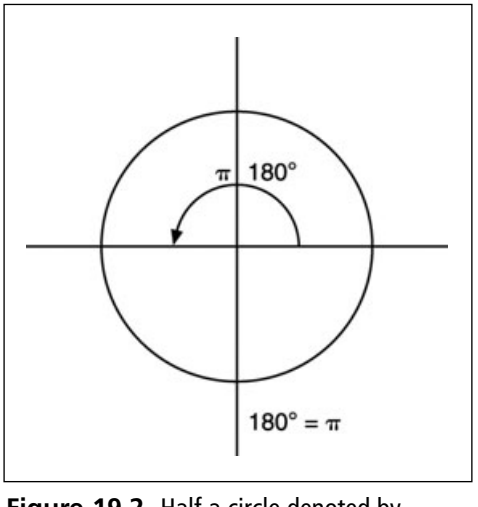


Figure 19.2 Half a circle denoted by radians and degrees

$$\text{Radians} = \frac{\text{Degrees} * \pi}{180}$$

Equation 19.1

$$\text{Degrees} = \frac{\text{Radians} * 180}{\pi}$$

Equation 19.2

```
double DegreeToRadian(double degree)
{
    return (degree * PI / 180);
}
double RadianToDegree(double radian)
{
    return (radian * 180 / PI);
}
```

Now that you know what a radian is, I'll explain how to use them. Take a look at Figure 19.3. From the angle and the circle radius, you can get the triangle's sides and angles. If you examine that circle a little bit closer, you will see that in any triangle that contains the center of the circle and the end of the arc as vertices, the hypotenuse of that triangle is the line formed from the circle's center to the end of the arc. Now you need to find the two other lines' lengths that form the triangle. You will find these using the cosine and sine functions. The three equations that are important in geometry are cosine, sine, and tangent, and they are directly related to the triangle. See the cosine Equation 19.3, the sine Equation 19.4, and the tangent Equation 19.5.

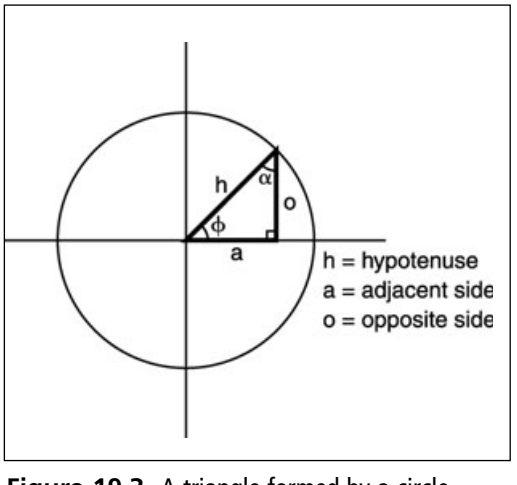


Figure 19.3 A triangle formed by a circle radius and an angle; π radians = 180 degrees

$$\text{cosine}(\phi) = \frac{\text{Adjacent Side}}{\text{Hypotenuse}}$$

Equation 19.3

$$\text{sine}(\phi) = \frac{\text{Opposite Side}}{\text{Hypotenuse}}$$

Equation 19.4

$$\text{tangent}(\phi) = \frac{\sin(\phi)}{\cos(\phi)} = \frac{\text{Opposite Side}}{\text{Adjacent Side}}$$

Equation 19.5

You can calculate these trigonometric operations using the MacLaurin series, but that is beyond the scope of this book. Now you can determine the length of the adjacent side of the triangle on the circle by using the cosine, as shown in Equation 19.6.

$$\text{cosine}(\phi) = \frac{\text{Adjacent Side}}{\text{Hypotenuse}} \Leftrightarrow$$

$$\text{Adjacent Side} = \text{Circle Radius} * \text{cosine}(\phi)$$

Equation 19.6

What if you want to know the angles at each side of the triangle? You use exactly the same equations as you used before to get the sine or the cosine. When you have them, you use the inverse of those operations to get the angles. Taking the triangle in Figure 19.3, you find two of the angles. You don't need to find one of the angles because you already know that the triangle is a right angle triangle, and as such, the angle formed is 90 degrees, or one-half π.

$$\text{cosine}(\phi) = \frac{\text{Adjacent Side}}{\text{Hypotenuse}} \Leftrightarrow$$

$$\phi = \text{cosine}^{-1}\left(\frac{\text{Adjacent Side}}{\text{Circle Radius}}\right)$$

Equation 19.7

$$\text{cosine}(\alpha) = \frac{\text{Opposite Side}}{\text{Hypotenuse}} \Leftrightarrow$$

$$\alpha = \text{cosine}^{-1}\left(\frac{\text{Opposite Side}}{\text{Circle Radius}}\right)$$

Equation 19.8

What is the difference between the two equations? If you look carefully, you are trying to get the angle ⟨ using the cosine and the opposite side. You do this because the opposite side of the angle ⟨ is actually the adjacent side in relation to that angle. So what does this mean? It means that the terms *adjacent* and *opposite* are relative to the angle to which they are referred. In the second calculation, the opposite side should actually be the adjacent side of that angle. Table 19.1 shows you the list of trigonometric functions. This might seem complicated, but it will become clearer when you start using all of this later.

Table 19.1 C Trigonometric Functions

Trigonometric	C Function	C Function Inversed
cosine	cos	acos
sine	sin	asin
tangent	tan	atan/atan2

* These functions are all defined in `math.h`.

Angle Relations

A couple of relations can prove useful when you are dealing with angles and trigonometric functions. One of the most important relations is the trigonometric identity shown in Equation 19.9.

$$\text{sine}^2(\phi) + \text{cosine}^2(\phi) = 1$$

Equation 19.9

This equation is the base of all the other relations. To be honest, these relations are used only for problem solving or optimizations. For that reason, I will not go over them in detail; I will simply show them to you so you can use them at your discretion. The following equations are derived from Equation 19.9 and should be used to optimize your code.

$$\text{sine}(2\phi) = 2\text{sine}(\phi) * \text{cosine}(\phi)$$

Equation 19.10

$$\text{cosine}(2\phi) = \text{cosine}^2(\phi) - \text{sine}^2(\phi)$$

Equation 19.11

$$\text{tangent}(2\phi) = \frac{2\text{tangent}(\phi)}{1 - \text{tangent}^2(\phi)}$$

Equation 19.12

Now you are done with trigonometry. Trigonometry isn't very useful per se, but it will prove an indispensable tool later when you use it with other concepts, such as vectors or matrices.

Vectors

A vector is an n-tuple of ordered real values that can represent anything with more than one dimension—for example, a 2D or 3D Euclidean space. Basically, vectors are nothing more than a set of components.

$$\vec{\text{Vector}} = \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_n \end{pmatrix} \in R^n$$

Equation 19.13

Vectors describe both magnitude and direction. In the two-dimensional case, the X and Y components represent the distance from the relative origin to the end of the vector, as you can see in Figure 19.4. Because you are using a 2D world, you define vectors using two components for convenience, with a commonly known notation (x, y). You can also represent just one component of the vector by using a subscript either with the order of the element or with the component identification, as shown in Equation 19.14.

```
#include <math.h>

typedef struct vector2d
```

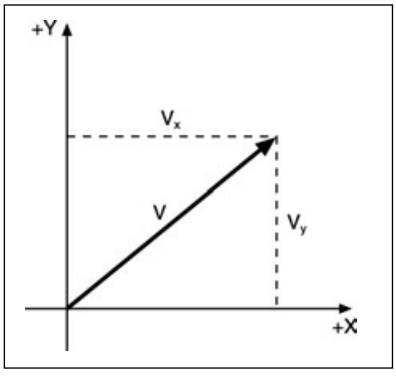


Figure 19.4 A 2D vector composed of two scalars defining the orientation

$$\vec{V}ector = (V_1;V_2) = (V_x;V_y)$$
$$\vec{V}ector = (12;9)$$

Equation 19.14

```
{
    double components[2];
}
vector2d;
```

As you can see, the vector is constituted by an array of two components: X (`components[0]`) and Y (`components[1]`).

Addition and Subtraction

Vectors can be added or subtracted to form new vectors. You can see in Equation 19.15 that the addition of two vectors is completed component by component, which is true for subtraction as well.

$$\vec{A}dded\ Vector = \vec{A} + \vec{B} \Longleftrightarrow$$
$$\vec{A}dded\ Vector = (A_x + B_x;A_y + B_y)$$

Equation 19.15

Equation 19.15 also shows that vector addition can be done in any order, but this isn't true for vector subtraction. If you take a look at Figure 19.5, you can see how the same vectors subtracted in different order produce a vector that is the same in length but different in orientation. Before I move on, I want to create your addition method.

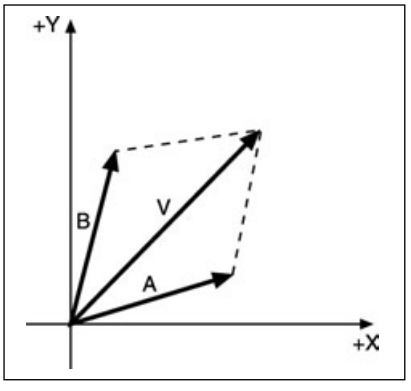


Figure 19.5 Addition of two vectors

```
vector2d vector2d_add(vector2d first, vector2d second)
{
    vector2d newvector;
    newvector.components[0] = first.components[0] + second.components[0];
    newvector.components[1] = first.components[1] + second.components[1];
    return newvector;
}
```

As you can see in Figure 19.6, the subtraction of two vectors gives you the distance between them, but it isn't commutative. If you subtract A – B you get the distance from A to B, whereas in B – A you get the distance from B to A. This is shown in Equation 19.16.

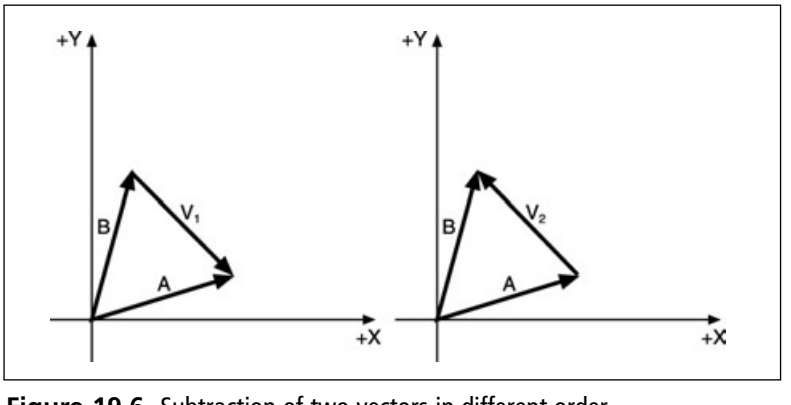


Figure 19.6 Subtraction of two vectors in different order

$$\vec{S}\text{ubtracted Vector} = \vec{A} + \vec{B} \Leftrightarrow$$
$$\vec{S}\text{ubtracted Vector} = (A_x + B_x; A_y + B_y)$$

Equation 19.16

note

In Figure 19.6, you can see that the product of the subtraction has its origin on the end of the first vector. This is incorrect. The vector origin should be the origin of the world.

To finalize this section, let's build the subtraction function.

```
vector2d vector2d_subtract(vector2d first, vector2d second)
{
    vector2d newvector;
    newvector.components[0] = first.components[0] - second.components[0];
    newvector.components[1] = first.components[1] - second.components[1];
    return newvector;
}
```

Scalar Multiplication and Division

You can scale vectors by multiplying or dividing them by scalars, just like normal scalar-to-scalar operations. To do this, you multiply or divide each vector component by the scalar. You can see this in Equation 19.17, which shows multiplication of each of the vector components by a scalar to produce a new vector.

$$\vec{M}\text{ultiplied Vector} = (V_x * \text{Scalar}; V_y * \text{Scalar})$$

Equation 19.17

In code you have:

```
vector2d vector2d_multiply(vector2d vect, double multiplier)
{
    vector2d newvector;
    newvector.components[0] = vect.components[0] * multiplier;
    newvector.components[1] = vect.components[1] * multiplier;
    return newvector;
}
```

You do the same thing for division, as you can see in Equation 19.18.

$$\vec{\text{Divided Vector}} = \left(\frac{V_x;V_y}{\text{Scalar}}\right)$$

Equation 19.18

To end the normal operations, let's build a division function.

```
vector2d vector2d_divide(vector2d vect, double divisor)
{
    vector2d newvector;
    newvector.components[0] = vect.components[0] / divisor;
    newvector.components[1] = vect.components[1] / divisor;
    return newvector;
}
```

Length

The length is the *size* of the vector. The length is used in several other vector operations, so it should be the first one you learn. If you remember the Pythagorean Theorem, you know that the square of the hypotenuse is equal to the sum of the square of each side. You use the same theorem to get the length of the vector, as you can see in Equation 19.19.

$$\| \vec{\text{Vector}} \| = \sqrt{V_x^2 + V_y^2}$$

Equation 19.19

As usual, I'll write a function to calculate the length of a vector.

```
double vector2d_length(vector2d vect)
{
    return (double) sqrt (vect.components[0] * vect.components[0] +
        vect.components[1] * vect.components[1]);
}
```

Normalization

As you saw earlier, vectors have both an orientation and a length, also referred to as the *norm*. Some calculations you use will need a vector of length 1.0. To force a vector to have

a length of 1.0, you must normalize the vector—in other words, divide the components of the vector by its total length, as shown in Equation 19.20.

$$\vec{\text{Normalized Vector}} = \left(\frac{V_x;V_y}{\|\vec{V}\|} \right)$$

Equation 19.20

```
vector2d vector2d_normalize(vector2d vect)
{
    vector2d newvector = vect;
    double length = vector2d_length(vect);
    if (length > 0)
    {
        newvector.components[0] /= length;
        newvector.components[1] /= length;
    }
    return newvector;
}
```

Perpendicular Operation

Finding the perpendicular of a vector is one of those operations you'll use once a year, but let's briefly talk about it anyway. A vector perpendicular to another is a vector that forms a 90-degree angle, or a half-π radians angle with the other. In Figure 19.7, you can see that vector B forms a 90-degree, counterclockwise angle with vector A.

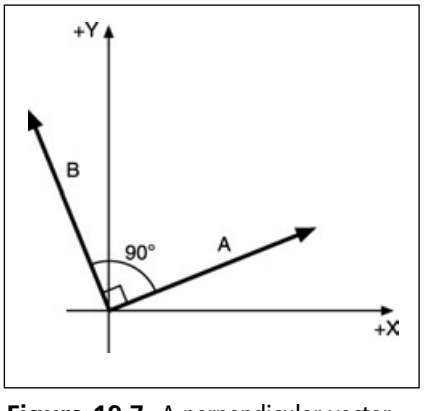


Figure 19.7 A perpendicular vector forming a 90-degree, counterclockwise angle with another vector

Finding the perpendicular vector of a 2D vector is easy; you simply need to negate the Y component and swap it with the X component of the vector, as shown in Equation 19.21.

$$\vec{P}\text{erpendicular Vector}_{\perp} = (-V_y;V_x)$$

Equation 19.21

Just one little thing…. You see that reversed T in Equation 19.21? That is the perpendicular symbol.

```
vector2d vector2d_perpendicular(vector2d vect)
{
    vector2d newvector = vect;
    newvector.components[0] = vect.components[1] * -1;
    newvector.components[1] = vect.components[0];
    return newvector;
}
```

Dot Product

The dot product is probably the most used operation with vectors. You can use it to multiply two vectors, as shown in Equation 19.22.

$$\vec{A} \cdot \vec{B} = A_x * B_x + A_y * B_y$$

Equation 19.22

```
double vector2d_dotproduct(vector2d first, vector2d second)
{
    return (double) first.components[0] * second.components[0] +
        first.components[1] * second.components[1];
}
```

Using the dot product isn't very informative per se, but the dot product can also be defined by Equation 19.23.

This equation gives a little more information, don't you agree? In case you didn't know, ø is the smallest angle formed by the two vectors. With a little thought and by combining Equations 19.22 and 19.23, you can get the equation to find the smallest angle of two vectors (see Equation 19.24).

$$\vec{A} \cdot \vec{B} = ||\vec{A}|| * ||\vec{B}|| * \text{cosine}(\phi)$$

Equation 19.23

$$\text{cosine}(\phi) = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| * ||\vec{B}||} \Leftrightarrow$$

$$\phi = \text{cosine}^{-1}\left(\frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| * ||\vec{B}||}\right)$$

Equation 19.24

You finally have some use for the dot product. If you calculate the arc cosine of the dot product of the two vectors divided by the product of their lengths, you have the smallest angle between them. Now you can build the angle function.

```
double vector2d_angle(vector2d first, vector2d second)
{
    return (double) acos (
        vector2d_dotproduct(first, second) /
        (vector2d_length(first) * vector2d_length(second)));
}
```

Perp-Dot Product

The perp-dot product is nothing new. It is the dot product of a calculated perpendicular vector. This operation is mostly used in physics, as you will see later. How do you find the perp-dot product? Easy—you find the perpendicular of a vector and calculate the dot product of that vector with another, as shown in Equation 19.25.

$$\vec{\text{Perp Dot}} = \vec{A}_{\perp} \cdot \vec{B}$$

Equation 19.25

```
double vector2d_perpdotproduct(vector2d first, vector2d second)
{
    return vector2d_dotproduct(vector2d_perpendicular(first), second);
}
```

Matrices

A simple way of defining a *matrix* is to say that it is a table of values. You can see in Equation 19.26 that a matrix is defined by a set of rows and columns. The number of columns is given by *p* and the number of rows by *q*. You can also access any element of the matrix using the letter *i* for the row and the letter *j* for the column. This is shown in Equation 19.27.

$$\text{Matrix}_{pq} = \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1q} \\ m_{21} & m_{22} & \dots & m_{2q} \\ \dots & \dots & \dots & \dots \\ m_{p1} & m_{p2} & \dots & m_{pq} \end{bmatrix} \in R$$

Equation 19.26

$$m_{ij} = \begin{bmatrix} \dots & \dots & \dots \\ \dots & \dots & m_{23} \\ \dots & \dots & \dots \end{bmatrix}$$

$$i = 2; j = 3$$

Equation 19.27

Addition and Subtraction

Matrix addition and subtraction is done exactly the same way as the vector addition and subtraction. You add (or subtract) each element of one matrix to (or from) the other to produce a third matrix, as shown in Equation 19.28 (for the addition operation).

Matrix addition is commutative (that is, independent of the order), but this isn't the case for subtraction, as you can see in Equation 19.29.

Scalars with Multiplication and Division

Again, to multiply or divide a matrix by a scalar, you multiply or divide each matrix element by the scalar, as shown in Equation 19.30 for multiplication.

$$\text{Matrix Added}_{ij} = A_{ij} + B_{ij}$$

$$\text{Matrix Added} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Leftrightarrow$$

$$\text{Matrix Added} = \begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$$

Equation 19.28

$$\text{Matrix Subtracted}_{ij} = A_{ij} - B_{ij}$$

$$\text{Matrix Subtracted} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Leftrightarrow$$

$$\text{Matrix Subtracted} = \begin{bmatrix} a-1 & b-2 \\ c-3 & d-4 \end{bmatrix}$$

Equation 19.29

$$\text{Matrix Multiplied}_{ij} = A_{ij} * \text{Scalar}$$

$$\text{Matrix Multiplied} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \text{Scalar} \Leftrightarrow$$

$$\text{Matrix Multiplied} = \begin{bmatrix} a * \text{Scalar} & b * \text{Scalar} \\ c * \text{Scalar} & d * \text{Scalar} \end{bmatrix}$$

Equation 19.30

This is exactly the same for the division process, shown in Equation 19.31.

Scalar operations in matrices are pretty easy and usually unnecessary. Next I will go over the most useful matrix operations.

Special Matrices

There are two special matrices I want to go over—the zero matrix and the identity matrix. First, the *zero matrix* is a matrix that, when added to any other matrix, produces the matrix shown in Equation 19.32.

$$\text{Matrix Divided}_{ij} = \frac{A_{ij}}{\text{Scalar}}$$

$$\text{Matrix Divided} = \frac{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}{\text{Scalar}} \Leftrightarrow$$

$$\text{Matrix Divided} = \begin{bmatrix} \frac{a}{\text{Scalar}} & \frac{b}{\text{Scalar}} \\ \frac{c}{\text{Scalar}} & \frac{d}{\text{Scalar}} \end{bmatrix}$$

Equation 19.31

$$\text{Matrix} = \text{Zero Matrix} + \text{Matrix}$$

$$\text{Zero Matrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Equation 19.32

As long as it is a 2×2 matrix, the result of the operation is M—no matter what M is. The *identity matrix* is a matrix that, when multiplied by any other matrix, produces the same matrix as shown in Equation 19.33.

$$\text{Matrix} = \text{Identity Matrix} * \text{Matrix}$$

$$\text{Identity Matrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Equation 19.33

Again, as long as it is a 2×2 matrix, the result of this operation is M—no matter what M is.

Transposed Matrices

A *transposed matrix* is a matrix in which the matrix values are swapped with the other diagonal element, proving Equation 19.34 true. This operation is usually used to change coordinate systems in 3D.

$$\text{Matrix Transposed}_{ij} = M_{ji}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Equation 19.34

Matrix Concatenation

You have reached one of the most needed (and one of the most complicated) matrix operations—matrix multiplication, or more correctly, concatenation. *Concatenation* is the real name for matrix multiplication. This operation enables you to concatenate matrices to produce various effects, such as rotating or shearing. Equation 19.35 presents an example of matrix multiplication.

$$\text{Matrix Concatenated}_{ij} = \sum_{u-1}^{3} A_{iu} * B_{uj}$$

Equation 19.35

Well, you have a new symbol in your game. The Σ symbol, in English, represents the sum. Look at the math in Equation 19.36.

$$\sum_{i=0}^{n} \text{mass}_i$$

Equation 19.36

There are three things to explain—the symbol, the number above it, and the number below it. What you do with this bit of math is sum all the masses you have in the equation above `n`. Suppose that mass is an array, such as `int mass [n]`, and you want to add every element of mass from `i = 0` to `n`.

It's easy if you think of it like a programmer would, isn't it? So basically, the sum symbol means that you will add each element of an array from `i` to `n`. In Equation 19.37, what you actually do is add all the products of the row of matrix A with the column of matrix B to get each element of the result matrix. It's easier to check this with the example in Equation 19.37.

$$\text{Matrix Concatenated} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Leftrightarrow$$

$$\text{Matrix Concatenated} = \begin{bmatrix} a*1+b*3 & a*2+b*4 \\ c*1+d*3 & c*2+d*4 \end{bmatrix}$$

Equation 19.37

I want to go over how you actually come to these results. First, you will find Matrix Concatenated$_{ij}$. If you look at Equation 19.38, you can see that:

$\text{MatrixConcatenated}_{ij} = A_{iu} * B_{uj} + A_{i(u+i)} * B_{(u+i)j}$

$$\text{Matrix Concatenated}_{ij} = \sum_{u=1}^{2} A_{iu} * B_{uj}$$

Equation 19.38

Since u starts at 1 and ends at 2, you can say that:

$\text{MatrixConcatenated}_{11} = A_{11} * B_{11} + A_{12} * B_{21}$, or $\text{MatrixConcatenated}_{11} = a*1+b*3$

You do the same for each element, as follows:

$\text{MatrixConcatenated}_{12} = A_{11} * B_{12} + A_{12} * B_{22} = a*2+b*4$

$\text{MatrixConcatenated}_{21} = A_{21} * B_{11} + A_{22} * B_{21} = c*1+d*3$

$\text{MatrixConcatenated}_{22} = A_{21} * B_{12} + A_{22} * B_{22} = c*2+d*4$

Vector Transformation

Being able to transform vectors by matrices is one of the fundamental tasks for 2D manipulation, but the concept behind it is very simple. If you treat a 2D vector as a matrix of size 1×2, you can multiply the *matrix vector* by another matrix the same way you would with two matrices, as shown in Equation 19.39.

You just treat the vector as a matrix, and there you have it.

$$\vec{\text{Vector Transformed}} = A * \vec{V}$$

$$\vec{\text{Vector Transformed}} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} \Leftrightarrow$$

$$\vec{\text{Vector Transformed}} = \begin{bmatrix} a*1+b*2 \\ c*1+d*2 \end{bmatrix}$$

Equation 19.39

Probability

Probability is a study of math that analyzes events and then tries to evaluate the odds of those events happening. I want to go over a simple example.

From yesterday's weather forecast, there is a good probability of heavy wind and a 50-percent chance of rain.

This forecast actually tells you the probability of heavy wind or rain happening. The text says there is a good probability of heavy wind, so you can say heavy wind has about a 75–90-percent chance of happening—and as for rain, only a 50-percent chance. What does this tell you? Well, if you had 100 days with the exact same forecast, you would probably end up with about 75–90 days with heavy wind, and 50 days with rain. In case you didn't know, 50 percent is actually 0.5.

Sets

A *set* is an unordered collection of objects. You evaluate the objects when you are dealing with probability. They can be numbers, letters, real objects, or just about anything. A set is denoted by a capital letter, and the objects in it are listed between curly braces, such as `SetA = {2, 5, 12, 22}`. Sets are usually defined as a circle with the letter caption and the objects contained, as shown in Figure 19.8.

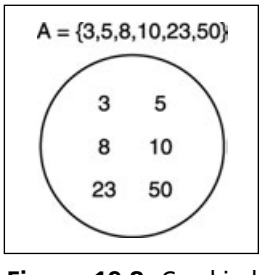


Figure 19.8 Graphical representation of sets

Union

The union operation creates a new set that combines both of the existing sets. You can see this in Equation 19.40.

Figure 19.9 shows a visual representation of the union of two sets.

$$
\begin{aligned}
&A = \{1,3,6,9\}\\
&B = \{2,7,10\}\\
&A \cup B = \{1,3,6,9\} + \{2,7,10\} \Leftrightarrow\\
&A \cup B = \{1,2,3,6,7,9,10\}
\end{aligned}
$$

Equation 19.40

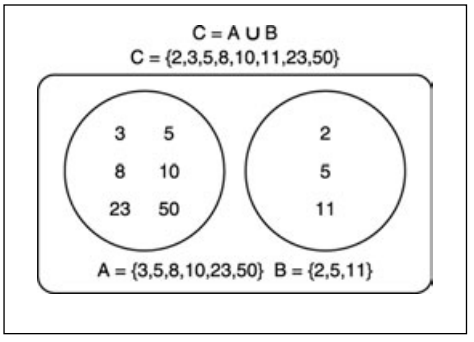


Figure 19.9 Union of two sets

Here is an example in pseudo-code:

```
List unionset;
List setA;
List setB;
unionset = setA;
For each element of setB
Begin
  If element exists in setA, do nothing
  Otherwise, add it to unionset
End
```

Intersection

The intersection operation is straightforward. You compare each element of a set to another set. The elements that are contained in both sets are elements that appear in the *intersected* set, as shown in Equation 19.41 and Figure 19.10.

$$A = \{1,2,5,9\}$$
$$B = \{2,5,7,10\}$$
$$A \cap B = \{1,2,5,9\} - \{2,5,7,10\} \Leftrightarrow$$
$$A \cap B = \{2,5\}$$

Equation 19.41

Here is some pseudo-code that describes the process:

```
List IntersectionSet;
List ListA;
```

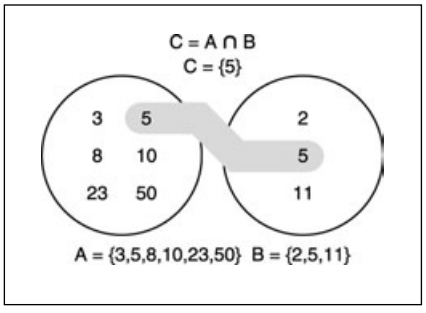


Figure 19.10 Intersection of two sets

```
List ListB;
For each element of SetB
Begin
  If element exists in SetA, add it to IntersectionSet
  Otherwise, do nothing
End
```

As you can see from the code, you go over each element of the set and see whether it exists in the other set. If it does, it is added to the final set; if it doesn't exist, it is ignored.

Functions

A function is really an equation, but because you used equation names for all the formulas before, you need to distinguish these functions from equations. I think an example will help. If I gain 0.22 pounds ever day, how much weight will I have gained after 15 days? You can multiply 0.22 pounds by 15 to get 3.31 pounds. This is correct, but what if you want to know how much I will weigh after 23 days? And what about after 93 days? You can mathematically represent this as a function, as shown in Equation 19.42.

$$\text{Weight Gained (Days)} = 100 * \text{Days}$$

Equation 19.42

You can see this graphically in Figure 19.11. Functions can be used to express various series, ideas, and so on. They are very helpful as a programming tool.

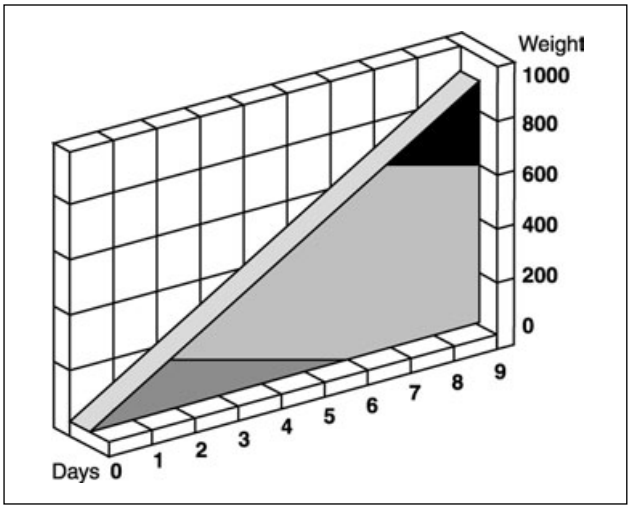


Figure 19.11 Graphical representation of a function

Integration

Differentiation and integration are advanced calculus math topics. I will go over some basic theories related to physics, since you will need it later. If you are driving a car and you press the gas pedal, producing an acceleration of 11.16 miles per hour, how do you get to the velocity and position functions? First you need to define your acceleration function, as shown in Equation 19.43.

$$\text{Acceleration (time)} = 5\text{m/s}^2$$

Equation 19.43

Looking at Equation 19.43, how do you get the velocity function? You need to integrate this function. How? This is a rather simple function, so you can easily do it, as shown in Equation 19.44.

$$\text{Velocity (time)} = \int \text{Acceleration (time)} * \Delta\text{time} \Leftrightarrow$$
$$\text{Velocity (time)} = \text{Initial Velocity} + \text{Acceleration} * \text{time} \Leftrightarrow$$
$$\text{Velocity (time)} = 5 * \text{time}$$

Equation 19.44

How do you know the integration is like this? You cheat. In Appendix B, "Useful Tables," you will find a table of useful integration constants. Now that you have the velocity function, how about getting the position function? Take a look at Equation 19.45.

$$\text{Position (time)} = \int \text{Velocity (time)} * \Delta\text{time} \Leftrightarrow$$
$$\text{Position (time)} = \text{Initial Position} + \text{Initial Velocity} * \text{time} + {}^{1}/_{2}\ \text{Acceleration} * \text{time}^2 \Leftrightarrow$$
$$\text{Position (time)} = 0 + 0 * \text{time} + {}^{1}/_{2}\ 5 * \text{time}^2 \Leftrightarrow$$
$$\text{Position (time)} = {}^{1}/_{2}\ 5 * \text{time}^2$$

Equation 19.45

You also can cheat and use Appendix B to get to the final equation.

Differentiation

A function differentiation gives you the slope of the function at any given position. Differentiating a function is the exact opposite of integrating. Using the example given in the integration section, you can get acceleration from velocity, and velocity from position, as shown in Equations 19.46 and 19.47.

$$\text{Velocity (time)} = \text{Position (time)}' \Leftrightarrow$$
$$\text{Velocity (time)} = ({}^{1}/_{2}\ 5 * \text{time}^2)' \Leftrightarrow$$
$$\text{Velocity (time)} = (5 * \text{time})$$

Equation 19.46

$$\text{Acceleration (time)} = \text{Velocity (time)}' \Leftrightarrow$$
$$\text{Acceleration (time)} = (5 * \text{time})' \Leftrightarrow$$
$$\text{Acceleration (time)} = 5$$

Equation 19.47

As in the integration process, you also can cheat and use the Appendix B tables to get the derivatives. Why am I not going through all of the integration and derivation processes? Honestly, because they would require an entire chapter by themselves.

Summary

I have covered a lot of ground here. Math is one of the fundamental aspects of game programming, but it has been mostly tucked away by game libraries such as DirectX and Allegro. This chapter introduced you to the basics and provided you with enough theory to get you through the basics so you will be prepared (at least marginally) for the mathematical calculations you are likely to find in many game engines today. There are many other mathematical concepts you will need to know during your game programming career, so don't hesitate to check the references in Appendix D for further reading.

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the study of angles and their relationships to shapes and various other geometries?
   A. Calculus
   B. Algebra
   C. Arithmetic
   D. Trigonometry

2. What is the name of the C function that calculates cosine?
   A. `cosine`
   B. `cos`
   C. `sine`
   D. `cosineof`

3. What is the name of the C function that calculates sine?
   A. `sin`
   B. `calc_sine`
   C. `sine`
   D. `sineof`

4. What is the name of the C function that calculates tangent?
   A. `tan`
   B. `tangent`
   C. `calc_tan`
   D. `tangentof`

5. Which C function calculates the inverse sine?
   A. `asine`
   B. `acos`
   C. `atan`
   D. `asin`

6. Which C function calculates the inverse tangent?
   A. `arctangent`
   B. `arctan`
   C. `atan`
   D. `calc_arctan`

7. What does a set intersection contain?
   A. The elements not contained in either set
   B. The elements inversely shared by both sets
   C. The elements that are contained in both sets
   D. The union of elements not shared by either set

8. What does a function differentiation return?
   A. The slope of the function at any given position
   B. The multiplication matrix for two parallel lines
   C. The vector points at both ends of a line
   D. The difference between two matrices

9. What is the opposite of function differentiation?
   A. Interpolation
   B. Conflagration
   C. Integration
   D. Congestion

10. What Greek letter is most often used in calculations of degrees or radians of a circle?
    A. Alpha
    B. Omega
    C. Pi
    D. Theta

This page intentionally left blank

CHAPTER 20

Publishing Your Game

You have finally made it. You have finished your game and you want to publish it. Now you can read the following pages for some advice on how you can do it.

Here is a breakdown of the major topics in this chapter:

- Is your game worth publishing?
- Whose door should you knock on?
- Understanding contracts
- Meeting milestones
- Interviews

Is Your Game Worth Publishing?

Before you seek a publisher, you must evaluate your game. Be truthful to yourself, and also ask friends, family, and even strangers to play your games and give you some feedback. Put yourself in the position of the buyer—would you buy your own game if you saw it in the stores? And if so, how much would you pay for it? These are very important questions to ask yourself when you are thinking about approaching a publisher. In this section, I'll go over a few steps you can follow to see whether your game is worth publishing. Please note that these aren't strict rules.

Probably the most important thing to evaluate in your game is whether it is graphically attractive. Don't get me wrong; I play my old Spectrum games (the good old days) more often than the new 3D perspective mumbo jumbo out there. But unfortunately, only a small group of people do so. Users want their $250 video cards to be stretched to the last polygon. They want to see an infinite number of lights, models, and huge maps, and unfortunately, games of that size require much time from many people.

Don't despair! There is still room for 2D games out there, but they must be very good to beat the new 3D ones. A nice user interface, friendly graphics, and some tricks can do the job, but understand that this is difficult to do. So, your game is fascinating? It has nice graphics and animations and even plays smoothly? Great, move on to the next topic.

The sound is not as important, but it's still a consideration. Does the sound match the actions? Is it immersive? One good way to test this would be to play the game and have a friend sit with his back to the computer and try to describe what the sounds depict to him. If he says that it sounds like a machine gun when you have exploded a mine, it isn't a good sign. You should also pay extra attention to the music. Music should immerse the player in the game, not make him deaf. Make sure the music is pleasing to the ears but still contains the mood of the game. An example of a bad soundtrack would be if you were doing a horror game and your soundtrack consisted of the Bee Gees and the Spice Girls. The music shouldn't force the user to turn it off; rather, it should make him feel he is in the game itself.

One thing to be critical about when evaluating your game is, does it have a beginning, a middle, and an end? Does the player progress through various parts of the game feeling as if he has achieved something? Nowadays, you can't just throw a game to the player and expect him to play if you don't reward him for accomplishing something or you don't explain why he should do things. Don't overlook this part of the game, because it's ten times more important than having cool alpha blend effects. The era of games that consisted of putting a player in a dungeon with a pistol and just letting him play are long gone, my friend.

You should also be concerned with whether the game pulls the player back to play. Is it attractive? Will it make the player be late to his job because he had the desire to kill the boss in level seven? If he does, then you have probably done your job well.

To see whether your game is worth publishing, you can finally determine whether it fits into any hardcore genre. For example, if your game isn't very pretty or doesn't have nice sound but it has a million and one options to run an army, it will probably be interesting to a small hardcore group. The people in these groups tend to buy the game that fits their genre (even if it isn't very impressive graphically) if it excels at simulating a subject in that genre or hobby. There are many types of games that fall into the sub-genres and niche product categories, such as war games, strategy games, and puzzle games.

Whose Door to Knock On

Whose door you should knock on depends much on the type of game and its quality. You can't expect Codemasters to pick your *Pac-Man* clone. Nor should you expect a company that is strictly into the strategy genres to pick your shooter. Knowing what type of game genres publishers are more interested in could help you immensely.

If you have no previous game published, it might be hard to find a publisher even if you have a very good game. You should start at the bottom and build up. Do some small games and sell them online or through budget publishers. Then, start to do more complex games and try to get some small publisher to take them. As you build a name for yourself or for your company, make a lot of contacts along the way, and it will be easier to get to publishers and work out some deals.

Another suggestion is to attend conferences, such as E3 (*Electronic Entertainment Expo*) and GDC (*Game Developers Conference*), and try to get the latest scoop about what publishers are looking for. You can even make some contacts and exchange business cards with some of them.

Learn to Knock Correctly

One of the worst errors new developers make is to get too excited about their games and bombard almost every publisher 20 times about their game. Learning to go through the correct channels to submit a game can help you greatly.

First, check the publisher's Web site and try to find information on how to submit games to them. If you can't find any information, such as a phone number or e-mail address, then e-mail the Webmaster and politely ask whom you should contact to talk about publishing opportunities. This usually works. If you know a publisher's phone number, you can call to get this information and take a chance to do some scouting.

When you have your contact, it's time to let him know you have a game. Send an e-mail to the person and say you have a game of a certain genre, give a two- to three-line description of the game, and explain that you would be interested in working some deal with them. If you have a Web site for the game, send the person a URL for the game's demo and/or screenshots. If the publisher is interested in your game, he will probably send an NDA (*non-disclosure agreement*) and give you the guidelines to submit the game.

Now it's up to you to convince the publisher that your game is worth publishing and that they should be the ones publishing it. Don't ever disrespect or attack the publisher even if they refuse your game. They might not want this game, but they might be interested in another one, and if you do anything to make them angry, you can forget about trying to go to that publisher again.

No Publisher, So Now What?

You couldn't get any publisher to take your game? Don't despair, because it isn't over yet. You can still sell the game yourself. Start a Web site, find a host that can handle credit card purchases (or pay for a payment service), and do a lot of advertising. You might still have a chance to profit from your game.

Contracts

The most important advice I can give you when you start dealing with a contract is, get a lawyer. Get a good lawyer. If possible, try to find a lawyer who has experience negotiating publishing contracts. The ideal one, of course, has experience in the game industry.

Getting a lawyer to analyze the contract for you, check for any loopholes, and see whether it is profitable for you is a must if you plan to publish your games. Don't count on only common sense when you are reading a contract. There are many paragraphs we law-impaired people might think we understand, but we don't. Again, get a good lawyer.

Also, make sure you put everything in writing. Don't count on oral agreements. If they promise you something, make sure it is documented in writing. Now that I gave you my advice, here's an overview of the types of papers you will need to sign.

Non-Disclosure Agreement

The NDA is probably the first thing the publisher will ask you to sign, even before any negotiation is made. This legally-bound paper works as a protection for both you and the publisher. Some people think the NDA is sort of a joke; beware, it isn't. A breach of any paragraph in the NDA can, and probably will, get you into trouble. NDAs are usually safe to sign without much hassle, but you should still check with a lawyer or someone with expertise in the field just to be safe.

The main objective of the NDA is to protect the confidentiality of all talks, papers, files, or other information shared between the publisher and the developer. Some NDAs also include some legal protection (mostly for the publisher) about future disputes that might arise from working together. Some topics the typical NDA covers are

- Confidentiality
- Protection of material submitted by either party
- The fact that all materials submitted by either party will not breach any existing law
- Damage liability
- Time of execution

The Actual Publishing Contract

The actual publishing contract is what you are looking for. The NDA doesn't give you any assurance on the part of the publisher that they will even take your game for review, but the actual contract ensures that you and the publisher have to execute all the paragraphs implied. There isn't much general information I can give you on this one because these contracts change depending on publisher, game type, and game budget. My main advice

is to run the contract by a lawyer because he will be able to help you more than I will. Just be sure to analyze dates and numbers yourself because your lawyer doesn't know how much time you need and how much money you want. Some of the typical topics a normal agreement covers are

- Distribution rights
- Modifications to the original game
- Schedule for milestones
- Royalties
- Confidentiality
- Dates for publishing

Milestones

So, you finally got the contract signed; it's time to lay back and expect the money to pour into your pocket, right? Wrong! You are now at the publisher's mercy. You have to make all the changes in your game that you agreed to in the contract, fix bugs that for some reason don't occur on your computer but happen on others, include the publisher's messages and splash images (including their logos), build demos, and do just about everything stated in the contract. It's a time-consuming task for sure. There are generally three main milestones in the development of a game—the alpha prototype (in which most artwork and programming is complete), the beta version (in which all artwork is final, but programming bugs are still being worked out), and finally the gold release (in which all artwork and programming is finished and a master CD-ROM is sent to the publisher).

Bug Report

You thought you were finished with debugging and bug fixing until the publisher sent you a list with 50-plus bugs? Don't worry; it's natural! When you get a bug report from the publisher, there are usually three types of bugs—critical, normal, and minimal (by order of importance). Some publishers require that you fix all the bugs; others only force you to fix the first two types. My advice is to fix them all! If it becomes public that your game has bugs, it will be a disaster!

Release Day

You made it to release day! Congratulations—not many do. It's time to start thinking of your next game. Start designing, program, and create art so you can have your second game on the shelves as soon as possible!

Interviews

Nothing better than a little insider input from the ones in the business, is there ? Paul Urbanis of Urbonix, Inc; Niels Bauer from Niels Bauer Software Design; and André LaMothe of Xtreme Games LLC were kind enough to answer the following questions.

Paul Urbanus: Urbonix, Inc.

Paul Urbanis is a longtime video game programmer whose experience goes back to the golden age of the video game industry (the 1970s and 1980s), when he was involved in designing both the hardware and software of early game machines.

Q: Thank you for agreeing to be interviewed for this book. Care to give our readers a little background about yourself and your experience?

A: I'm pretty blown away by the tools that are available today. I'm a former video game programmer myself, but I certainly didn't plan it. When I was in school for my electrical engineering degree, I took a cooperative education [co-op] job in the Home Computer Division of Texas Instruments in Lubbock, Texas. At that time, Texas Instruments was manufacturing and selling the TI 99/4. The 99/4 was enhanced in 1981 by adding another graphics mode and a more typewriter-like keyboard, and was called the TI 99/4A, which replaced the 99/4.

I had two co-op phases with TI, and when I returned to school after my first co-op phase, I had a single board TMS9900 [the TI 16-bit micro used in the 99/4A] computer with an instant assembler, a dumb terminal, a 99/4A system, and a complete listing of the monitor ROM. I spent way too much time understanding that machine and too little time on school. I didn't flunk out, but that system for me was like a light bulb for a moth—I was mesmerized and on a quest for knowledge. My ultimate use for this knowledge was to write a video game, since I also spent time playing video games that would have been better spent in homework.

When I returned to TI for my second co-op stint after a year in school, I was much more knowledgeable about the TI-99 architecture. TI was about to introduce their improved machine, the 99/4A. My first assignment was to generate a pass/fail matrix of video chip supply voltage versus temperature. So, I would put the 99/4A into a temperature chamber, set the voltage, wait for the temperature to stabilize, and then log the pass/fail result for all of the voltages. As you can imagine, this was B-O-R-I-N-G, but exactly the kind of work that was pushed off to a co-op student. And I couldn't complain, because I was making good money [$7 per hour in 1982].

Q: What was it like having an electronics degree, working for a computer hardware company, and then finding yourself working on games?

A: Well, when I returned to TI, I discovered that they were working on an editor/assembler package. I was very excited because up to that point, all of my 9900 assembly language programming had been on the single-board computer, and there was no source code storage except for the thermal printer on my dumb terminal. The editor/ assembler was in the internal testing phase. This is where the video chip testing re-enters the picture.

While waiting for the temperature chamber to stabilize, I would be playing around/testing the new editor/assembler package. How cool was this—you could type in code for an hour, and it *wasn't lost* when the computer was turned off? Soon, I was reading about the new graphics mode and had some assembly-language eye candy (screensaver-like stuff) on the screen. This bit of eye candy dramatically changed my co-op job in the Home Computer Division. Because a few days later the head of the HC division, Don Bynum, was walking around, just visiting with everyone—as was his practice—and he saw my graphics experimentation and asked questions such as, "Who are you and what do you have running here?" I explained that I was testing the new graphics chip and the editor/assembler package, and wanted to play with the new graphics mode because no one in the software development group was doing anything with it. He nodded in acknowledgement and continued his visit with the troops.

Later that day, I was called into Don's office, and he told me that I was being reassigned from the Hardware Development group to the Advanced Development group. Now, the real fun started. The first thing I did was get the source code to the assembler on the TI single-board computer I had used to learn TMS9900 assembly language, and port this code to a new cartridge we were working on. I also included my "Lines" eye-candy demo in source and object format so buyers of this cartridge would have an example of using the new graphics mode. After I finished this project, Don Bynum called me and Jim Dramis into his office and told us he wanted us to work on a game together. He suggested a space game, but told us that wasn't written in stone. And we had carte blanche to do whatever we wanted. There was no storyboard, script, or anything else. Just collaborate and write a game. Wow! Life couldn't get any better unless I could get the royal treatment at the Playboy Mansion. By the way, Jim Dramis was responsible for developing TI's best-selling games, *Car Wars* and *Munchman* (a *Pac-Man* clone). Eventually, we wrote a space game and it was named (not by us) *Parsec*.

Q: Believe it or not, I actually owned a TI-44/Plus computer and jury-rigged my dad's tape recorder to save/load programs! What else can you tell me about this space game?

A: *Parsec* was a horizontal scroller, somewhat similar to *Defender*. Unfortunately, due to the architecture of the 99/4A, the graphics memory was not directly in the memory map of the CPU, but instead was accessible only through some video chip control registers. Mainly, there was a 14-bit address register (two consecutive writes to the same 8-bit address) and an 8-bit data register. So the sequence to read/write one or more bytes of graphics memory was

1. Write first byte of address N.
2. Write second byte of address N.
3. Read/write byte of data at N.
4. Read/write byte of data at N+1.
5. Continue until non-contiguous address is needed, then go back to Step 1.

As you can see, random access of graphics memory to do bit-blitting was painfully slow and a real competitive disadvantage when using the 99/4A for gaming. And, in the early 1980s, video games were hot! Of course, that whole market crashed big in 1983/1984 and Nintendo stepped in to fill the void, but that's another story entirely.

Back to *Parsec*. In writing *Parsec*, Jim did most of the game flow and incorporated my suggestions. I contributed two technical breakthroughs to allow *Parsec* to do things that hadn't been possible before—and both of these contributions were directly attributable to my background as a hardware guy and reading the chip specs in detail. I was able to use a small amount of SRAM that only required two clock cycles to cache both the code and scroll buffer for the horizontal scrolling routine. This increased the speed about two times (my best recollection) and made scrolling feasible. The other thing I did was figure out how to use a new user hook [added in the /4A version] into the 60-Hz vertical interrupt to allow speech synthesis [when the speech module was connected] data to be transferred during the vertical interrupt. Prior to this, when any speech was needed the application stopped completely while the speech synthesizer was spoon-fed in a polled loop. I also did the graphics for the asteroid belt. These were actually done using TI LOGO, a LISP-like language enhanced with direct support of the 99/4A graphics. The LOGO files were converted to assembler DATA statements using a utility I wrote.

Q: LOGO! Now that brings back some fond memories, doesn't it? I actually did a lot of "Turtle Graphics" programming when I was just starting to learn how to program.

A: I've been told that TI actually produced around one million *Parsec* games. Of course, after they exited the home computer business at the end of 1983, many of these may have been buried in a landfill somewhere. Also, *Parsec* was the first TI game where the programmers' names were allowed to be included in the manual—at the beginning, no less.

Q: What did you do after that game was completed?

A: After *Parsec*, it was time for me to go back to school, which was New Mexico State University in Las Cruces. I was in school for the fall semester of 1982, and at that time I began to have conversations with two ex-TIers who had opened a computer store in Lubbock, where the Home Computer Division was located. We talked about forming a video gaming company patterned after Activision. The company would initially consist of the two business/marketing guys and the three top TI game programmers—myself, Jim Dramis, and Garth Dollahite. Garth had written *TI-Invaders*, an improved *Space Invaders* knockoff, while he was a co-op student and was hired by TI after he completed his degree. At that time, the home computer video game market was extremely hot. So, in January of 1983, I moved to Lubbock. But Jim and Garth hadn't quit TI yet, even though they had agreed they would do so. And I was sharing an apartment with Garth, as we were both single. In February, they both resigned and Sofmachine was born.

Q: How was the company organized?

A: The stock in Sofmachine was evenly divided between the five principals. The plan was for the business types to raise money by selling shares in a limited partnership, and we programmers would each write a game. As it turned out, I ended up doing lots of work making development tools, since I was the hardware guy. I designed and built emulator cartridges [not much different in principle than GBA flash carts], as well as an eprom programmer for the 99/4A. I also modified the TI debugger so all I/O was through the serial port because our games were too hooked into the video system to share it with the debugger. I also added a disassembler to the TI debugger.

Q: So your new company focused mainly on the TI-99?

A: Our games were progressing fine, although mine was behind because of all of the support development I needed to do. However, the business guys weren't having much success. In fact, by mid-summer they had raised exactly *zero* dollars. Keep in mind, they had income from a computer store they were running, while we had quit our jobs. Their only additional expense was to install a phone line in their store that they answered as "Sofmachine." Of course, there was expense for preparing and printing up the limited partnership prospectus. Needless to say, we were getting nervous. Jim was married with two kids, so he was burning through his savings

at a high rate. And I had taken out a personal loan, co-signed with my dad, and it wasn't going to float me too much longer.

In the middle of the summer, Sofmachine was contacted by Atarisoft. Atarisoft had been buying the rights to port the popular full-size arcade games to game consoles and home computers of the day. And Atarisoft wanted us to convert three games: *Jungle Hunt*, *Pole Position*, and *Vangard*. We agreed to do so, at $35,000 for each game—except for *Pole Position*, which we managed to get $50,000 to do. So we started coding in earnest. Meanwhile, absolutely *no* funding of Sofmachine was happening. So Jim Dramis and I decided that the business guys needed to be out of the corporation because it would be unfair for them to get 40 percent of the Atarisoft revenue for doing *nothing*. We had delivered 99/4A games to be manufactured and marketed, but they hadn't delivered the means to manufacture and market them. This was complicated even more because Garth was a former high school student of one of the business people, whose name was Bill Games. In fact, Bill had recruited Garth to TI as a co-op student. Garth didn't think we should kick out the business types, but eventually he relented. We had a meeting and we agreed to pay all expenses incurred in the limited partnership offering, as well as other tangible expenses—phone and copy costs—plus five percent of the Atarisoft contract. Everyone agreed, and they signed their shares over to us programmers.

Q: I played those games quite a bit as a kid. It must have been fun working on arcade ports. How did that go?

A: We finished both *Jungle Hunt* and *Pole Position* at the end of 1983. About midway through the *Vangard* project, which I was doing, Atarisoft cancelled the project and agreed to pay half of the $35,000. When Jim finished *Jungle Hunt*, he accepted a job with IBM in Florida. Meanwhile, Garth and I were waiting on Atarisoft to decide whether they wanted us to port *Pole Position* to the ColecoVision game console. We really wanted that because we knew the game and already had the graphics. And, while the ColecoVision used a Z80 CPU, it had a TMS9918A video chip—the same as the TI 99/4A. I had already reverse-engineered the ColecoVision and generated a schematic. In fact, I designed a TMS9900-based single-board computer [SBC] that attached to the ColecoVision expansion bus, and used DMA to access the ColecoVision memory space. That way, we were able to use a slightly modified version of our 99/4A debugger that was ported to the SBC. In fact, Garth even modified the 9900 disassembler so it would disassemble the Z80 code in a ColecoVision cartridge. We were all set to make some easy money on the *Pole Position* conversion. But the video game industry was in the midst of imploding, so Atarisoft decided they didn't want to do this project. Garth moved back to

California and took a job with a defense contractor, and I used my Home Computer connections to get a job back at TI in their Central Research Laboratories [CRL].

Q: So you went full circle. What was the CRL all about?

A: At TI's CRL, I joined the Optical Processing branch, which was researching and developing the DMD. This is a light modulator technology along the lines of an LCD. It uses an array of small mirrors—17 microns on a side originally, now 14 microns—to display an image. This image is magnified by projection optics. TI now refers to this technology as DLP [*Digital Light Processing*], and it is used in over 50 percent of the conference room portable projectors and almost all of the digital cinema installations. In 1990, this technology was moved out of CRL and spun into its own operating group. While in CRL, I was the systems engineer for DMD, even though I didn't actually have a degree. In 1989, thirteen years after graduating high school, I received my BSEE from the University of Texas at Dallas. Of course, TI paid for my books and tuition while I worked on my degree.

I worked at TI as an employee until 1995. When I left, I had a project lined up with Cyrix, which was making X86 clone products at the time. This project lasted about a year, and just after it was completed, I got a call from the DLP guys, and they needed some help for about six months. I ended up doing contract engineering for them for five years. Then, they decided I either needed to become an employee or leave. So I left.

Q: Now tell me a little something about the company you founded and are still involved with at this time.

A: I formed Urbonix, Inc. (http://www.urbonix.com), which in reality had existed as a DBA [*Doing Business As*] since 1995. Before I cut the cord with TI, I was contacted by a company on the East coast, Dimensional Media Associates, who was getting ready to produce a 3D display using TI's DLP. This company, now LightSpace Technologies (http://www.lightspacetech.com), is still developing and marketing this product. Urbonix designed and built the first prototypes for the DMD display boards, as well as the image processor/formatter board that is used in the Z|1024 product that you see on the LightSpace Technologies Web site. My company, Urbonix, currently has a contract with Texas Instruments, where I am developing and supporting FPGA-based boards and peripherals for ASIC emulation.

Q: Thank you very much for your time.

A: Through all of this, I'm still interested in game design. It's been a pleasure; thank you.

Niels Bauer: Niels Bauer Software Design

Niels Bauer has been programming since he was 10 years old. He owns Niels Bauer Software Design and is studying law at the University of Freiburg in Germany. Niels Bauer Software Design (http://www.nbsd.de), located in Germany, has concentrated on complex (but still easy to learn) games. One of their best games, *Smugglers 2*, is an elite-like game from a strategic point of view. It features a lot of new ideas, such as crew management, boarding enemy ships, attacking planets, treasure hunting, and smuggling. If you want to make a game in the *Smugglers* universe under the loose guidance of this company, get in touch with them. You can reach them via the Web sites just mentioned or by e-mail, at contact@nbsd.de. Niels Bauer is now working on *Smugglers 3.*

Q: You founded Niels Bauer Software Design in 1999. Was it hard for a single person to develop the games alone?

A: In two years, I finished three games. Unfortunately, they weren't very successful. In spring of 2001, I wanted to leave the game business and do something else. Finally, I decided to make only one more game, *Smugglers*, and just for myself and nobody else. I decided to use Delphi because I wanted to concentrate 100 percent on the gameplay. I wanted a game that I would really like to play myself, even after weeks of development. When the game was finished, after about one month I showed it to some friends, and they immediately became addicted. Suddenly I became aware of the potential of the game and decided to release it. As you can see from this little story, the most difficult part of working alone is keeping yourself motivated until you have the first hit. *Smugglers 2* is the last game where I wrote most of the code myself. In the future, I will concentrate more on the business and design part.

Q: I've noticed that *Smugglers* has been a cover mount on some computer magazines. How easy or difficult was it to achieve this?

A: I would say it was very difficult and pure luck that I got the necessary contacts. I sent e-mails to many magazines, but from most I didn't even get a reply. The main [reason] for this could have been that *Smugglers 1* didn't have cool graphics and you needed to play the game to become addicted. Those editors became addicted and so they made a very good offer that I couldn't turn down, but unfortunately, from the feedback I got this is very uncommon.

Q: What do you think made *Smugglers* so popular?

A: Well, this is a difficult question. There are a lot of elite-like games out there. Unfortunately, most are too complex to be understood by the casual player. Even [I], as an experienced player, have problems with most. *Smugglers*, on the other hand, is

very easy to learn and play. With the short interactive tutorial, you can really start off immediately. On the other hand, it could have been so successful because it provided the player with a lot of freedom while still keeping the complexity low. For example, he can be a trader, a smuggler, a pirate, or even fight for the military. Or, for example, you can fly capital ships and attack planets. These are a lot of options. What I especially liked was the opportunity to receive ranks and medals depending on your own success. The last time I saw something like this was in *Wing Commander 1*, and this was a while back.

Q: You released *Smuggler 2* recently. Any projects for the future?

A: Yes, definitely. The team [has] already begun work on an online version. This time we say goodbye to the menu system used in previous *Smugglers* titles and use a very nice top-down view of the universe. I am very excited about the possibility of such a game.

Q: From a developer's perspective, what do you think of the game industry at this moment?

A: I feel very sorry for it. Where [have] all the cool games like *Pirates*, *Wing Commander*, *Civilization*, *Ultima 7*, and *Elite* gone to? I can tell you. They all landed in the trashcan because they don't have high-tech graphics. Only those games with the best graphics get bought these days in huge masses, and unfortunately, these games are the least fun and have the most bugs. I can't imagine a single game—except *Counter-Strike* and that was a mod—that I really liked to play for longer than a couple of hours. I don't believe I can change this with *Smugglers*, but maybe I can provide a safe haven for some people who feel like I do. Considering the attention I got for *Smugglers*, it might not be a few.

Q: Any final advice to the starting game developer?

A: Concentrate on the gameplay. I needed two years to understand that it's not C++ and DirectX that make a game cool. There are thousands of those games out there. What makes a game really good are two important factors:

1. It's extremely easy to learn. (Your mother needs to be able to play it right off.)
2. You need to like it to play it yourself all day long.

Someone said in a book, which I unfortunately don't remember [the name of] now, that you most likely need to make 10 crappy games before you will finally make a good game. This is definitely true.

André LaMothe: Xtreme Games LLC

André LaMothe has been in the computing industry for more than 24 years. He has worked in just about every field of computing and he even worked for NASA. He currently owns Xtreme Games LLC, a computer games publishing company. Xtreme Games LLC was founded five years ago and develops and publishes games for the PC, Palm, and Pocket PC platforms.

Q: At this time, with gamers wanting 3D environments with cube mapping and realistic particle systems, what game type do you think a small developer would have more luck with?

A: That's really hard to say. Even if a small developer makes a game better than *Quake III*, it really doesn't matter since it's nearly impossible to get distribution these days, and publishers screw developers with percentage rates of 5 to 10 percent being common. So my advice is, simply make what you want to play.

Q: Being Xtreme Games LLC, a publisher, what are the minimum requirements for publishing a game with you?

A: That the game be of professional quality, bug-free, and competitive with other value games on the market.

Q: With the new growth of Xtreme Games LLC, what kind of games would you be more interested in seeing?

A: Value sports games, 3D games leveraging the Genesis engine, etc., and quality Palm and Pocket PC games.

Q: What steps are involved? And what is the process from the point that a developer gives you a complete game to retail distribution?

A:

1. The game is tested until all bugs are removed.
2. The packaging of the product is created.
3. Buyers at chains make purchase orders for the product.
4. The product is manufactured and units are shipped to distribution points and warehouses.
5. The product is shelved.
6. The money for the product is paid. (It takes three to six months.)
7. Royalties are dispersed.

Q: From a developer's perspective, what do you think of the current state of the industry at this time?

A: Very bad. I'm sorry to say, corporate America has got into it really deep now, and completely taken the fun out of game development. Programmers work 100+ hours a week trying to meet impossible schedules dictated by marketing, distribution, and manufacturing that aren't even "real," and in the end 99 percent of all games don't even break even. On top of that, game programmers are not paid well; their average pay is less than programmers that are nowhere near as technically skilled but work in more mainstream software endeavors like Internet, database, etc. The problem with the entire game development industry is that the people running it still to this day don't understand it. If the developers ran it, we would all be a lot happier. Just because we are nerds doesn't mean we aren't smarter than MBAs when it comes to business. They better not ever let us in charge. Instead of a business that is replete with failure, huge losses, and dismal earnings to gross revenues, we would actually make money!

Q: Do you have any final advice to the small developer who wants to try to get into this challenging industry?

A: Don't think about how to make "them" happy; just do what makes you happy, stay focused, and finish what you start. Keep this up and sooner or later something good has to happen.

Xtreme Games is always looking for good products to license. If you're interested, contact us at:

Xtreme Games LLC
http://www.xgames3d.com
info@xgames3d.com

Summary

You have been through a crash course in software publishing, and this was just the tip of the iceberg. There are many options, many contracts, and many publishers you need to check, and that's just the beginning. As you get more experience, you will start to easily recognize the good and bad contracts, as well as the good and bad publishers. So what are you waiting for? Finish the game and start looking!

References

Below are some URLs of publishing companies. Please note that neither I nor Premier Press recommend any one publisher over another; the list is alphabetical.

Codemasters: http://www.codemasters.com
E3: http://www.e3expo.com
ECTS: http://www.ects.com
eGames: http://www.egames.com
Game Developers Conference: http://www.gdconf.com
GarageGames: http://www.garagegames.com
MonkeyByte Games: http://www.mbyte.com
On Deck Interactive: http://www.odigames.com
RealArcade Games: http://realguide.real.com/games
Xtreme Games Conference: http://www.xgdx.com
Xtreme Games LLC: http://www.xgames3d.com

Chapter Quiz

You can find the answers to this chapter quiz in Appendix A, "Chapter Quiz Answers."

1. What is the first step you must take before attempting to get your game published?
   A. Evaluate the game.
   B. Sell the game.
   C. Test the game.
   D. Release the game.

2. What is the most important question to consider in a game before seeking a publisher?
   A. Is it challenging?
   B. Is it fun to play?
   C. Is it graphically attractive?
   D. Is it marketable?

3. What is the second most important aspect of a game?
   A. Graphics
   B. Sound
   C. Music
   D. Input

4. What is an important factor of gameplay, in the sense of a beginning, middle, and ending, that must be considered?
   A. Progression
   B. Goals
   C. Difficulty
   D. Continuity

5. What adjective best describes a best-selling game?
   A. Large
   B. Complex
   C. Cute
   D. Addictive

6. What is an NDA?
   A. Never Diverge Anonymity
   B. No Disco Allowed
   C. Non-Disclosure Agreement
   D. Non-Discussion Agreement

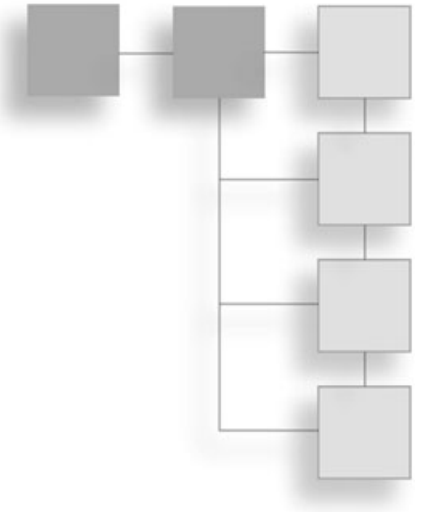
7. What is a software bug?
   A. An error in the source code
   B. A mistake in the design
   C. A digital life form
   D. A tracking device

8. What term describes a significant date in the development process?
   A. Deadline
   B. Milestone
   C. Achievement
   D. Release

9. Who created the game *Smugglers 2*?
   A. Niels Bauer
   B. André LaMothe
   C. John Carmack
   D. Ellie Arroway

10. For whom should you create a game for the purpose of entertainment?
    A. Yourself
    B. Gamers
    C. Publishers
    D. Marketers

Epilogue



I tend to say this each time I reach this point, but I can honestly say that this book has been the most enjoyable book I have written so far. Exploring the vast feature set of the Allegro library has been an absolute blast, and I am grateful to have had the opportunity to write this book on such a fascinating subject. I hope you have enjoyed it, too!

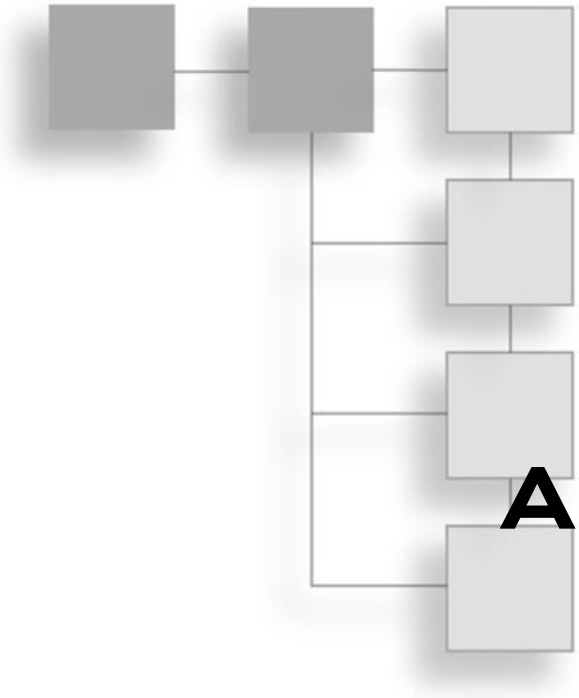
Although I do not know you personally, I have gotten to know many readers through online forums, so there is a certain feeling of coming full circle at this point. I hope you have found this book not just helpful, but invaluable as a reference and enjoyable to read. I have strived to cover all the bases of 2D game programming, and I hope you have enjoyed it.

Although every effort was made to ensure that the content and source code presented in this book is error-free, it is possible that errors in print or bugs in the sample programs might have missed scrutiny (especially when multiple compilers are involved, as was the case here). If you have any problems with the source code, sample programs, or general theory in this book, please let me know! You can contact me at support@jharbour.com. I'll do my best to help you work though any problems (and I'll try to respond within a day or so). I also welcome constructive criticism and comments that you might have regarding the content of this book. Reader feedback was the reason for this dramatic revision to a book that was once based on Windows and DirectX, but is now cross-platform and based on open-source tools!

Finally, whether you are an absolute beginner or a seasoned professional, I welcome you to visit my online forum at http://www.jharbour.com to share your games, ideas, and questions with other Allegro fans! Membership is free and open to the public.

As always, I look forward to hearing from you!

This page intentionally left blank

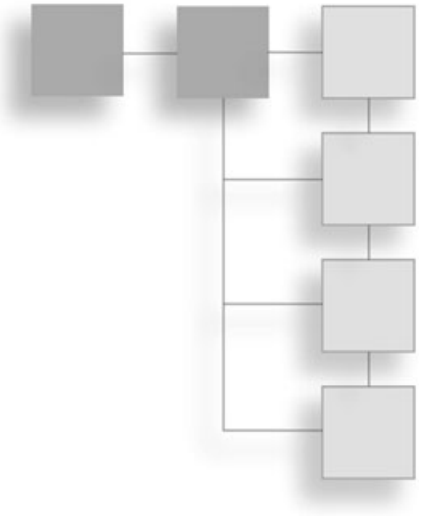


PART IV

Appendixes

Welcome to Part IV of *Game Programming All in One, 2nd Edition*. Part IV includes seven appendixes that provide reference information for your use, including some useful tables, an ASCII chart, a list of helpful books and Web sites, an overview of hexadecimal and binary numbering systems, a tutorial on configuring Allegro and compiling the Allegro library, and an overview of the included CD-ROM.

Appendix A



Chapter Quiz Answers

Chapter 1

1. What programming language is used in this book?
   A. C

2. What is the name of the free multi-platform game library used in this book?
   C. Allegro

3. What compiler can you use to compile the programs in this book?
   D. All of the above

4. Which operating system does Allegro support?
   D. All of the above

5. Which of the following is a popular strategy game for the PC?
   C. *Real War*

6. What is the most important factor to consider when working on a game?
   C. Gameplay

7. What is the name of the free open-source IDE/compiler included on the CD-ROM?
   B. Dev-C++

8. What is the name of the most popular game development library in the world?
   C. DirectX

9. Which of the following books discusses the gaming culture of the late 1980s and early 1990s with strong emphasis on the exploits of id Software?
   A. *Masters of Doom*
10. According to the author, which of the following is one of the best games made in the 1980s?
   D. *Starflight*

Chapter 2

1. What game features an Avatar and takes place in the land of Brittania?
   B. *Ultima VII: The Black Gate*
2. GNU is an acronym for which of the following phrases?
   A. GNU is Not Unix
3. What is the primary Web site for Dev-C++?
   B. http://www.bloodshed.net
4. What is the name of the compiler used by Dev-Pascal?
   A. GNU Pascal
5. What is the name of the powerful automated update utility for Dev-C++?
   D. WebUpdate
6. What are the Dev-C++ update packages called?
   B. DevPaks
7. What distinctive feature of Dev-C++ sets it apart from commercial development tools?
   D. All of the above
8. What is the name of the game programming library featured in this chapter?
   D. Allegro
9. What function must be called before you use the Allegro library?
   C. `allegro_init()`
10. What statement must be included at the end of `main()` in an Allegro program?
   B. `END_OF_MAIN()`

Chapter 3

1. What is the term used to describe line-based graphics?
   A. Vector
2. What does CRT stand for?
   C. Cathode Ray Tube
3. What describes a function that draws a simple geometric shape, such as a point, line, rectangle, or circle?
   B. `Graphics Primitive`
4. How many polygons does the typical 3D accelerator chip process at a time?
   C. 1
5. What is comprised of three small streams of electrons of varying shades of red, green, and blue?
   D. Pixel
6. What function is used to create a custom 24- or 32-bit color?
   A. `makecol`
7. What function is used to draw filled rectangles?
   D. `rectfill`
8. Which of the following is the correct definition of the circle function?
   A. `void circle(BITMAP *bmp, int x, int y, int radius, int color);`
9. What function draws a set of curves based on a set of four input points stored in an array?
   C. `spline`
10. Which text output function draws a formatted string with justification?
    D. `textprintf_justify`

Chapter 4

1. What is the primary graphics drawing function used to draw the tanks in *Tank War*?
   A. `rectfill`
2. What function in *Tank War* sets up a bullet to fire it in the direction of the tank?
   C. `fireweapon`

3. What function in *Tank War* updates the position and draws each projectile?
   D. `updatebullet`
4. What is the name of the organization that produced GCC?
   A. Free Software Foundation
5. How many players are supported in *Tank War* at the same time?
   B. 2
6. What is the technical terminology for handling two objects that crash in the game?
   C. Collision detection
7. What function in *Tank War* keeps the tanks from colliding with other objects?
   B. `clearpath`
8. Which function in *Tank War* helps to find out whether a point on the screen is black?
   A. `getpixel`
9. What is the standard constant used to run Allegro in windowed mode?
   D. `GFX_AUTODETECT_WINDOWED`
10. What function in Allegro is used to slow the game down?
    C. `rest`

Chapter 5

1. Which function is used to initialize the keyboard handler?
   B. `install_keyboard`
2. What does ANSI stand for?
   C. American National Standards Institute
3. What is the name of the array containing keyboard scan codes?
   A. `key`
4. Where is the real stargate located?
   C. Colorado Springs, Colorado
5. Which function provides buffered keyboard input?
   C. `readkey`

6. Which function is used to initialize the mouse handler?
   A. `install_mouse`
7. Which values or functions are used to read the mouse position?
   A. `mouse_x` and `mouse_y`
8. Which function is used to read the mouse x and y mickeys for relative motion?
   D. `get_mouse_mickeys`
9. What is the name of the main `JOYSTICK_INFO` array?
   B. `joy`
10. Which struct contains joystick button data?
    C. `JOYSTICK_BUTTON_INFO`

Chapter 6

1. What is the best way to get started creating a new game?
   D. Play other games to engender some inspiration.
2. What types of games are full of creativity and interesting technology that PC gamers often fail to notice?
   A. Console games
3. What phrase best describes the additional features and extras in a game?
   C. Bells and whistles
4. What is usually the most complicated core component of a game, also called the graphics renderer?
   D. The game engine
5. What is the name of an initial demonstration of a game that presents the basic gameplay elements before the actual game has been completed?
   B. Prototype
6. What is the name of the document that contains the blueprints for a game?
   C. Design document
7. What are the two types of game designs presented in this chapter?
   A. Mini and complete

8. What does NPC stand for?
   D. Non-Player Character
9. What are the chances of a newcomer finding a job as a full-time game programmer or designer?
   D. Negligible
10. What is the most important aspect of game development?
    A. Design

Chapter 7

1. What does "blit" stand for?
   B. Bit-block transfer
2. What is a DHD?
   C. Dial home device
3. How many pixels are there in an 800×600 screen?
   A. 480,000
4. What is the name of the object used to hold a bitmap in memory?
   D. `BITMAP`
5. Allegorically speaking, why is it important to destroy bitmaps after you're done using them?
   C. Because the trash will pile up over time.
6. Which Allegro function has the potential to create a black hole if used improperly?
   A. `acquire_bitmap`
7. What types of graphics files are supported by Allegro?
   B. BMP, PCX, LBM, and TGA
8. What function is used to draw a scaled bitmap?
   B. `stretch_blit`
9. Why would you want to lock the screen while drawing on it?
   A. If it's not locked, Allegro will lock and unlock the screen for every draw.
10. What is the name of the game you've been developing in this book?
    D. *Tank War*

Chapter 8

1. What is the term given to a small image that is moved around on the screen?
   B. Sprite
2. Which function draws a sprite?
   A. `draw_sprite`
3. What is the term for drawing all but a certain color of pixel from one bitmap to another?
   C. Transparency
4. Which function draws a scaled sprite?
   A. `stretch_sprite`
5. Which function draws a vertically-flipped sprite?
   B. `draw_sprite_v_flip`
6. Which function draws a rotated sprite?
   D. `rotate_sprite`
7. Which function draws a sprite with both rotation and scaling?
   B. `rotate_scaled_sprite`
8. What function draws a pivoted sprite?
   C. `pivot_sprite`
9. Which function draws a pivoted sprite with scaling and vertical flip?
   A. `pivot_scaled_sprite_v_flip`
10. Which function draws a sprite with translucency (alpha blending)?
    B. `draw_trans_sprite`

Chapter 9

1. Which function draws a standard sprite?
   C. `draw_sprite`
2. What is a frame in the context of sprite animation?
   A. A single image in the animation sequence

3. What is the purpose of a sprite handler?
   A. To provide a consistent way to animate and manipulate many sprites on the screen

4. What is a struct element?
   D. A variable in a structure

5. Which term describes a single frame of an animation sequence stored in an image file?
   B. Tile

6. Which Allegro function is used frequently to erase a sprite?
   A. `rectfill`

7. Which term describes a reusable activity for a sprite that is important in a game?
   D. Behavior

8. Which function converts a normal sprite into a run-length encoded sprite?
   B. `get_rle_sprite`

9. Which function draws a compiled sprite to a destination bitmap?
   C. `draw_compiled_sprite`

10. What is the easiest (and most efficient) way to detect sprite collisions?
    A. Bounding rectangle intersection

Chapter 10

1. Does Allegro provide support for background scrolling?
   A. Yes, but the functionality is obsolete.

2. What does a scroll window show?
   A. A small part of a larger game world

3. Which of the programs in this chapter demonstrated bitmap scrolling for the first time?
   C. *ScrollScreen*

4. Why should a scrolling background be designed?
   D. To achieve the goals of the game

5. Which process uses an array of images to construct the background as it is displayed?
   C. Tiling
6. What is the best way to create a tile map of the game world?
   A. By using a map editor
7. What type of object comprises a typical tile map?
   C. Numbers
8. What was the size of the virtual background in the *GameWorld* program?
   A. 800×800
9. How many virtual backgrounds are used in the new version of *Tank War*?
   B. 1
10. How many scrolling windows are used in the new *Tank War*?
    C. 2

Chapter 11

1. Why is it important to use a timer in a game?
   A. To maintain a consistent frame rate
2. Which Allegro timer function slows down the program using a callback function?
   D. `rest_callback`
3. What is the name of the function used to initialize the Allegro timer?
   B. `install_timer`
4. What is the name of the function that creates a new interrupt handler?
   D. `install_int`
5. What variable declaration keyword should be used with interrupt variables?
   C. `volatile`
6. What is a process that runs within the memory space of a single program but is executed separately from that program?
   C. Thread
7. What helps protect data by locking it inside a single thread, preventing that data from being used by another thread until it is unlocked?
   A. Mutex

8. What does pthread stand for?
   C. Posix Thread

9. What is the name of the function used to create a new thread?
   B. `pthread_create`

10. What is the name of the function that locks a mutex?
    D. `pthread_mutex_lock`

Chapter 12

1. What is the home site for Mappy?
   C. http://www.tilemap.co.uk

2. What kind of information is stored in a map file?
   A. Data that represent the tiles comprising a game world

3. What name is given to the graphic images that make up a Mappy level?
   D. Tiles

4. What is the default extension of a Mappy file?
   C. FMP

5. Where does Mappy store the saved tile images?
   B. Inside the map file

6. What is one example of a retail game that uses Mappy levels?
   B. *Hyperspace Delivery Boy*

7. What is the recommended format for an exported Mappy level?
   D. Text map data

8. Which macro in Mappy fills a map with a specified tile?
   A. Solid Rectangle

9. How much does a licensed copy of Mappy cost?
   D. It's free!

10. Which MappyAL library function loads a Mappy file?
    A. `MapLoad`

Chapter 13

1. In which game genre does the vertical shooter belong?
   A. Shoot-em-up
2. What is the name of the support library used as the vertical scroller engine?
   C. MappyAL
3. What are the virtual pixel dimensions of the levels in *Warbirds Pacifica*?
   D. 640×48,000
4. What is the name of the level-editing program used to create the first level of *Warbirds Pacifica*?
   B. Mappy
5. How many tiles comprise a level in *Warbirds Pacifica*?
   A. 30,000
6. Which of the following games is a vertical scrolling shooter?
   B. *Mars Matrix*
7. Who created the artwork featured in this chapter?
   C. Ari Feldman
8. Which MappyAL function loads a map file?
   B. `MapLoad`
9. Which MappyAL function removes a map from memory?
   D. `MapFreeMem`
10. Which classic arcade game inspired *Warbirds Pacifica*?
    C. *1942*

Chapter 14

1. Which term is often used to describe a horizontal-scrolling game with a walking character?
   B. Platform
2. What is the name of the map-editing tool you have used in the last several chapters?
   A. Mappy

3. What is the identifier for the Mappy block property representing the background?
   A. BG1
4. What is the identifier for the Mappy block property representing the first foreground layer?
   A. FG1
5. Which dialog box allows the editing of tile properties in Mappy?
   D. Block Properties
6. Which menu item brings up the Range Alter Block Properties dialog box?
   B. Range Edit Blocks
7. What is the name of the MappyAL struct that contains information about tile blocks?
   C. `BLKSTR`
8. What MappyAL function returns a pointer to a block specified by the (x,y) parameters?
   A. `MapGetBlock`
9. What is the name of the function that draws the map's background?
   A. `MapDrawBG`
10. Which MappyAL block struct member was used to detect collisions in the sample program?
    C. `tl`

Chapter 15

1. What is the name of the function that initializes the Allegro sound system?
   A. `install_sound`
2. Which function can you use to play a sound effect in your own games?
   C. `play_sample`
3. What is the name of the function that specifically loads a RIFF WAV file?
   B. `load_wav`
4. Which function can be used to change the frequency, volume, panning, and looping properties of a sample?
   D. `adjust_sample`

5. What function would you use to shut down the Allegro sound system?
   B. `remove_sound`
6. Which function provides the ability to change the overall volume of sound output?
   A. `set_volume`
7. What is the name of the function used to stop playback of a sample?
   D. `stop_sample`
8. Within what range must a panning value remain?
   D. 0 to 255
9. What parameter should you pass to `install_sound` to initialize the standard digital sound driver?
   C. `DIGI_AUTODETECT`
10. What is the name of the function that plays a sample through the sound mixer?
    B. `play_sample`

Chapter 16

1. What is the shorthand term for an Allegro data file?
   B. datafile
2. What compression algorithm does Allegro use for compressed datafiles?
   A. LZSS
3. What is the command-line program that is used to manage Allegro datafiles?
   D. dat.exe
4. What is the Allegro datafile object struct called?
   B. `DATAFILE`
5. What function is used to load a datafile into memory?
   D. `load_datafile`
6. What is the data type format shortcut string for bitmap files?
   C. `BMP`
7. What is the data type constant for wave files, defined by Allegro for use in reading datafiles?
   C. `DAT_SAMPLE`

8. What is the `dat` option to specify the type of file being added to the datafile?
   A. `-t <type>`

9. What is the `dat` option to specify the color depth of a bitmap file being added to the datafile?
   C. `-bpp <depth>`

10. What function loads an individual object from a datafile?
    D. `load_datafile_object`

Chapter 17

1. Which company developed the FLI/FLC file format?
   A. Autodesk

2. Which product first used the FLI format?
   C. Animator

3. Which product premiered the more advanced FLC format?
   A. Animator Pro

4. What is the common acronym used to describe both FLI and FLC files?
   D. FLIC

5. Which function plays an FLIC file directly?
   A. `play_fli`

6. How many FLIC files can be played back at a time by Allegro?
   A. 1

7. Which function loads an FLIC file for low-level playback?
   C. `open_fli`

8. Which function moves the animation to the next frame in an FLIC file?
   A. `next_fli_frame`

9. What is the name of the variable used to set the timing of FLIC playback?
   D. `fli_timer`

10. What is the name of the variable that contains the bitmap of the current FLIC frame?
    B. `fli_bitmap`

Chapter 18

1. Which of the following is *not* one of the three deterministic algorithms covered in this chapter?

   C. Conditions

2. Can fuzzy matrices be used without multiplying the input memberships? Why or why not?

   A. No, it is absolutely necessary to multiply the input memberships.

3. Which type of system solves problems that are usually solved by specialized humans?

   A. Expert system

4. Which type of intelligence system is based on an expert system, but is capable of determining fractions of complete answers?

   B. Fuzzy logic

5. Which type of intelligence system uses a method of computing solutions for a hereditary logic problem?

   C. Genetic algorithm

6. Which type of intelligence system solves problems by imitating the workings of a brain?

   D. Neural network

7. Which of the following uses predetermined behaviors of objects in relation to the universe problem?

   B. Deterministic algorithm

8. Which type of deterministic algorithm "fakes" intelligence?

   C. Random motion

9. Which type of deterministic algorithm will cause one object to follow another?

   A. Tracking

10. Which type of deterministic algorithm follows preset templates?

    D. Patterns

Chapter 19

1. What is the study of angles and their relationships to shapes and various other geometries?
   D. Trigonometry
2. What is the name of the C function that calculates cosine?
   B. `cos`
3. What is the name of the C function that calculates sine?
   A. `sin`
4. What is the name of the C function that calculates tangent?
   A. `tan`
5. Which C function calculates the inverse sine?
   D. `asin`
6. Which C function calculates the inverse tangent?
   C. `atan`
7. What does a set intersection contain?
   C. The elements that are contained in both sets
8. What does a function differentiation return?
   A. The slope of the function at any given position
9. What is the opposite of function differentiation?
   C. Integration
10. What Greek letter is most often used in calculations of degrees or radians of a circle?
    C. Pi

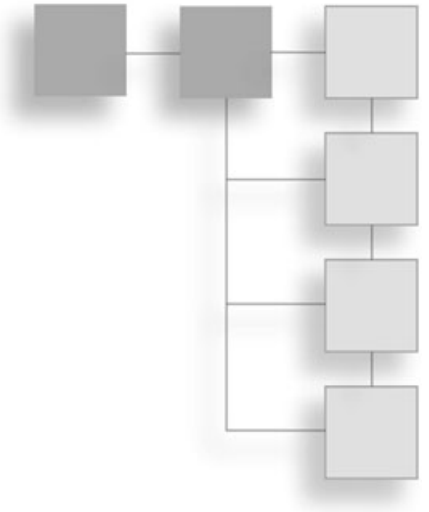
Chapter 20

1. What is the first step you must take before attempting to get your game published?
   A. Evaluate the game.
2. What is the most important question to consider in a game before seeking a publisher?
   C. Is it graphically attractive?

3. What is the second most important aspect of a game?
   B. Sound
4. What is an important factor of gameplay, in the sense of a beginning, middle, and ending, that must be considered?
   D. Continuity
5. What adjective best describes a best-selling game?
   D. Addictive
6. What is an NDA?
   C. Non-Disclosure Agreement
7. What is a software bug?
   A. An error in the source code
8. What term describes a significant date in the development process?
   B. Milestone
9. Who created the game *Smugglers 2*?
   A. Niels Bauer
10. For whom should you create a game for the purpose of entertainment?
    A. Yourself

This page intentionally left blank

Appendix B



Useful Tables

This appendix includes an ASCII table and three mathematical tables containing integral equations, derivative equations, and inertia equations.

Integral Equations Table

$\int x^n dx = \frac{x^{(n+1)}}{n+1} + C$
$\int \frac{1}{x} dx = \ln\|x\| + C$
$\int e^x dx = e^x + C$
$\int b^x dx = \frac{b^{(r+1)}}{\ln(b)} + C$
$\int \ln(x) dx = x\ln(x) - x + C$
$\int \sin(x) dx = -\cos(x) + C$
$\int \cos(x) dx = s\ln(x) + C$
$\int \tan(x) dx = -\ln\|\cos(x)\| + C$
$\int \arcsin(x) dx = x\arcsin(x)x + \sqrt{1-x^2} + C$
$\int \arccos(x) dx = x\arccos(x)x - \sqrt{1-x^2} + C$
$\int \arctan(x) dx = x\arctan(x)x - \frac{1}{2}\ln(1+x^2) + C$

Derivative Equations Table

$(u^x)' = xu^{n-1}$
$(e^x)' = x'e^x$
$1n(u)' = \frac{u'}{u}$
$\sin(u)' = u'\cos(u)$
$\cos(u)' = -u'\text{sen}(u)$
$\tan(u)' = \frac{u'}{\cos^2(u)}$

Inertia Equations Table

Object	Inertia Equations
Solid cylinder (horizontal axis)	$I = \frac{1}{2}mr^2$
Solid cylinder (vertical axis)	$I = \frac{1}{4}mr^2 + \frac{1}{12}ml^2$
Ring (horizontal axis)	$I = mr^2$
Ring (vertical axis)	$I = \frac{1}{2}mr^2 + \frac{1}{12}ml^2$
Empty sphere	$I = \frac{2}{3}mr^2$
Solid sphere	$I = \frac{2}{5}mr^2$
Cone	$I = \frac{3}{10}mr^2$

ASCII Table

This is a standard ASCII chart of character codes 0 to 255. To use an ASCII code, simply hold down the ALT key and type the value next to the character in the table to insert the character. This method works in most text editors; however, some editors are not capable of displaying the special ASCII characters (codes 0 to 31).

Char	Value	Char	Value	Char	Value
null	000	∟	028	8	056
☺	001	↔	029	9	057
☻	002	▲	030	:	058
♥	003	▼	031	;	059
♦	004	space	032	<	060
♣	005	!	033	=	061
♠	006	"	034	>	062
•	007	#	035	?	063
◘	008	$	036	@	064
○	009	%	037	A	065
◙	010	&	038	B	066
♂	011	'	039	C	067
♀	012	(	040	D	068
♪	013	)	041	E	069
♫	014	*	042	F	070
☼	015	+	043	G	071
►	016	,	044	H	072
◄	017	-	045	I	073
↕	018	.	046	J	074
‼	019	/	047	K	075
¶	020	0	048	L	076
§	021	1	049	M	077
▬	022	2	050	N	078
↨	023	3	051	O	079
↑	024	4	052	P	080
↓	025	5	053	Q	081
→	026	6	054	R	082
←	027	7	055	S	083

Char	Value	Char	Value	Char	Value
T	084	u	117	û	150
U	085	v	118	ù	151
V	086	w	119	ÿ	152
W	087	x	120	Ö	153
X	088	y	121	Ü	154
Y	089	z	122	¢	155
Z	090	{	123	£	156
[	091	|	124	¥	157
\	092	}	125	Pts	158
]	093	~	126	*f*	159
^	094	⌂	127	á	160
_	095	Ç	128	í	161
`	096	ü	129	ó	162
a	097	é	130	ú	163
b	098	â	131	ñ	164
c	099	ä	132	Ñ	165
d	100	à	133	ª	166
e	101	å	134	º	167
f	102	ç	135	¿	168
g	103	ê	136	⌐	169
h	104	ë	137	¬	170
i	105	è	138	½	171
j	106	ï	139	¼	172
k	107	î	140	¡	173
l	108	ì	141	«	174
m	109	Ä	142	»	175
n	110	Å	143	░	176
o	111	É	144	▒	177
p	112	æ	145	▓	178
q	113	Æ	146	│	179
r	114	ô	147	┤	180
s	115	ö	148	╡	181
t	116	ò	149	╢	182

Char	Value	Char	Value	Char	Value
╖	183	╪	216	·	249
╕	184	┘	217	·	250
╣	185	┌	218	√	251
║	186	█	219	ⁿ	252
╗	187	▄	220	²	253
╝	188	▌	221	■	254
╜	189	▐	222		255
╛	190	▀	223		
┐	191	α	224		
└	192	ß	225		
┴	193	Γ	226		
┬	194	π	227		
├	195	Σ	228		
─	196	σ	229		
┼	197	µ	230		
╞	198	τ	231		
╟	199	Φ	232		
╚	200	Θ	233		
╔	201	Ω	234		
╩	202	δ	235		
╦	203	∞	236		
╠	204	φ	237		
═	205	ε	238		
╬	206	∩	239		
╧	207	≡	240		
╨	208	±	241		
╤	209	≥	242		
╥	210	≤	243		
╙	211	⌠	244		
╘	212	⌡	245		
╒	213	÷	246		
╓	214	≈	247		
╫	215	°	248		

This page intentionally left blank

Appendix C

Numbering Systems: Binary and Hexadecimal

There are three numbering systems commonly used in computer programming—binary, decimal, and hexadecimal. The binary numbering system is called Base-2 because it has only two digits: 0 and 1.The decimal system is called Base-10; it is the one with which you are most familiar because it is used in everyday life. The hexadecimal system is called Base-16 and is comprised of the numerals 0–9 and the letters A–F to represent values from 0–15. Computers use the binary system exclusively in the hardware, but to make programming easier, compilers support decimal and hexadecimal (and the little-used Octal numbering system—Base-8).

Binary

Binary numbers use the Base-2 system, in which the numbers are represented by digits of either 0 or 1. This is the system the computer uses to store all the data in memory. Each digit in the number represents a power of two. Table C.1 shows the values in the binary system.

Table C.1 Binary System

Position	Digit
1	0
2	1

The best way to read a binary number is right to left; the first digit is to the far right and the last digit is to the far left. The number 1101, read from right to left, has the order 1, 0,

1, 1. The position of each digit determines the value of that digit, and each position is twice as large as the previous (with the first digit representing 0 or 1). Table C.2 provides a breakdown.

Table C.2 Binary Values Table

Position	Value
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512
11	1,024
12	2,048
13	4,096
14	8,192
15	16,384
16	32,768
17	65,536
18	131,072
19	262,144
20	524,288
21	1,048,576
22	2,097,152
23	4,194,304
24	8,388,608
25	16,777,216
26	33,554,432
27	67,108,864
28	134,217,728
29	268,435,456
30	536,870,912
31	1,073,741,824
32	2,147,483,648

Using this table you can decode any binary number as long as you remember to read the number from right to left and add up each value. How about an example?

The number 10101110 can be decoded as:

```
0 * 1 = 0
1 * 2 = 2
1 * 4 = 4
1 * 8 = 8
0 * 16 = 0
1 * 32 = 32
0 * 64 = 0
1 * 128 = 128
```

Adding up the values 2 + 4 + 8 + 32 + 128 = 174. Anyone can read a binary number in this way, as long as it is read from right to left. With a little practice you will be converting binary numbers in your head in only a few seconds.

Decimal

You have probably been using the decimal system since childhood and you don't even think about counting numbers in specific digits because you have been practicing for so long. The Base-10 numbering system is a very natural way for humans to count because we have 10 fingers. But from a scientific point of view, it's possible to decode a decimal number by adding up its digits, as you do for binary.

For example, try to decode the number 247. What makes this number "two hundred forty seven?" The decimal system has 10 digits (thus the name *decimal*) that go from 0–9. Just as with the binary system, you decode the number from right to left (although it is read from left to right in normal use). Because each digit in 247 represents a value to the power of 10, you can decode it as:

```
7 * 1 = 7
4 * 10 = 40
2 * 100 = 200
```

Adding up the values 7 + 40 + 200 = 247. Now this is asinine for the average person, but for a programmer, this is a good example for understanding the other numbering systems and it is a good lesson.

Hexadecimal

The hexadecimal system is a Base-16 numbering system that uses the numbers 0–9 and the letters A–F (to represent the numbers 10–15, since each position must be represented

by a single digit). Decoding a hexadecimal number works exactly the same as it does for binary and decimal—from right to left, by adding up the values of each digit. For reference, Table C.3 provides a breakdown of the values in the hexadecimal system.

Table C.3 Hexadecimal Table

Value	Digit
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

To read a hexadecimal number (in other words, to convert it to decimal so a human can understand it), just decode the hexadecimal digits from right to left using the table of values and multiply each digit by a successive power of 16. It was easy to calculate Base-2 multipliers, but it is a little more difficult with hexadecimal. Since hex numbers increase quickly in value, there are usually very few digits in a hex number—just look at the huge number after only 10 digits! Table C.4 shows multipliers for Base-16.

Using this newfound information, you should be able to decode any hex number. For instance, the hex number 9C56D is decoded like this:

D: 1 * 13 = 13
6: 16 * 6 = 96
5: 256 * 5 = 1,280
C: 4,096 * 12 = 49,152
9: 65,536 * 9 = 589,824

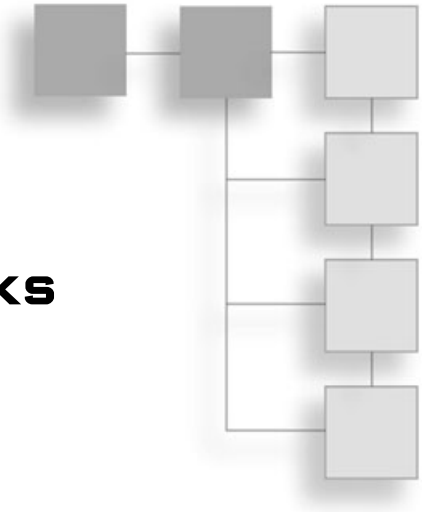
Table C.4 Hexadecimal Table

Position	Multiplier
0	1
1	16 (16^1)
2	16 (16^2)
3	256 (16^3)
4	4096 (16^4)
5	65,536 (16^5)
6	1,048,576 (16^6)
7	16,777,216 (16^7)
8	268,435,456 (16^8)
9	4,294,967,296 (16^9)
10	68,719,476,736 (16^10)

Adding these values results in 13 + 96 + 1,280 + 49,152 + 589,824 = 640,365. Because these numbers grow so quickly in Base-16, they are usually grouped in twos and fours when humans need to read them. Any hex number beyond four digits is usually too much for the average programmer to calculate in his head. However, the small size of a hex number usually means it cuts out several digits from a decimal number, which makes for more efficient storage in a file system. For this reason, hex numbers are used in compression and cryptography.

This page intentionally left blank

Appendix D



Recommended Books and Web Sites

Here is a collection of sites related to game development that I highly recommend.

All in One Support on the Web

I have set up a Web site to provide online support for this book. This site features an overview, sample programs and screenshots, related links, and downloads: http://www. jharbour.com/allinone.

In addition, I have set up an online forum dedicated to game development, focused on providing additional support for this book from other readers and fans of Allegro. The online forums are at http://www.jharbour.com/forums.

Game Development Web Sites

Here are some excellent game development sites on the Web that I visit frequently:

Allegro Home Site: http://www.talula.demon.co.uk/allegro
GameDev LCC: http://www.gamedev.net
FlipCode: http://www.flipcode.com
MSDN DirectX: http://msdn.microsoft.com/directx
MSDN Visual C++: http://msdn.microsoft.com/visualc
Game Development Search Engine: http://www.gdse.com
CodeGuru: http://www.codeguru.com
Programmers Heaven: http://www.programmersheaven.com
AngelCode.com: http://www.angelcode.com
OpenGL: http://www.opengl.org

NeHe Productions: http://nehe.gamedev.net
NeXe: http://nexe.gamedev.net
Game Institute: http://www.gameinstitute.com
Game Developer: http://www.gamedeveloper.net
Wotsit's Format: http://www.wotsit.org

Publishing, Game Reviews, and Download Sites

Keeping up with all that is happening is a daunting task, to say the least. New things happen every minute all over the world, and hopefully, the next set of links will help you keep up to date with it all.

Thomson/Course Technology: http://www.course.com
Premier Press: http://www.premierpressbooks.com
Games Domain: http://www.gamesdomain.com
Blue's News: http://www.bluesnews.com
Happy Puppy: http://www.happypuppy.com
Download.com: http://www.download.com
Tucows: http://www.tucows.com
Slashdot: http://slashdot.org
Imagine Games Network (IGN): http://www.ign.com

Engines

Sometimes it is not worth reinventing the wheel. There are several good engines, both 2D and 3D, out there. Following are some of the engines I have had the pleasure (or pain) of working with that I want to recommend to you. Some are expensive, but then again, some are free. See which is best for you and start developing.

Touchdown Entertainment (LithTech Engine): http://www.lithtech.com
Jet3D: http://www.jet3d.com
Genesis3D: http://www.genesis3d.com
RenderWare: http://www.renderware.com
Crystal Space: http://crystal.sourceforge.net

Independent Game Developers

You know, almost everyone started as you are starting, by reading books and magazines or getting code listings from friends or relatives. Some of the developers in the following list have worked hard to complete some great games.

Longbow Digital Arts: http://www.longbowdigitalarts.com
Spin Studios: http://www.spin-studios.com
Positech Games: http://www.positech.co.uk

Samu Games: http://www.samugames.com
QUANTA Entertainment: http://www.quanta-entertainment.com
Satellite Moon: http://www.satellitemoon.com
Myopic Rhino Games: http://www.myopicrhino.com

Industry

If you want to be in the business, you need to know the business. Reading magazines and visiting association meetings will help you for sure.

Game Developers Magazine: http://www.gdmag.com
GamaSutra: http://www.gamasutra.com
International Game Developers Association: http://www.igda.com
Game Developers Conference: http://www.gdconf.com
Xtreme Game Developers eXpo: http://www.xgdx.com
Association of Shareware Professionals: http://www.asp-shareware.org
RealGames: http://www.real.com/games

Computer Humor

Here are some great sites to visit when you are looking for a good laugh.

Homestar Runner (Strong Bad!): http://www.homestarrunner.com
User Friendly: http://www.userfriendly.org
Geeks!: http://www.happychaos.com/geeks
Off the Mark: http://www.offthemark.com/computers.htm
Player Versus Player: http://www.pvponline.com

Recommended Books

I've provided a short description for each of the books in this list because they are either books I have written (plug!) or that I highly recommend and have found useful, relaxing, funny, or essential on many an occasion. You will find this list of recommended books useful as references to the C language and as complementary titles and references to subjects covered in this book, such as Linux and Mac game programming (with a few unrelated but otherwise interesting titles thrown in for good measure).

3D Game Engine Programming

Oliver Duvel, et al; Premier Press; ISBN 1-59200-351-6

"Are you interested in learning how to write your own game engines? With [this book] you can do just that. You'll learn everything you need to know to build your own game engine as a tool that is kept strictly separate from any specific game project, making it a tool that you can use again and again for future projects. You won't have to give a second

thought to your engine. Instead, you'll be able to concentrate on your game and the gameplay experience."

3D Game Programming All in One

Kenneth Finney; Premier Press; ISBN 1-59200-136-X

An introduction to programming 3D games using the Torque engine by GarageGames.

AI Techniques for Game Programming

Mat Buckland; Premier Press; ISBN 1-931841-08-X

"[This book] takes the difficult topics of genetic algorithms and neural networks and explains them in plain English. Gone are the torturous mathematic equations and abstract examples to be found in other books. Each chapter takes you through the theory a step at a time, explaining clearly how you can incorporate each technique into your own games."

Beginner's Guide to DarkBASIC Game Programming

Jonathan S. Harbour and Joshua R. Smith; Premier Press; ISBN 1-59200-009-6

This book provides a good introduction to programming Direct3D, the 3D graphics component of DirectX, using the C language.

Beginning C++ Game Programming

Michael Dawson; Premier Press; ISBN 1-59200-205-6

"If you're ready to jump into the world of programming for games, [this book] will get you started on your journey, providing you with a solid foundation in the game programming language of the professionals. As you cover each programming concept, you'll create small games that demonstrate your new skills. Wrap things up by combining each major concept to create an ambitious multiplayer game. Get ready to master the basics of game programming with C++!"

Beginning DirectX 9

Wendy Jones; Premier Press; ISBN 1-59200-349-4

An excellent introduction to the new features in DirectX 9.

C Programming for the Absolute Beginner

Michael Vine; Premier Press; ISBN 1-931841-52-7

This book teaches C programming using the free GCC compiler as its development platform, which is the same compiler used to write Game Boy programs! As such, I highly recommend this starter book if you are just learning the C language. It sticks to the basics. You will learn the fundamentals of the C language without any distracting material or commentary—just the fundamentals of what you need to be a successful C programmer.

C++ Programming for the Absolute Beginner

Dirk Henkemans and Mark Lee; Premier Press; ISBN 1-931841-43-8

If you are new to programming with C++ and you are looking for a solid introduction, this is the book for you. This book will teach you the skills you need for practical C++ programming applications and how you can put these skills to use in real-world scenarios.

Character Development and Storytelling for Games

Lee Sheldon; Premier Press; ISBN 1-59200-353-2

"[This book] begins with a history of dramatic writing and entertainment in other media. It then segues to writing for games, revealing that while proven techniques in linear media can be translated to games, games offer many new challenges on their own, such as interactivity, non-linearity, player input, and more. It then moves beyond linear techniques to introduce the elements of the craft of writing that are particularly unique to interactive media. It takes us from the relatively secure confines of single-player games to the vast open spaces of virtual worlds and examines player-created stories, and shows how even here writers on the development team are necessary to the process, and what they can do to aid it."

Game Design: The Art and Business of Creating Games

Bob Bates; Premier Press; ISBN 0-7615-3165-3

This very readable and informative book is a great resource for learning how to design games—the high-level process of planning the game prior to starting work on the source code or artwork.

Game Programming for Teens

Maneesh Sethi; Premier Press; ISBN 1-59200-068-1

An excellent introduction to Windows game programming with DirectX.

High Score! The Illustrated History of Electronic Games

Rusel DeMaria and Johnny L. Wilson; McGraw-Hill/Osborne; ISBN 0-07-222428-2

This is a gem of a book that covers the entire video game industry, including arcade machines, consoles, and computer games. It is jam-packed with wonderful interviews with famous game developers and is chock full of color photographs.

Mac Game Programming

Mark Szymczyk; Premier Press; ISBN 1-931841-18-7

"Covering the components that make up a game and teaching you to program these components for use on your Macintosh, you will work your way through the development of a complete game. With detailed information on everything from graphics and sound to physics and artificial intelligence, [this book] covers everything that you need to know as you create your first game on your Mac."

Mathematics for Game Developers

Christopher Tremblay; Premier Press; ISBN 1-59200-038-X

"[This book] explores the branches of mathematics from the game developer's perspective, rejecting the abstract, theoretical approach in favor of demonstrating real, usable applications for each concept covered. Use of this book is not confined to users of a certain operating system or enthusiasts of particular game genres; the topics covered are universally applicable."

Microsoft C# Programming for the Absolute Beginner

Andy Harris; Premier Press; ISBN 1-931841-16-0

Using game creation as a teaching tool, this book teaches not only C#, but also the fundamental programming concepts you need to learn any computer language. You will be able to take the skills you learn from this book and apply them to your own situations. *Microsoft C# Programming for the Absolute Beginner* is a unique book aimed at the novice programmer. Developed by computer science instructors, the *Absolute Beginner* series is the ideal tool for anyone with little to no programming experience.

Microsoft Visual Basic .NET Programming for the Absolute Beginner

Jonathan S. Harbour; Premier Press; ISBN 1-59200-002-9

Whether you are new to programming with Visual Basic .NET or you are upgrading from Visual Basic 6.0 and looking for a solid introduction, this is the book for you. It teaches the basics of Visual Basic .NET by working through simple games that you will learn to create. You will acquire the skills you need for more practical Visual Basic .NET programming applications and learn how you can put these skills to use in real-world scenarios.

Linux Game Programming

Mark Collins, et al; Premier Press; ISBN 0-7615-3255-2

"This book offers Linux users the information they need to create a game using their OS of choice. Featuring an overview of the game development cycle, using tools and libraries, developing graphics applications, and adding extras such as sound, this book provides clear, concise information on developing games for and with the Linux OS."

Pocket PC Game Programming: Using the Windows CE Game API

Jonathan S. Harbour; Premier Press; ISBN 0-7615-3057-6

This book will teach you how to program a Pocket PC handheld computer using Visual Basic and Visual C++. It includes coverage of graphics, sound, stylus and button input, and even multiplayer capability. Numerous sample programs and games demonstrate the key topics you need to know to write complete Pocket PC games.

Programming Role Playing Games with DirectX, Second Edition

Jim Adams; Premier Press; ISBN 1-59200-315-X

"In the second edition of this popular book, you'll learn how to use DirectX 9 to create a complete role-playing game. Everything you need to know is included! You'll begin by learning how to use the various components of DirectX 9…. Once you have a basic understanding of DirectX 9, you can move on to building the basic functions needed to create a game—from drawing 2D and 3D graphics to creating a scripting system. Wrap things up as you see how to create an entire game—from start to finish!"

Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games

Neal and Jana Hallford; Premier Press; ISBN 0-7615-3299-4

This book is a fascinating overview of what it takes to develop a commercial-quality role-playing game, from design to programming to marketing. This is a helpful book if you would like to write a game like *Zelda*.

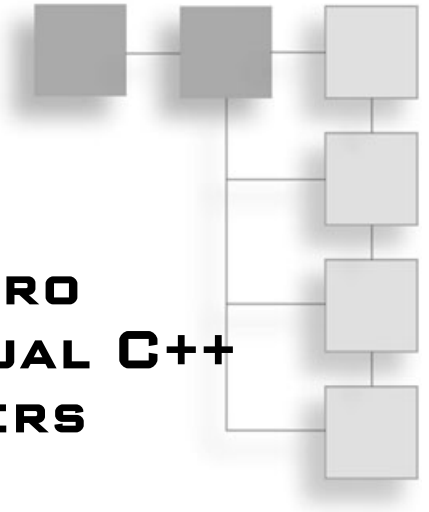
Visual Basic Game Programming with DirectX

Jonathan S. Harbour; Premier Press; ISBN 1-931841-25-X

This book is a comprehensive programmer's tutorial and a reference for everything related to programming games with Visual Basic. After a complete explanation of the Windows API graphics device interface meant to supercharge 2D sprite programming for normal applications, the book delves into DirectX 7.0 and 8.1 and covers every component of DirectX in detail, including Direct3D. Four complete games are included, demonstrating the code developed in the book.

This page intentionally left blank

Appendix E



Configuring Allegro for Microsoft Visual C++ and Other Compilers

This appendix provides instructions for configuring Allegro with the most popular compilers, including Microsoft Visual C++, KDevelop 3.0 (Linux), and Dev-C++ 5. It is amazing how there are so many compilers that support Allegro, and several of them are free! In the past, high-caliber development tools like these were very expensive and hard to find. These tutorials assume that you have already compiled the Allegro source code (per Appendix F, "Compiling the Allegro Source Code") or you have copied the headers and library files provided on the CD-ROM that accompanies this book.

I have pre-compiled the Allegro library and made it available. The easiest way to examine the configuration is to open one of the pre-configured source code projects from the CD-ROM. The second option is to read further and find out how to set up the compiler yourself. Linking is a complicated subject. If you run into any problems, be sure to refer to Appendixes E and F whenever necessary.

To make things as easy as possible (especially for those who are not as experienced with configuring compilers), I have included on the CD-ROM the completed project files for every program in the book for all three primary compilers that are supported: Dev-C++, Visual C++, and KDevelop. It is more common to use the dynamic library with Visual C++, so those projects all reference the dynamic library and require the DLL (alleg40.dll). Dev-C++ and KDevelop projects are configured for the statically linked library. If you examine the CD-ROM that accompanies this book, you'll find a folder called \sources, in which the source code projects for the book are separated into three subfolders: msvc, devcpp, and kdevelop. Within each of these folders you will find all of the projects for each chapter. You can simply open the projects directly if you do not want to configure the compilers yourself. The project files for Visual C++ have an extension of .dsw; project files for Dev-C++ have an extension of .dev; and project files for KDevelop have an extension of .kdevprj.

Microsoft Visual C++

The Microsoft compilers are all very similar in options and configuration, so this short tutorial is applicable to all recent versions of Visual C++. The dialog boxes might look slightly different from what is shown here in VC5 or VC7, but the process is simple enough that you should be able to adapt the basic concept to your needs.

These instructions take for granted that the DirectX SDK is installed on your system. Although DirectX 9 is available at the time of this writing, you really only need DirectX 8 to compile programs with Allegro. I'm not just talking about the run time—you need to install the DirectX SDK, which includes the header and library files. If you have not already installed it, refer to the CD-ROM folder called \libraries. Note that if you are using the dynamic version of Allegro, you don't need the DirectX SDK. You only need the DirectX SDK if you need to compile Allegro or if you plan to statically link it to your programs.

Here is how you can configure Visual C++ 6 to use the Allegro library. Create a new Win32 Application type project (see Figure E.1).

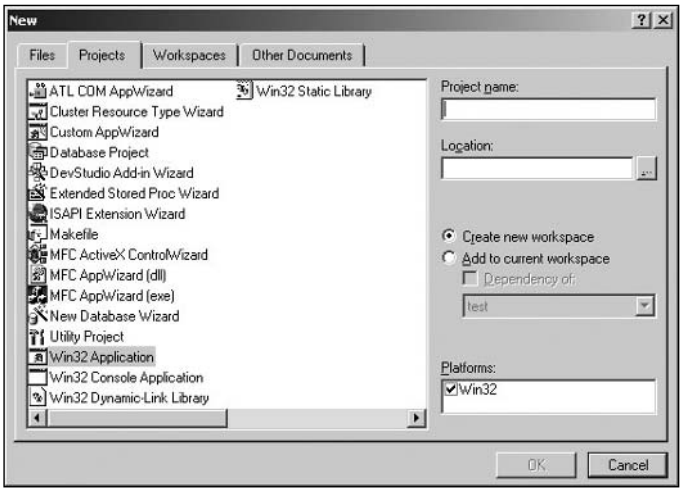


Figure E.1 The New Project dialog box in Visual C++ 6

Next, open the Project menu and select Settings to bring up the Project Settings dialog box (see Figure E.2). Click on the Link tab and look for the Object/Library Modules text field. Clear the entire field and type in **alleg.lib** in place of the other library files.

Now you need to make sure the linker can find Allegro. Add a new source file to the project and type in the following code. This program does very little, but it verifies that Allegro has been linked to your program.

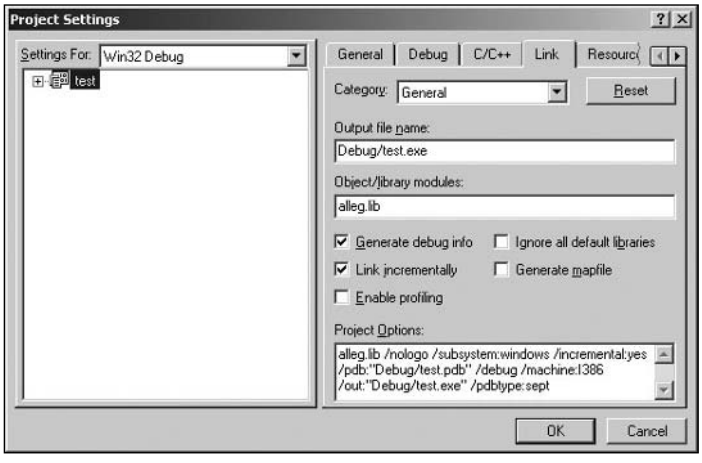


Figure E.2 The Project Settings dialog box in Visual C++ 6

```
#include "allegro.h"
int main(void)
{
    allegro_init();
    allegro_message("Welcome To Allegro!");
    return 0;
}
END_OF_MAIN();
```

If all goes as expected, the compilation output window should show "0 error(s), 0 warning(s)" (see Figure E.3), and upon running the program, you should see a message box with the phrase "Welcome To Allegro!" If there are any errors, be sure to check for typos, and then refer to Appendix F to verify that you have compiled and installed Allegro correctly.

If you want to use the static library of Allegro in Visual C++ (which is not usually the case), then you'll want to replace the single alleg.lib entry in the Object/Library Modules text box with the following entries. (Be sure to separate each with a space.)

```
alleg_s.lib
gdi32.lib
winmm.lib
ole32.lib
dxguid.lib
dinput.lib
ddraw.lib
dsound.lib
```

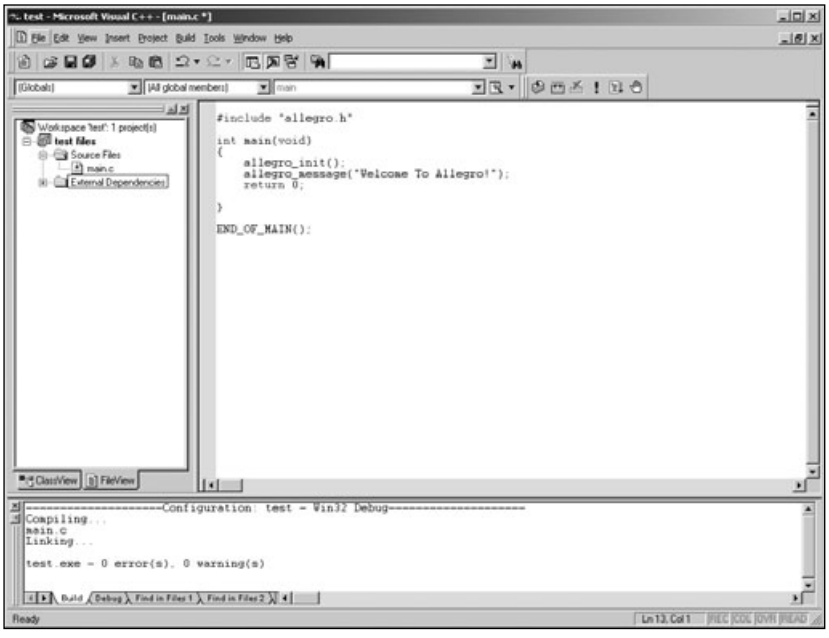


Figure E.3 Compiling an Allegro program with Visual C++ 6

You also need to tell Allegro to use the static library by including the following line at the top of any program that will be compiled static:

```
#define ALLEGRO_STATICLINK
```

Dev-C++

Dev-C++ was covered briefly in Chapter 2, but this is a more thorough explanation of configuring Allegro, assuming that you have already compiled the sources or copied the Allegro library files and headers as described in Appendix F. Dev-C++ comes with a WebUpdate tool that will automatically download and install the Allegro headers and library files into Dev-C++ so it's ready to use (covered in Chapter 2). But in this tutorial, I'm going to explain how to set up Dev-C++ without the benefit of WebUpdate.

Dev-C++ is a fully capable Windows compiler with support for all the usual Windows libraries (kernel32, user32, gdi32, and so on), including DirectX. If you have ever installed Microsoft's DirectX SDK, you'll recall how big it is—hundreds of files, hundreds of megabytes. There is a much smaller public domain implementation of DirectX 8 for the MinGW32 system (the version of GCC used by Dev-C++) included on the CD-ROM that accompanies this book. You will find it in the folder \directx, in the file dx80_mgw.zip. Extract the \include and \lib folders from dx80_mgw.zip inside the main Dev-C++ folder on your hard drive. (This will usually be C:\Dev-Cpp.) The easiest way to do this is to copy

dx80_mgw.zip to C:\Dev-Cpp and unzip it there, where the header and library files will be extracted into the existing include and lib folders. That's all there is to installing DirectX for Dev-C++ (or more specifically, for GCC). You might note that the header files are copyrighted by Microsoft, while the lib files actually have the usual GCC library extension of .a. This way, you can differentiate between the official Microsoft DirectX SDK and the public domain implementation.

You should note that MinGW32 (bundled with Dev-C++) also includes public domain implementations of the entire Windows 32-bit API. This means that someone actually wrote compatible versions of the entire Windows API for use with tools such as MinGW32 to make GCC compatible with Windows. This provides you with all the power of Visual C++ in a very tight, small package. Because I do not write Windows GUI apps with C/C++ (involving dialogs, windows, and controls, for which I use Visual Basic), I find Dev-C++ with DirectX, OpenGL, and other libraries (such as Allegro) to be an awesome alternative to Visual C++ for game programming. The real benefit to Visual C++ is the comprehensive documentation in the form of MSDN, the dialog editor/form designer, and other value-added features. Because Dev-C++ includes the MinGW32 implementation of the Win32 API, I find it useful to keep Internet Explorer open to the Microsoft Developer Network Web site at http://msdn.microsoft.com/visualc, where the latest edition of MSDN is available online (and is equivalent to the distributed version of the MSDN subscription).

I won't debate the fact that Microsoft produces an exemplary and untouchable C/C++ compiler and IDE for Windows in the form of Visual C++. What I find most convenient about Dev-C++ is the very small footprint in memory (only about 12 MB), the small install file, and the simple installer. Combined with the MinGW32 implementation of the Windows API and the third-party implementation of DirectX, you have a great little game development package for Windows.

Now that I have presented the pros and cons of using Dev-C++, allow me to explain how to set up Allegro for this compiler for either dynamic or static link. You must first install Allegro. Even if you install the Allegro DevPak, I still recommend copying the Allegro 4.0.3 library and header files over to C:\Dev-Cpp. The header files should be copied to C:\Dev-Cpp\include and the library files should be copied to C:\Dev-Cpp\lib. (Your path might be different depending on where you chose to install Dev-C++.) While I have included the library and dll files in \libraries on the CD-ROM, you'll want to copy the contents of \include from the CD-ROM to C:\Dev-Cpp\include as well. This is especially important if you are upgrading to a new version of Allegro. For instance, the DevPak included on the CD-ROM only supports Allegro 4.0.0, but you need 4.0.3 for the programs in this book. So even if you install the DevPak, you still need to copy the 4.0.3 includes and libraries over to C:\Dev-Cpp into their respective folders. If this whole process is confusing, just follow these simple guidelines:

- Copy CD-ROM\include*.* into C:\Dev-Cpp\include.
- Copy CD-ROM\libraries\devcpp*.* into C:\Dev-Cpp\lib.

If you still have trouble getting the sample programs to compile, refer to Appendix F to actually compile the Allegro sources. You will benefit from an install script included with Allegro that installs everything where it's supposed to go. I have provided the include and lib files on the CD-ROM so you can just copy them over and get up and running quickly. Compiling the sources is pretty easy, so you should give it a try even if your compiler is configured and working correctly.

Now I want to cover how to set up Dev-C++ for the static library. Run Dev-C++ and open the File menu, and then select New, Project to bring up the New Project dialog box (Figure E.4).

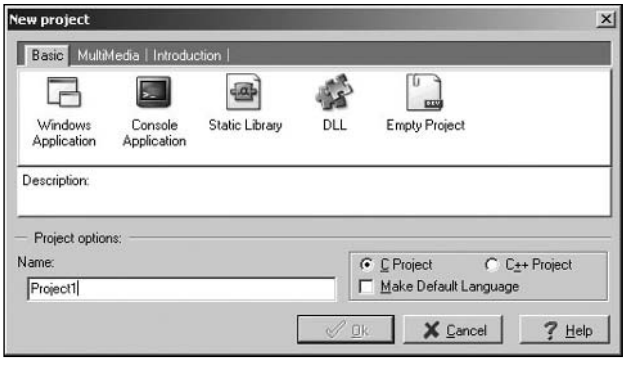


Figure E.4 The New Project dialog box in Dev-C++

If you have installed the Allegro DevPak as explained in Chapter 2, then you have the option of using one of the Allegro project templates (see Figure E.5).

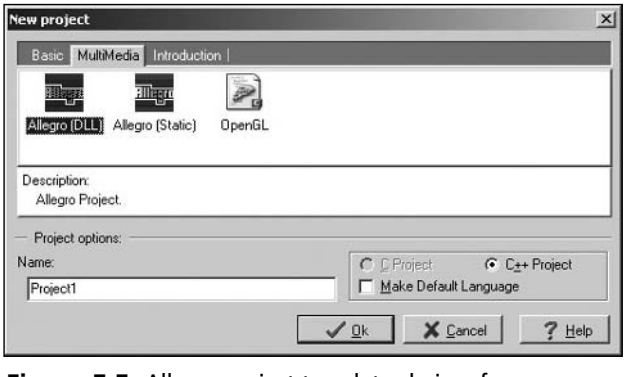


Figure E.5 Allegro project template choices for a new project in Dev-C++

If you choose to create an Allegro (Static) project, I need to inform you of a little bug in the template. The ALLEGRO_STATICLINK option is in the wrong box for the C compiler when the template is designed for the C++ compiler. Move -DALLEGRO_STATICLINK from the first options box to the second box, labeled C++ compiler, and then it will compile correctly (see Figure E.6). This bug might be fixed in a new release of the Allegro.DevPak, and it is not an issue at all if you configure the project yourself. (More on that in a minute.)

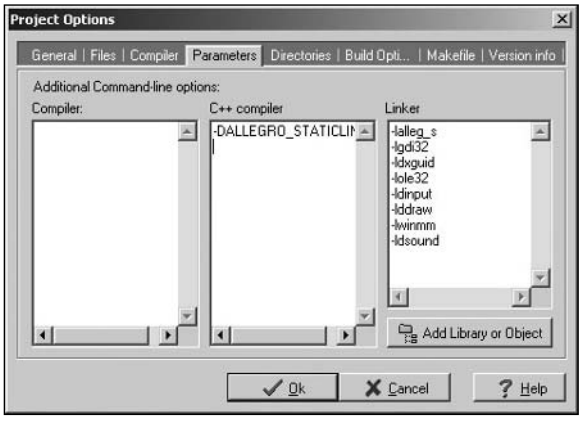


Figure E.6 The Project Options dialog box in Dev-C++

The other option is to configure a Dev-C++ project yourself, without using the templates. You would need to do this, for instance, if you have not installed the Allegro.DevPak. The DevPaks are really useful because they install the appropriate files in the correct folders for compiling a program with the requisite library. But if you want to just create a standard project, here's how to do it.

First, fire up Dev-C++, and then click on File, New, Project to bring up the New Project dialog box (see Figure E.7). Choose Windows Application for the project type, and be sure to select C Project instead of the C++ option because you will be setting up the project for the standard C compiler. This will be very similar to the settings that were applied with the project template.

Once the project is created, open the Project Options dialog box and click on the Parameters tab. All you have to do to create a dynamically linked Allegro program is add -lalleg to the Linker box. Note that the -l is a linker switch that tells the linker to include the library file named liballeg.a. (Remember that the lib at the front and .a at the end are assumed.) You could also simply insert the actual filename with the full path if you want; for instance, -lc:\allegro\lib\liballeg.a is a valid option. It is generally a good idea to copy library files into the lib folder for the compiler you're using. In the case of Dev-C++, that folder is usually C:\Dev-Cpp\lib.

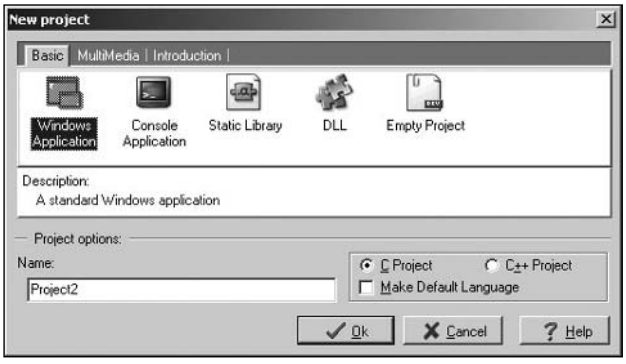


Figure E.7 Creating a new Windows Application project in Dev-C++

If you want to configure the project to use the static library, it requires two extra steps. In the first box, labeled Compiler, type in the static library option **-DALLEGRO_STATICLINK**. In the Linker box, enter the following options:

```
-lalleg_s
-lgdi32
-lwinmm
-lole32
-ldxguid
-ldinput
-lddraw
-ldsound
```

Figure E.8 shows the Project Options dialog box with the settings needed for a statically linked Allegro program.

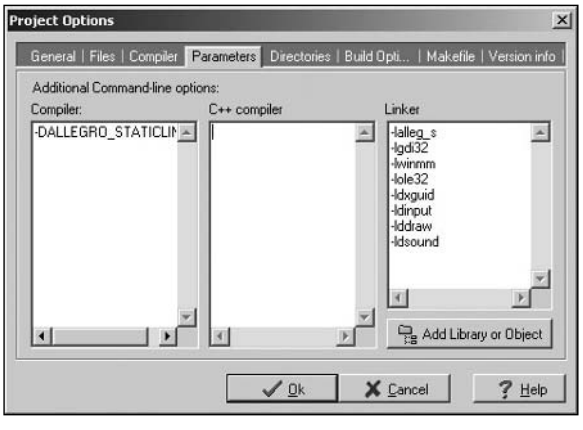


Figure E.8 Configuring Dev-C++ for a statically linked Allegro program

That's all there is to configuring the static library the manual way. Now you can write a short program to test the configuration and make sure Allegro is properly installed and the compiler is properly configured. Type the following program into the code window. (Delete any existing code first.)

```
#include "allegro.h"
int main(void)
{
    char version[80];
    allegro_init();
    sprintf(version, "Allegro library version = %s", allegro_id);
    allegro_message(version);
    return 0;
}
END_OF_MAIN();
```

Compile and run the program, and you should see a message pop up on the screen with the following text, which will indicate that Allegro is configured:

```
Allegro library version = Allegro 4.0.3, MinGW32.s
```

If you have any problems compiling the program at this point, it is most likely due to missing DirectX files, missing Allegro include files, or missing Allegro library files. You do not need alleg40.dll to run the static version. If you have double-checked these issues and you still have problems compiling, refer to Appendix F to do a full install of Allegro for Dev-C++.

KDevelop for Linux

The Linux operating system is a good choice for writing games with Allegro because the GCC compiler is always installed with the operating system, and you can type in programs with a simple text editor and get them to run with very little effort. However, Linux is not for the faint at heart, so if you are a beginner trying to get up to speed with Linux, you might have to pick up a book on using Linux. It's beyond the scope of this meager appendix to explain how to install KDevelop, the development tool used in this book for compiling Allegro programs for Linux. Assuming you have KDevelop already installed (as part of the KDE window system/user interface), then you can forge ahead. Even if you are using another window manager, such as GNOME, you can still run KDevelop by simply installing the KDE libraries. More than likely your distribution of choice provided KDevelop as an install option, or might have simply installed it with KDE automatically. If you want to download the latest version of KDevelop for your Linux box, browse to http://www.kdevelop.org. By the way, KDevelop is merely a front-end GUI for GCC (and a fine IDE at that).

Now you need to configure KDevelop for the Allegro library. If you haven't installed Allegro yet, jump to Appendix F and follow the instructions for compiling and installing Allegro under Linux. Because Allegro's installer script copies files into so many locations, it is really foolhardy to attempt a manual install by copying the includes and libraries yourself. Besides, that's the hard way. Compiling the sources is the single best way to install Allegro! Once you have done that, then return here and proceed to configure KDevelop as the following instructions explain.

First, fire up KDevelop, and then create a new project by opening the Project menu and selecting New. Choose a C terminal program (as shown in Figure E.9).

Figure E.9 Creating a new C terminal program in KDevelop

In the ApplicationWizard dialog box that appears, I recommend disabling the following three unneeded options:

- GNU-Standard-Files (INSTALL,README,COPYING...)
- User-Documentation
- Ism-File - Linux Software Map

However, you should keep the Generate Sources and Headers option selected, as shown in Figure E.10.

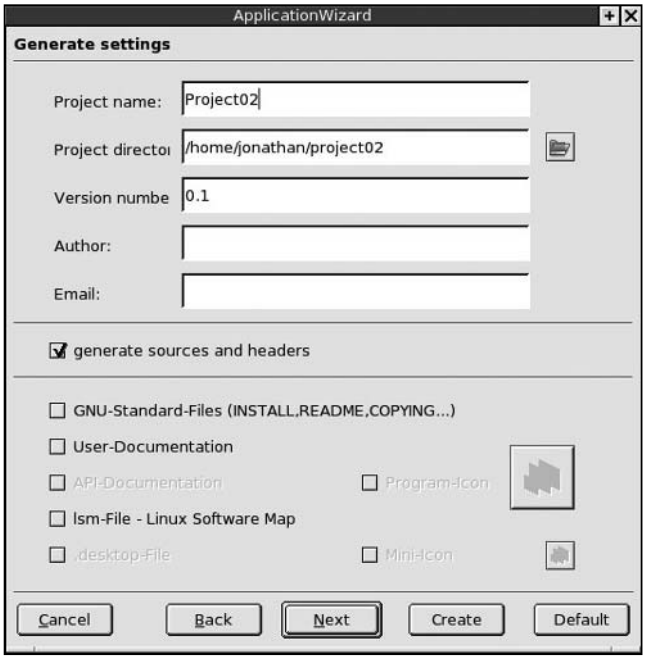


Figure E.10 Setting parameters in KDevelop's ApplicationWizard

You can skip the Version Control System Support dialog box. In the next two dialog boxes, turn off Headertemplate for .h-files and Headertemplate for .c-files, which clog up the source code. Finally, you will come to a Processes dialog box in the ApplicationWizard. Click on Create to build the new project, and ignore any obscure errors that appear regarding missing files. When you are finished, click on the Exit button. KDevelop will create a new project for you, as shown in Figure E.11.

Now you can set up the project for Allegro. Open the Project menu and select Options to bring up the Project Options dialog box. Click on the Linker Options icon on the left. Select the X11 and kdeui library check boxes, and type the following in the Additional Libraries text box:

```
-L/usr/local/lib
-L/usr/X11R6/lib
-lalleg
-lpthread
-lXxf86dga
-lXxf86vm
-ldl
```

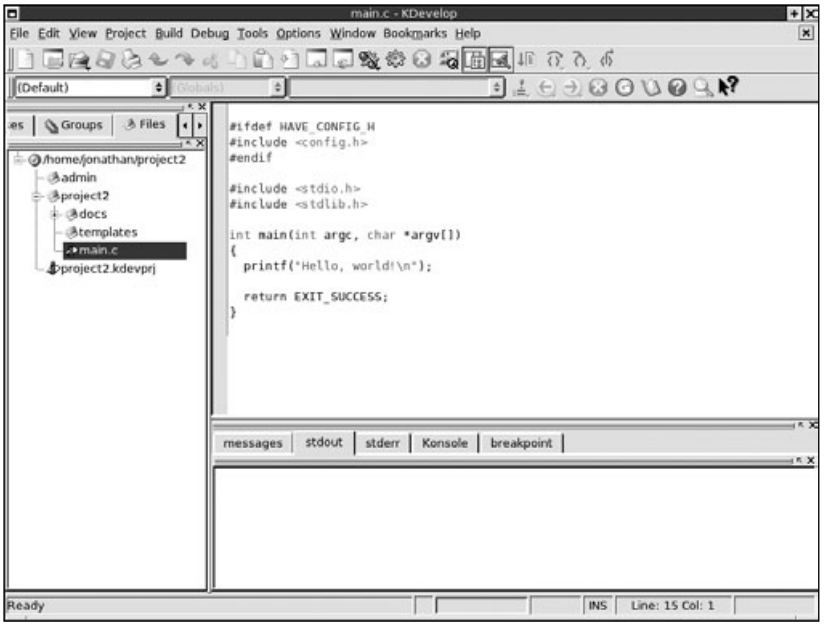


Figure E.11 The new C terminal program, ready to run in KDevelop

Note the uppercase L in the first two linker options; these tell the linker to include every library file found in the supplied folder name, if required by the smart linker (see Figure E.12).

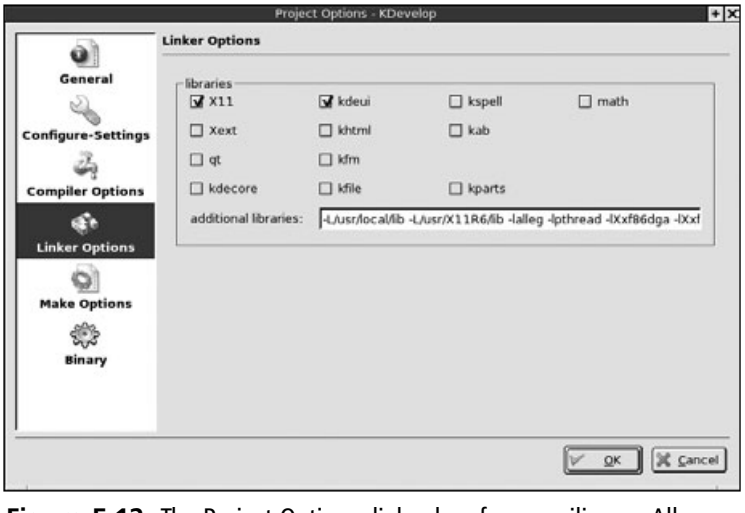


Figure E.12 The Project Options dialog box for compiling an Allegro program with KDevelop

Returning to the editor window, you can type in a program that actually demonstrates that the Allegro library is indeed working as expected. Here is a short program that will do just that.

```
#include "allegro.h"
int main(void)
{
    char version[80];
    allegro_init();
    sprintf(version, "Allegro library version = %s", allegro_id);
    allegro_message(version);
    return 0;
}
END_OF_MAIN();
```

If the project has been configured correctly and if Allegro was installed correctly, the program should compile and run with an output like the one shown in Figure E.13.

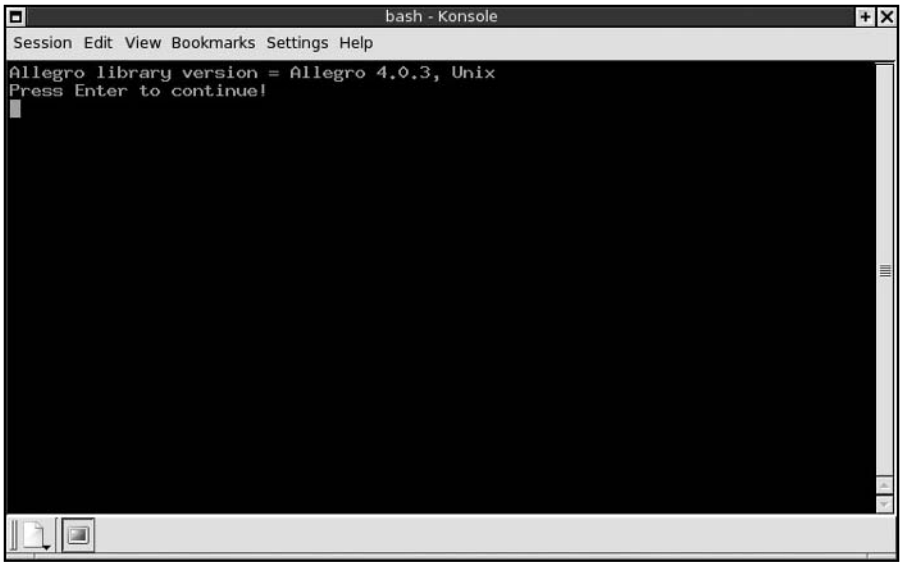


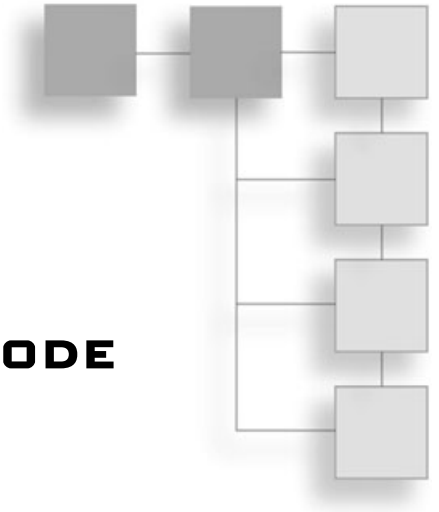
Figure E.13 Testing the Allegro program compiled with KDevelop

Final Comments

One final note on the Allegro game library: There are online discussions currently raging about the future of Allegro. It is very possible that future versions of Allegro will not be backward compatible with the 4.0.3 version used in this book. If that is the case, then I

suggest using 4.0.3 and not getting tied up trying to get a newer version to work. It's very possible that 5.0 and later will support 4.0.3 programs, but based on my experience with as recent a version as 4.1.12, which is not compatible with 4.0.3, it seems that the developers of Allegro are facing the dilemma of providing new functionality without breaking existing code. If you download a new version of Allegro, it will be up to you to get it to work. I am only officially supporting 4.0.3 (the most stable version, in my opinion) in this book.

Appendix F



Compiling the Allegro Source Code

This appendix will walk you through the process of compiling the Allegro source code. Allegro is an open-source game programming library that is distributed in source code form. You must compile Allegro in order to use it. There are some friendly people on the Web who have compiled Allegro for various operating systems, but it is best to compile Allegro yourself, if for no better reason than to verify that your compiler is compatible with the version of Allegro you intend to use. (I strongly recommend sticking with 4.0.3.) This is a fairly simple process that I'll walk you through for Microsoft Visual C++ 6.0, Borland C++ 5.5 (including C++Builder 3.0 or later), Bloodshed Dev-C++ 4.9 (or later), and KDevelop 2.1 (or later).

Note that Visual C++ 7.0 or later should be similar to 6.0; if you have problems, refer to the tutorials at http://www.allegro.cc or visit http://www.jharbour.com for assistance. A good collection of pre-compiled Allegro library files (and DLLs) is also located at http://www.allegro.cc/files, including makefiles for MSVC6, MSVC7, and Borland C++ Builder. If you have trouble compiling Allegro yourself, you can use these pre-built versions.

Microsoft Visual C++

There is good news and bad news about working with Visual C++. The good news is that Allegro compiles just fine under Visual C++. The bad news is that Microsoft's make utility was created just for Microsoft projects and does not support any type of standard makefile format.

So, you have to use the make.exe program that comes with Dev-C++ (or any version of GCC, for that matter). This shouldn't be a problem because Dev-C++ is rather small, and you can install a simple version of GCC by downloading it off the Web if you want, just to acquire the make.exe program. (Browse to http://gcc.gnu.org.) The Allegro makefiles

were designed specifically for the GCC make, not the make program that comes with the various compilers. If you haven't installed Dev-C++ but you want to use Visual C++ anyway, install Dev-C++ to the default folder, which should be C:\Dev-Cpp. After you have done so, add it to your path so the make program will work.

```
set path=C:\Dev-Cpp\bin;%path%
```

You also will need to add the path to Visual C++ in a moment.

Before you can compile the Allegro library, you must extract it from the ZIP file. If you look on the CD-ROM under \allegro, you will find a file called all403.zip. This file contains the complete Allegro 4.0.3 source code distribution for all platforms.

Extract the all403.zip file to the root of your hard drive (which is usually C:\) using WinZip, WinRar, or another archiving utility. Be sure to extract with the directory structure intact (which is usually the default with WinZip and similar programs). This will create a new folder on your root called C:\allegro. The Allegro sources (including tools and examples) will use up about 15.5 MB of disk space after they are extracted.

Open a command prompt window. (Select Start, Run, and type cmd.exe on most Windows systems.) Change to the C:\allegro folder.

```
CD \allegro
```

Now type in the following line at the command prompt:

```
fix msvc
```

This invokes the fix.bat batch file to configure the Allegro sources for Microsoft Visual C++. By default, the Allegro makefiles are set up to compile the standard optimized version of Allegro (which you would want to use for the release build of a game).

You are almost ready to compile Allegro. But first, have you set a path to your installation of Visual C++? The standard installation of Visual C++ includes a batch file that will configure the environment variables and paths for Visual C++. If you have installed Visual C++ to the default folder, you can use this command to set the path:

```
set path=C:\Program Files\Microsoft Visual Studio\VC98\Bin;%path%
```

Now you're ready to compile Allegro. There are three versions that you can compile (standard optimized version, debug version, and profiler version) for the three standard project types in a C/C++ compiler. If you want to compile the debug version, you must first set an environment variable.

```
set DEBUGMODE=1
```

That is followed by this command:

```
make lib
```

If you want to compile the profile version of Allegro, type the following lines:

```
set PROFILEMODE=1
make lib
```

For your purposes, you want to compile all three versions of Allegro, along with all the example programs. So type the following line:

```
make all
```

It will usually take several minutes for the entire Allegro library to compile, along with all of the support programs (such as dat.exe and grabber.exe) and example programs that demonstrate Allegro's features.

Finally, as I discussed in Appendix E, you can compile the static version of Allegro in order to link Allegro inside the executable file of your game. The static library can be compiled like so:

```
set STATICLINK=1
make lib
```

In general, just remember the `fix msvc` and `make all` commands, and you'll have no problems. After you have compiled the Allegro source code, you'll want to look inside \allegro \lib\msvc for the compiled library files and DLL, and then copy these files to your Visual C++ lib folder (usually located in \Program Files\Microsoft Visual Studio\VC98\Lib). You will also want to look in \allegro\include and copy all files and subfolders to the include folder for Visual C++ (usually located in \Program Files\Microsoft Visual Studio\VC98 \Include).

Borland C++/C++Builder

I did not formally support Borland C++ in the book because it was quite a feat just to keep my sanity with three different compilers! However, Allegro should compile for Borland C++ 5.5 (but not 5.0), as well as C++Builder 3.0 or later. The cool thing about this is that if you have been a Borland fan for many years, or you just prefer to use a compiler other than GCC, you can download the command-line compiler from Borland's Web site for free; it is perfectly suitable for compiling Allegro, as well as the source code from this book. (Browse to http://www.borland.com/products/downloads/download_cbuilder.html.) However, Borland C++ does not enjoy as much support from Allegro as MSVC and Dev-Cpp.

The process of compiling the Allegro library sources is the same as for Microsoft Visual C++, with one exception. First, you must configure the Allegro makefiles for Borland using this command:

```
fix bcc32
```

Aside from this change, you can follow the instructions in the previous section for compiling Allegro for Visual C++, because the remaining instructions are the same. After you have compiled the Allegro source code, you'll want to look inside \allegro\lib\bcc32 for the compiled library files and DLL, and copy these files to your Borland C++ lib folder. You will also want to look in \allegro\include and copy all files and subfolders to the include folder for Borland C++.

Dev-C++

Dev-C++ is the free integrated development environment provided by Bloodshed Software. It used throughout the book interchangeably with Visual C++ and KDevelop. The instructions for Dev-C++ are similar to Borland C++/C++Builder. You can follow the instructions laid out for Visual C++ with a single exception: You must configure the Allegro makefiles for Dev-C++ (which *really* means you need to configure the makefiles for MinGW32/GCC).

```
fix mingw32
```

From here, you can set the environment variables for compiling the standard, debug, and profile versions of Allegro (along with the optional `STATICLINK` option to create a statically-linked library file, if you want).

But first, have you set a path to your installation of Dev-C++? If you have a question about this, refer to Chapter 2 and Appendix E for instructions on how to install and configure Dev-C++. The default install folder is C:\Dev-Cpp, so you might need to set a path as follows:

```
set path=C:\Dev-Cpp\bin;%path%
```

From this point, follow the directions for Visual C++ to set the options and compile Allegro using the makefiles. After you have compiled the Allegro source code, you'll want to look inside \allegro\lib\mingw32 for the compiled library files and DLL, and copy these files to your Dev-C++ lib folder (usually located in \Dev-Cpp\lib). You will also want to look in \allegro\include and copy all files and subfolders to the include folder for Dev-C++ (usually located in \Dev-Cpp\include).

KDevelop for Linux

I'm going to have to assume you are somewhat familiar with Linux already because it is a little more difficult to compile Allegro under Linux than it is under Windows. But once you have Allegro extracted and you are in the Allegro folder, it's very easy and automated for the most part.

Open a command-shell window. Locate the Allegro sources on the CD-ROM in a file called allegro-4.0.3.tar.gz. You can extract the Allegro library sources using `gzip` and `tar` from the command line, or just use Nautilus or another GUI program that can read archive files.

Type this command first to extract the tar file out of the gz file:

```
gzip -d allegro-4.0.3.tar.gz
```

Next, type this to extract the files out of the tar archive:

```
tar -xf allegro-4.0.3.tar
```

If you look in the current folder, you should now see a subfolder called allegro-4.0.3. Move into this folder using `cd allegro-4.0.3`.

Now for the steps involved in configuring the project files for compilation. First, assuming you are in shell and currently in allegro-4.0.3, type this command:

```
./configure
```

This will configure the Allegro sources and makefiles and prepare them for the GCC compiler. After the configuration script runs, you should convert the files to UNIX format and set the makefile. This is not absolutely necessary, especially if you just extracted Allegro from a tar file, but it's worth knowing how to configure the sources for UNIX.

```
./fix.sh unix
```

Now you're ready to compile the Allegro library source code. It's best to just build all the versions (standard, debug, and profile) at the same time.

```
make all
```

The compilation might take several minutes because all of the utility programs (such as dat.exe and grabber.exe) are compiled, along with all of the example programs that demonstrate the features of Allegro.

If all goes well (compilation errors are rare due to the configuration script), you should soon be ready to compile Allegro programs using KDevelop. You can refer to Appendix E for instructions on how to set up an Allegro project in KDevelop. After the Allegro sources

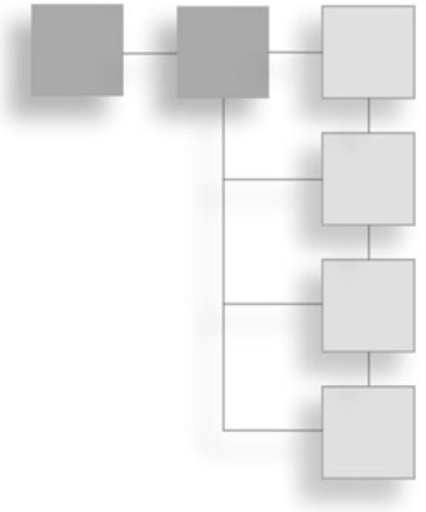
have been compiled, you will need to install Allegro into the proper directories for it to work properly with KDevelop and other development environments. (Make sure you have root access.) Type the following line:

```
su -c "make install"
```

This command will copy the Allegro library and header files into the shared locations. Next, you can install the manuals (man pages) and info docs using these commands:

```
su -c "make install-man"
su -c "make install-info"
```

Appendix G



Using the CD-ROM

The CD that comes with this book contains some important files you will want to use when you are working through the sample programs in the book. The CD comes preloaded with a very nice Autorun program that includes a menu for installing the various programs and files to your hard drive.

The most important files on the CD are the source code files for the sample programs in the book. The programs are stored in folders on the CD that are organized by chapter from the root \sources folder. Inside \sources, you will find the platform subfolders \sources\msvc, \sources\devcpp, and \sources\kdevelop, which contain the chapter folders \sources\msvc\chapter01, \sources\msvc\chapter02, and so on. I recommend that you copy the entire \sources folder for your particular platform/compiler to your hard drive and turn off the read-only property for all of the files so you will be able to peruse the sample projects for the book more easily. Zip files have also been provided for each compiler: sources_msvc.zip, sources_devcpp.zip, and sources_kdevelop.tar.gz.

Because this book focuses mainly on using Dev-C++ with the GCC compiler, Dev-C++ has also been provided on the CD-ROM in the \dev-cpp folder. You can run the installer from the Premier Press CD-ROM menu or directly off the CD by running the devcpp4980.exe installer. To save you time downloading DevPaks for Dev-C++, I have also included a folder called \devpaks, which contains the most useful updates for the content in this book.

The Allegro Game Library has been provided on the CD-ROM in the \allegro folder for both Windows (all403.zip) and Linux (allegro-4.0.3.tar.gz). I have also included the Allegro header files in \allegro\include. Most importantly, I have compiled the Allegro library for Visual C++, Dev-C++, and KDevelop. You can avoid the difficulty of compiling Allegro

yourself by simply copying the appropriate library files into \lib, where your compiler is located. The dynamic, static, and debug builds of Allegro have been provided in \allegro\ libraries.

Windows programmers will need to install the DirectX SDK. (Version 8 or later will work.) If you are using Visual C++, you will want to install Microsoft's official DirectX SDK. However, if you are using Dev-C++ and GCC, you will need a special version of DirectX that has been compiled into .a files (the library format used by GCC). The GCC version of DirectX 8 is located in \directx in a file called dx80_mgw.zip.

INDEX

Numbers

A

B

C

D

E

F

G

J

K

N

O

P

Q-R

S

T

U

V

W-Z

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.

Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.  This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker. signature of Ty Coon, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:
The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:
The enclosed disc is warranted by Thomson Course Technology PTR to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Thomson Course Technology PTR will provide a replacement disc upon the return of a defective disc.

Limited Liability:
The sole remedy for breach of this limited warranty shall consist entirely of replacement of the defective disc. IN NO EVENT SHALL THOMSON COURSE TECHNOLOGY PTR OR THE AUTHOR BE LIABLE FOR ANY other damages, including loss or corruption of data, changes in the functional characteristics of the hardware or operating system, deleterious interaction with other software, or any other special, incidental, or consequential DAMAGES that may arise, even if THOMSON COURSE TECHNOLOGY PTR and/or the author has previously been notified that the possibility of such damages exists.

Disclaimer of Warranties:
THOMSON COURSE TECHNOLOGY PTR and the author specifically disclaim any and all other warranties, either express or implied, including warranties of merchantability, suitability to a particular task or purpose, or freedom from errors. Some states do not allow for EXCLUSION of implied warranties or limitation of incidental or consequential damages, so these limitations mIGHT not apply to you.

Other:
This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Thomson Course Technology PTR regarding use of the software.