Game Mathematics

e-Institute Publishing, Inc.

Editor: Susan Nguyen
Cover Design: Adam Hoult

E-INSTITUTE PUBLISHING INC
www.gameinstitute.com

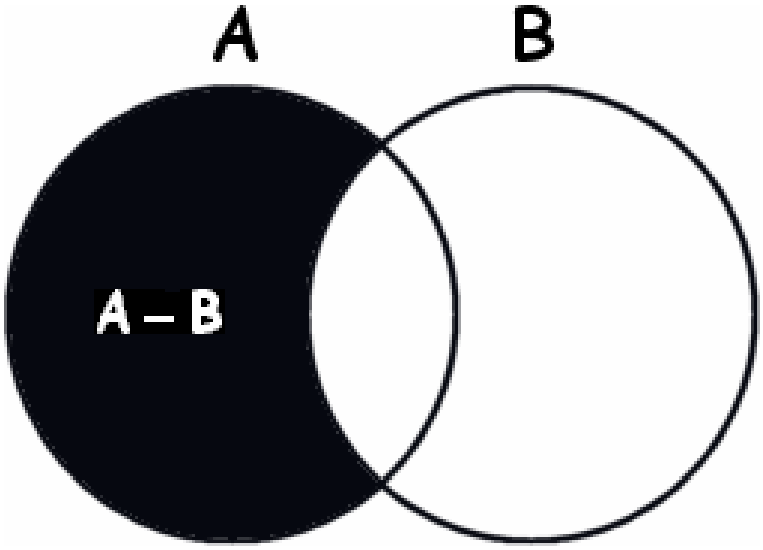
John DeGoes. *Game Mathematics*

E-INSTITUTE PUBLISHING titles are available for site license or bulk purchase by institutions, user groups, corporations, etc. For additional information, please contact the Sales Department at sales@gameinstitute.com

Table of Contents

Chapter One

Set Theory and Functions



Introduction

“Game Mathematics” refers to the mathematics that every game programmer needs to know in order to write cutting-edge games; whether two-dimensional or three-dimensional, strategy, fantasy, action or adventure. It is the math you need to know to develop games from Pong to Doom III™ or whatever other game idea is simmering in the back of your mind. In this lesson we will talk about why mathematics is so important to game developers and describe some of the interesting things you will be able to do when you finish this course. We will also begin to lay the framework for later chapters by introducing set theory and relating this branch of mathematics to the concept of mathematical functions, which will prove invaluable to us down the road. Mathematics is not always easy for a lot of students, so if something does not make sense or if you have a question that we do not cover in the material, please make a note of it. Later you can post to our course message board or save it until our real-time question and answer sessions. Do the exercises, play with the interactive programs and, above all, have fun. Remember, what you are learning now is going to make all the difference in the world when it comes time to write your very own games.

Today's games showcase some very technologically sophisticated software design. If you surf over to your favorite software cyber-store, and load up a virtual cart full of games, before long you will be basking in the glory of high-resolution, richly colored, vividly detailed scenes with fluid animation and three-dimensional, acoustically correct sound. These entertaining, interactive games present highly immersive worlds in which things look, sound, and act, to an ever-increasing degree, just like the real thing.

As an aspiring game developer, you should be probably wondering how it is that they can make games that are so realistic. Certainly a large part of the credit goes to the high-tech artists who design the objects in the game world using Computer Aided Design (CAD) software and digitally paint them so that they look realistic. But as much credit goes to the game programmers, who are responsible for turning all of that raw information into an interactive environment, complete with animation, sounds that reflect off of walls, lights that cast shadows, fog that fills the air, and objects that move realistically in response to forces acting on them.

All of those diverse special effects that programmers write into their games (and even the not so fancy ones, like those required to animate the computer screen) have one thing in common: they depend on mathematics. Without math there would be no special effects, no computer games, and, in fact, no computers. Fortunately, even if you were one of those people who would have rather gone to the dentist than do algebra in high school, it is never too late to pick up all of the mathematics you need to know to design cutting-edge computer games. In fact, you will probably even find the mathematics more engaging, since we will be using the concepts for such an enjoyable purpose.

Is more math really going to help you become a better *game* programmer? Absolutely! After finishing this course, you will have mastered many subjects directly related to game development, including,

- *Sets and functions.* You will understand mathematical functions, used everywhere in game mathematics.
- *Trigonometry.* You will use trigonometry to calculate the trajectories of your game's missiles and enemies, rotate 2-D and 3-D points around any place you like, and taunt your feeble-minded enemies with your mastery of a subject that has plagued many an unsuspecting student.
- *Vector math.* You will use the power of vectors to add shadows, realistic movement, collision detection, and much more to your games. Without a doubt, vector math is one of the most powerful and frequently used tools in the arsenal of 3-D game programmers.
- *Matrix math.* You will use matrices to write your own fully interactive, animated 3-D game. Even if you have already used matrices before, chances are you do not know what they are, how they work, or all the cool things you can do with them. In this course, you will get the works -- no skimping on theory here.
- *Analytic geometry.* You will get to use your newfound knowledge of trig, vectors, and matrices to solve very challenging problems in analytic geometry.

If you are a newbie programmer, do not worry -- there are plenty of programs at Game Institute that apply these mathematical concepts to real-world problems, which you can dissect to your heart's content. More advanced programmers may want to write their own programs. For all skill levels, there are many quizzes to test your growing knowledge and interactive programs to enhance your learning.

We will now begin our journey into the powerful, precise world of symbols and theorems: the mathematical side of game development.

1.1 Introduction to Set Theory

I remember as if it were yesterday (perhaps because it *was* indeed yesterday) the scene of my 2-year-old nephew standing atop a six-inch kitchen stepping stool, his hands outstretched toward the countertop, where his tiny fingers held tight, precariously keeping his body in place as he leaned ever backwards -- eyes to the ceiling. He was having a blast, as one could tell from the adorable smile on his chubby little face. I do not know how I did it, but somehow I summoned the inner strength to go over to the little guy and let him know just how dangerous what he was doing actually was. I explained in detail how his small muscles could not bear his weight for long periods of time or for very steep angles, and how when his hands gave out, his head would go plunging straight into the cupboard behind him and start oozing blood. Ouch. "He cannot understand you, John," quipped my younger brother standing nearby. Yeah, sure, I knew that. Oh, the little tyke looked a tad worried, but for the most part, the words and constructions I used went way over his 200-word vocabulary and rudimentary *Subject | Verb | Direct Object* grammatical parser. Well, at least I tried.

The obvious moral of this story is that you need to lift weights so that you can hang off kitchen countertops for as long as you want. But the other, and perhaps more relevant, moral is that complex concepts oftentimes require (or at the very least lend themselves to) a *larger vocabulary* and an *extended set of rules* for using that vocabulary than simple concepts do. Mathematical concepts, many of which

are indeed complex, are no exception, and so much time spent learning mathematics is spent learning new vocabularies and rules for using those vocabularies.

We are going to start this lesson by introducing the vocabulary and related rules for *set theory*, a branch of mathematical logic that has proven itself foundational to just about all other areas of mathematics. We will not venture too deeply into set theory (entire books are devoted to the subject, although that is true of just about every branch of mathematics), but we will go deep enough to build the foundation for an all-important concept in this course and beyond: the concept of *mathematical functions* (which are similar to, but different than, functions in computer programming languages).

First we will cover the vocabulary of set theory, which includes terms like *set*, *subset*, and *element*, and then we will cover the rules -- or, more formally, the *algebra* of set theory. (The term "algebra" is actually a fairly generic word that can be applied to nearly any mathematical entity that you use according to a set of rules). This includes the operations you can perform on sets, like *union*, *intersection*, and *difference*.

1.1.1 The Lingo of Set Theory

What exactly is a set? Surprisingly, this term is not usually defined precisely, since it is difficult to explain what a set is without using some synonym (like *collection* or *group*) that also needs to be defined just as much as *set* does! But nevertheless, we can very loosely define a set as an *unordered collection of unique things*. Each word in this definition is important, so let us examine them in more detail.

By *unordered*, we mean that the *items* in a set have no particular order. In the set of all letters of the English alphabet, for example, 'a' does not come before 'b'. The set itself provides no such information. All it tells you is that both 'a' and 'b' (along with all the other letters of the alphabet) are in the set.

By *unique*, we mean that a set cannot contain two or more copies of the same thing. If you add an additional letter 'a' to the set of all letters just mentioned, for example, then you do not end up with a set, since each item in a set must be unique.

By *things*, we mean just about *anything at all* (it turns out there *are* a few restrictions on what sets can contain, but they are not of concern to us here). A set can contain letters, numbers, words, phrases, symbols, images, abstract concepts like hours and days, real-world objects like toys, cars, and chocolate doughnuts. You name it (practically), and it can be an item in a set.

> **Note:** The version of set theory we are covering here is often referred to as *naïve* or *intuitive* set theory. It was invented by the German mathematician, Cantor, about one hundred years ago. Unfortunately, naïve set theory allows the construction of certain paradoxical sets. For example, a set that contains all sets that do not contain themselves. A natural question to ask is, does this set contain itself? If it does, it should not and if it does not, it should. A version of set theory designed to eliminate these problems is known as *axiomatic* set theory. It is based on a set of axioms designed explicitly to eliminate the construction of paradoxical sets. For our purposes, however, naïve set theory will work just fine.

Figure 1.1: A graphical representation of a set.

A helpful way to think of a set is as a big box that you can fill with anything that you want, provided that you never put the same item in twice and that the items are not ordered. Figure 1.1 illustrates such a box that contains the symbol *x*, an asterisk, the first few digits of π (Pi), a rhinoceros, the German flag, and a Ferrari. These are certainly strange items to have in a set, but the set is perfectly valid -- all items are unique and unordered.

Using pictures is a nice way to visualize the contents of a set, but it is rather bulky and ill-suited to describing abstract sets or those that contain many items. As a result, sets are typically denoted symbolically: the set itself is enclosed by curly braces ('{' and '}'), and commas separate the items in the set. For example, we can denote the set of all integers from 0 to 10 inclusive by the following notation:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Note: The word "inclusive" means the end points of the range are included. In this case, 0 and 10. The word "exclusive", on the other hand, means that the end points of the range are excluded. For example, if we refer to the set of all integers from 3 to 7 exclusive, we mean the integers 4, 5, and 6, excluding 3 and 7.

Remember that a set is unordered, so the above set is exactly the same set as this one:

{ 6, 1, 0, 9, 5, 3, 7, 2, 4, 8, 10 }

No matter how you arrange the elements of this set, it is still the same old set, because sets are unordered.

It is convenient to give sets *names* so that we can talk about them without having to list all of their items. You can name a set by specifying the name, followed by the equals sign ('=') and then the set itself. For example, we might want to call the set that contains an apple and orange *the fruit set*. We could do that with the following notation:

the fruit set = { apple, orange }

Realistically though, giving sets long names (like the one above) is both unnecessary and unwieldy. For this reason, sets are almost always named with a single letter, typically upper-case and italicized, such as *A* or *B*. Of course, there are only 26 letters in the English alphabet, and we will probably want to work with more sets than this during the course of our lifetime, so we will often have to reuse names. At one time *A* might stand for the set of integers, and at another time, the set of *rationals* (numbers in the form *p*/*q*, where both *p* and *q* are integers and *q* is not zero). But do not worry -- the context will always make the meaning clear.

Some sets are too large to describe explicitly. For example, you could never list all integers, since there are infinitely many of them! But we can certainly speak of a set that contains all integers, so how do we

denote such a set? One way to do it is with an ellipsis, a series of three dots (...) used to indicate that something is missing. Using ellipses, we could denote the set of all integers like this:

$$A = \{ ..., -3, -2, -1, 0, 1, 2, 3, ... \}$$

This method works fine for many large sets, but not all. The set of all people, for example, is too large to denote explicitly, yet ellipses are of little help. The set of all real numbers is another good example of a set where ellipsis will just not get the job done.

For these sets and those like them, we need a more powerful notation. One invented for just such a purpose is called *set builder notation*. In set builder notation, sets are not described explicitly, but rather, they are described by listing one or more properties. All things that have these properties are items in the set.

The notation is very similar to what you have just learned, except that instead of listing the *items* of the set inside the curly braces, you list the *properties* of the set. When there is more than one property, you can use a comma to separate them or you can use the word "and" (the comma is preferred for its compactness).

The following example describes the set of all people using set builder notation:

$$P = \{ \text{ the set of all people } \}$$

Another variant of this notation you will see quite frequently involves the use of the symbol '|' and one or more *variables* (symbols that do not have a fixed value). The symbol '|' is read as "such that", and you can think of the variables as placeholders for the things being described.

In this next example, the set of all planets smaller than the earth is described using this variant notation:

$$Q = \{ x \mid x \text{ is a planet and the radius of } x \text{ is less than the radius of Earth } \}$$

You can read this as, "*Q* is the set of all *x* such that *x* is a planet and the radius of *x* is less than the radius of Earth." Since *x* is just a temporary name, a mere placeholder, it does not matter what you call it -- you could have used any symbol, such as '*y*' or '*g*' and the set would still be the same. (This is analogous to variables in computer programming languages: no matter what you call them, the code still works.)

Table 1.1 contains a few more examples to acquaint you with set builder notation.

Table 1.1

Set Builder Notation	Translation
A = { y | y is a human being, y has brown hair }	The set of all people with brown hair.
R = { x | x is a number }	The set of all numbers.
B = { d | d is a day of the week and d is not Thursday }	The set of all days except Thursday.
G = { x | x is a game, x has 3D graphics }	The set of all games that have 3D graphics.
P = { all purple dinosaurs }	The set of all purple dinosaurs.

Some sets are important enough to be given special names, two of which are the *null set*, and the *universal set*. The null set, often represented by the symbol ϕ, contains nothing, hence its name. The universal set is a bit trickier to define. Sometimes it literally contains *everything* that a set can contain. More often, however, it just contains everything we are talking about (for example, if we are talking about integers, and mention the universal set, we probably refer to the set of all integers). The universal set is also sometimes called the *universe of discourse*, and is often denoted by the symbols U or Ω ("omega").

A few other sets that have special names are listed in Table 1.2.

Table 1.2

Symbol	Description
$\boldsymbol{R}$	The set of all real numbers.
$\boldsymbol{Z}$	The set of integers.
$\boldsymbol{N}$	The set of natural numbers (0, 1, 2, …)
$\boldsymbol{Z}^+$	The set of positive integers (1, 2, 3, …).
$\boldsymbol{Z}^-$	The set of negative integers (…, -3, -2, -1).

In the next section, we will talk a little bit more about the *contents* of sets, and introduce some useful notation.

1.1.2 Set Membership

When a thing is *in* a set, we call it an *item* of that set. Equivalently, we may also call it a *member* or an *element* of the set. For example, the character '!' is an element of the set { $, *, z, ! }. Similarly, the letter 'a' is a member of the set { $x \mid x$ is a letter of the alphabet }. Likewise, the computer game *Pong* is an item of the set { $y \mid y$ is a computer game, y was created before the year 1995 }.

Sometimes it is helpful to use a variable to stand for a member of a set. For example, we can define x to be a member of the set of all integers. In this case, x is not necessarily any *specific* integer in the set -- it may be any integer at all.

You can symbolically denote that a thing is a member of a set by listing the thing, the symbol $\in$ (which you can read as, "...is an element of..."), and then the set -- in that order. You can denote that a thing is *not* a member of a set in a similar way, using the symbol $\notin$ instead of $\in$ (which you can read as, "...is not an element of...").

For example,

$x \in \{ x, y, z \}$

$1 \notin \{ 2, 3, 4, ... \}$

Smith $\in$ { ®, Smith, % }

$\pi \notin \{ x \mid x$ is an integer }

Just like items can relate to sets (they can be members or not), sets themselves can relate to other sets. In the next section, I describe some of the common relational operators defined for sets.

1.1.3 Subsets, Supersets, and Equality

Sometimes all items in one set are contained in another set. For example, the set of all integers is contained in the set of all real numbers. When two sets are related in this way, we say the one is a *subset* of the other. If the set A contains the set B, we can express this symbolically by the notation $B \subseteq A$, which you can read as "B is a subset of A." We can also express this same idea by writing $A \supseteq B$, which you can read as "A contains B," or, equivalently, "A is a *superset* of B."

Since any given set contains all of its own members, our definition forces us to say that every set is a subset of itself. Also, our definition requires us to say the null set is a subset of every set. How so? Well, saying that the null set is a subset of every set is the same as saying every set is a superset of the null set; that is, every set contains all the members of the null set. Since the null set contains no members, this is *trivially* true. These conclusions may seem a bit odd, and we could go back and change our definitions to avoid them, but it actually turns out that in combinatorics (a branch of mathematics that deals with counting), as well as other disciplines of mathematics, these conclusions make life simpler.

The subset operator is very similar to the less than or equals operator (<=) for numbers. Every number is less than or equal to itself just like every set is a subset of itself. Also recall that if x is less than or equal to y and y is less than or equal to z then x is less than or equal to z. For example, one is less than or equal to five and five is less than or equal to 9 so 1 is less than or equal to 9. Similarly, if A is a subset of B and B is a subset of C then A is a subset of C.

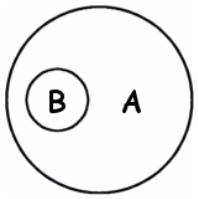


Figure 1.2: A Venn diagram for the relation $B \subseteq A$.

A helpful way of visualizing relationships between sets involves the use of *Venn diagrams*. Venn diagrams, invented in 1881 by the English mathematician John Venn, depict each set as a shape, most typically a circle. The geometric points inside the shape are the members of the set. Figure 1.2 shows such a diagram for the relation $B \subseteq A$ (which, remember, is the same relation as $A \supseteq B$).

When two sets *A* and *B* contain exactly the same items, we say the sets are equal, and denote it *A = B*. You should verify for yourself that if two sets are equal, then they are subsets of each other. This alternate way of defining equality turns out to be very helpful if you are proving that two sets are equal to each other.

Note: You can prove two sets are equal to each other by proving that the first is a subset of the second and then proving the second is a subset of the first. This two-step breakdown makes the problem much easier to deal with.

1.1.4 The Algebra of Set Theory

Just like you can add, subtract, multiply, and divide numbers, there are plenty of operations you can do with sets as well. One operation parallels addition, another subtraction, and a third has no clear parallel in the algebra of real numbers. But before we discuss what these operations are, it may be helpful to review some of the names mathematicians give to algebraic operations that satisfy certain properties.

An operation is said to be *commutative* if you can rearrange the order of the terms without changing the value of the expression. Multiplication and addition of real numbers, for example, are both commutative: $a + b = b + a$, and $ab = ba$. An operation is said to be *associative* if you can add parentheses anywhere you want without changing the value of the expression, for example, $(a + b) - c = a + (b - c)$. Lastly, an operation $\otimes$ is distributive with respect to an operation $\oplus$ if you can write $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$. These symbols may seem a little strange at first but keep in mind that they are just like variables. The difference is that they are not standing for numbers or items in a set, but operators like addition, subtraction and multiplication. In algebra, for example, you can write $a(b + c) = ab + ac$ (the *distributive property*), so multiplication is distributive with respect to addition.

Now we are ready to tackle the main operators in set theory: *complement*, *union*, *intersection*, *difference*, *cardinality*, and *Cartesian product*.

The complement operator allows you to talk about everything that is *not* in a given set; it is similar to the negation operator for real numbers. Formally, a set that contains everything in the universal set that is not in a given set *A* is called the *complement* of *A*, and is denoted by A^C, which you can read as "*A* complement." Some authors also designate the complement of a set *A* by *A′* ("*A* prime") or $\overline{A}$ ("*A* bar").

Figure 1.3: A Venn diagram for the complement operator.

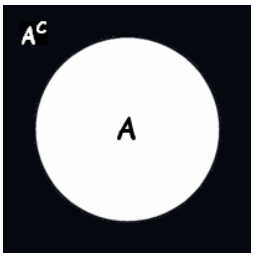
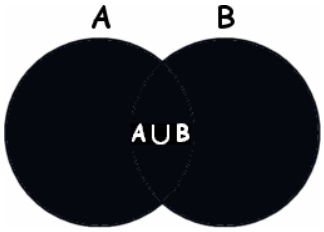


Figure 1.3 shows a Venn diagram for the complement operator. The entire square, including the circle and everything outside of it, is the universe of discourse. The circle in the middle is *A*, and everything outside of *A* is A^C -- this region is shaded black. If the universal set was the set of all integers, and *A* was the set of all even integers, then A^C would be the set of all odd integers.

One notable property of the complement operator is that the complement of a complement of a set is that set itself -- in symbols, $(A^C)^C = A$. So the complement operator is very similar to the word NOT in the English language. For example, if you say that you are not not speaking, that means that you are speaking. It is also similar to the negative sign in algebra: - (-2) is actually +2.

Figure 1.4: A Venn diagram for the union operator.



The union operator is a way of combining sets, and as such, is similar to the addition operator. The union of two sets *A* and *B* is a set that contains every item in *A* and every item in *B*, and nothing else. Symbolically, we write this as $A \cup B$, and pronounce it "*A* union *B*." Some older books write this as $A + B$, which has the advantage of being very suggestive, but the disadvantage of being confused with the addition operator for real numbers.

You can use the union operator on any number of sets: $A \cup B \cup C \cup D$, for example, is a set that contains all items in *A*, *B*, *C*, and *D*, but nothing else. This is true for other operators (intersection and difference) as well.

A Venn diagram for the union operator is shown in Figure 1.4. If *A* represents the set of all integers less than -3, and *B* is the set of all integers greater than -10, then $A \cup B$ is the set of all integers, since all integers fall into one (or both) of those categories. Similarly, if *A* is the set of all humans, and *B* is the set of all gremlins, then $A \cup B$ is the set that contains all humans and gremlins.

The union of a set *A* with the null set is just that set *A* again; in symbols, $A \cup \phi = A$ (this should remind you of the familiar relation in the algebra of real numbers: $a + 0 = a$). The union of a set *A* with the universal set is the universal set: $A \cup \Omega = \Omega$ (this may call to mind $a + \infty = \infty$, although infinity is not a number, so this expression does not have a formal meaning, but you still may have seen it).

Since $A \cup B$ is the same as $B \cup A$, we can see the union operator is commutative. You should verify for yourself that it is also associative.

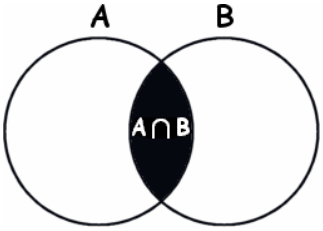


Figure 1.5: A Venn diagram for the intersection operator.

The intersection operator has no nice parallel in the algebra of real numbers. The intersection of sets *A* and *B* is a set that contains every item that is in *both* *A* and *B*, and nothing else. The word "both" is crucial: if an item is in one set but not the other, then it is *not* in the intersection of the two sets. Symbolically, you can write the intersection of two sets *A* and *B* as $A \cap B$ (read as "*A* intersect *B*").

If you have ever used a 3D modeling program like GILES™ that supports constructive solid geometry (CSG), then you probably already have a good grasp of what the intersection operator does, since intersection is one of the operations you can perform on two geometric objects to yield a third object. The third object contains strictly the points in *both* of the two geometric objects from which it is formed.

A Venn diagram for the intersection operator is shown in Figure 1.5. Using a previous example, if *A* represents the set of all integers less than -3, and *B* is the set of all integers greater than -10, then $A \cap B$ is the set { -9, -8, -7, -6, -5, -4 }, since every item in this set is in both *A* and *B*.

As with the union operator, the intersection operator is both commutative and associative. It is also distributive with respect to the union operator -- that is, $A \cap (B \cup C) = A \cap B \cup A \cap C$. You can prove this using the two-step process mentioned earlier (prove each side is a subset of the other) or convince yourself of its truth by drawing a Venn diagram.

> **Note:** You will notice an increasing obsession with proofs as we move through the material. Why bother proofing theorems anyway? Well, when you are deriving equations on your own, you will want to be able to rigorously verify your results before you use them in your games. Otherwise, you may spend hours or days trying to find out what is wrong with an algorithm you wrote when it is really your equations that are at fault. Or you may just spend a long time implementing a set of equations only to later find out they are completely wrong and then end up having to do it all over again. Thus, proofs are quite valuable even for game developers. Proofs also make it easier for you to share your results with other programmers since they will not have to count on your divine inspiration -- they can examine the evidence for themselves for your equations.

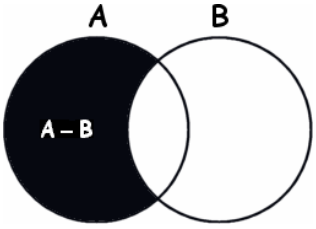


Figure 1.6: A Venn diagram for the difference operator.

The difference operator is similar to the subtraction operator for real numbers. The difference of *A* and *B*, written *A* - *B*, is a set that contains all elements that are in *A* but not in *B*. Figure 1.6 depicts this operator as a Venn diagram.

If *A* denotes the set of all letters of the alphabet, and *B* is the set of all vowels, then *A* - *B* is a set that contains all consonants in the alphabet.

The complement operator can actually be written in terms of the difference operator: $A^C = U - A$, where *U* is the universal set.

The difference operator is associative, but *not* commutative -- *A* - *B* is *not*, in general, the same as *B* - *A*. The only time this will be true is when both sets are equal, in which case *A* - *B* and *B* - *A* are both equal to the null set, and thus are equal to each other. This may be the first time you have seen a non-commutative operator, but as we progress through the course, we will see that there are other non-commutative operators as well. The vector cross-product and the matrix multiplication operators are other examples we will encounter.

The cardinality operator is the simplest of all: the cardinality of a set is simply the number of items in that set. Most mathematicians denote the cardinality of a set *A* as $|A|$ or *card(A)* (the latter is an example of functional notation, which we will see later in this lesson's material -- do not worry too much about it for now). Of course, it does not make sense to talk about the cardinality of infinite sets, so the cardinality operator only applies to finite sets (those that contain a finite number of items).

The last operator we will discuss is the strangest of all: the Cartesian product. It is strange because the result of this operator is a set of *ordered pairs*. An ordered pair is simply something in the form (*x*, *y*). The pairs are referred to as "ordered" because the pair (*x*, *y*) is considered a different pair from (*y*, *x*) (this contrasts with sets -- recall that the sets { *x*, *y* } and { *y*, *x* } refer to the same set).

The Cartesian product of *A* and *B*, written as $A \times B$ ("*A* cross *B*"), is a set of ordered pairs such that the first element in the ordered pair comes from *A* and the second element comes from *B*. Formally, we can write this as:

$$A \times B = \{(x, y) \mid x \in A, y \in B\}$$

This definition can be extended to any number of sets, say *n*, in which case the result is not a set of ordered pairs, but rather, a set of ordered *n*-tuples (an *n*-tuple is essentially just a list of *n* ordered items). The formal definition then becomes,

$$A_1 \times A_2 \times A_3 \cdots \times A_n = \{(x_1, x_2, x_3, ..., x_n) \mid x_1 \in A_1, x_2 \in A_2, x_3 \in A_3, ..., x_n \in A_n\}$$

If you cross a set *A* with itself (which is perfectly valid), you can write either $A \times A$ or A^2. By extension, if you cross a set *A* with itself *n* times, then you can either write a whole list of *A*'s (*n* of them), each pair separated by the Cartesian product operator, or you can simply write A^n. This latter form is preferred for its clarity and conciseness.

The main use of the Cartesian product is geometric: if you cross the real numbers with the real numbers ($\mathbf{R}^2$), for example, then you get a set of ordered pairs (*x*, *y*) such that both *x* and *y* are real numbers. You can think of these ordered pairs as points on a Cartesian plane. We will return to this topic in our next lesson when we talk about how to graph functions. For now, just concentrate on getting familiar with how the Cartesian product operator works.

1.2 Mathematical Functions

There is a joke about a mathematician, a physicist, and an engineer who go vacationing together in Scotland. While on a train, the engineer glances out the window and spots a black sheep on the grassy countryside, commenting, "I did not know Scottish sheep were black." The physicist jumps in to correct him: "No, the most you can say is that *some* Scottish sheep are black." The mathematician rolls his eyes, sighs deeply, and informs them both in a matter-of-fact tone, "No, the most you can say is that there exists at least one sheep in Scotland, at least one half of which is black."

This humorous story illustrates the tendency of mathematicians to be precise. This precision enables them to prove theorems, and in many cases, to simplify the expression of ideas. For example, the

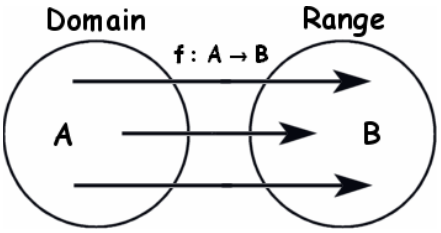
majority of this lesson's material has been focused on laying the exacting foundations of set theory. All the formality may seem a bit much at first -- after all, none of the examples really communicated concepts better than we could communicate them in English. But as we move into the concept of mathematical functions, this groundwork will prove itself to be an immensely simplifying tool.

So what is a mathematical function, anyway, and why should we care what it is? With our set theory groundwork in place, the answer to the first question is surprisingly elegant: a function *f* is a mapping from a set *A* to a set *B* such that with each item in set *A*, the function associates exactly one item in set *B*. This is usually denoted symbolically as $f : A \to B$, which you can read as, "*f* is a function from set *A* to set *B*." The set *A* is referred to as the *domain* of the function, and the set *B*, as the *range*.

> **Note:** The domain is also sometimes called the *pre-image*. Other terms for the range include *co-domain* and *image*.

Though easily stated, this definition may take some getting used to, so we will explore it in more detail next.

Figure 1.7: A function from set *A* to *B*.



One way to think of a function is as a machine: you feed it an element of a set *A*, it will spit out exactly one element of a set *B*. Another way to think of it is as a set of rules: the rules tell you which element of *A* corresponds to a given element of *B*.

A helpful way of visualizing a function is as a series of arrows drawn from one set to another, as shown in Figure 1.7. Under this way of looking at a function, we might say that *f sends* each element of *A* to exactly one element in *B*. Suppose set *A* is our domain and set *B* is our range. Then another way to think of functions is to think of all the items in set *A* as baseballs. Each baseball has a label on it that tells you what item it represents. Off in the distance is a row of boxes. Each box represents a different item in set *B*. When you want to know what box is associated with a given baseball, you just bring that baseball to the pitcher who then lobs it into one of the boxes. Under this view, the pitcher is the function: he sends each item in set *A* to exactly one item in set *B*.

Suppose we have a set *A* = { 1, 2, 3 }, and a set *B* = { Mary, Larry }. We can define a function $f : A \to B$ in many ways. One way is to say that *f* sends 1 and 2 to Mary, and 3 to Larry. A graphical representation for this function is shown below:

$1 \to$ Mary

$2 \to$ Mary

$3 \to$ Larry

Suppose we want to create a function *g* from the real numbers to the real numbers. There are many (in fact infinite many!) ways we can do this, but let us suppose that *g* takes the real number that we feed it, squares it, and outputs the result. Thus *g* sends 1 to 1, 2 to 4, 3 to 9, and so on.

For sake of brevity, we often say that an item *a* in the range is sent to *f(a)* ("*f* of *a*") by the function *f*. You can regard *a* as the input to the function *f*, and *f(a)* as the output. If you are familiar with a programming language such as C or C++, this notation should be immediately familiar. In fact, you can construct mathematical functions in these languages using almost the same notation. For example, the function *g* we defined in the last paragraph can be expressed in C/C++ with the following code:

```
float g ( float x ){
    return x*x;
}
```

You invoke this function by writing *g(x)* in your code somewhere. The function takes the parameter *x*, squares it, and returns the result. Hence there is a strong parallel between mathematical functions and functions in computer programming languages (the former is almost certainly the source of the latter -- most of the early pioneers in computer science were mathematicians).

There is a property of functions implicit in our definition that might not be obvious: a function must send each item in its domain to *exactly and only* one item in its range -- it can never send it to two or more items. That is, if *f(a) = b*, and *f(a) = c*, then *b = c*. There are no exceptions. If you have some sort of mapping from one set to another that does not obey this rule, then it is not a function. We will explore a graphical, intuitive way of viewing this property of functions in the next lesson's material. Another property that might not be obvious is that every single item in the domain *must* be sent to the range. You cannot have items sitting in the domain that are not sent anywhere. Mappings like this are not functions at all. This is not true for the range, though. You can have items in the range that are not associated with any item in the domain.

Be careful to notice that these restrictions on functions do not force you to send different items in the domain to different items in the range. There is nothing illegal about sending any number of different items in the domain to *one* item in the range (in fact several of the examples of functions listed above do just that). It is perfectly okay, for example, if a function *f* sends *a* to *c*, and *b* to *c*, where *a* and *b* are distinct items in the domain, and *c* is an item of the range.

There is however, a special name for functions that send each item in the domain to its own *unique* item in the range, and that contain no more items in the range than in the domain. These functions are called *invertible*, since you can construct a function, called the *inverse* of the original function, which sends each item in the range *back* to its corresponding item in the domain. You cannot construct such a function if the original function sends two or more *distinct* items in the domain to the *same* item in the range, since otherwise, the inverse function would have to send that item in the range to the two or more items in the domain, which would violate the definition of a function. Nor can you construct such a function if the range contains more items than the domain, since that would imply either the inverse function would have to send one item in the range back to two or more items in the domain, or that the inverse function would not send one or more items in the range to anything at all in the domain.

> **Note:** There are special names given to other kinds of functions as well. If a function sends each item in the domain to exactly one item in the range then the function is called *one-to-one*. This is true even if the range contains more items than the domain. A function is called *onto* if each item in the range is associated with one or more items in the domain. You should verify that if a function is both one-to-one and onto, it is also invertible.

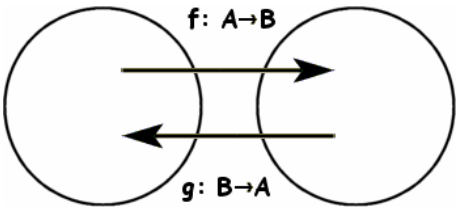


Figure 1.8: A function and its inverse.

Figure 1.8 depicts a function and its inverse graphically. The function sends items from *A* to *B*, and the inverse, from *B* to *A*. The usual notation for the inverse of a function *f* is f^{-1} (read "*f* inverse"). You should not confuse this with raising a number to the -1 power, or, as we deal with algebraic functions later on, with the reciprocal of an algebraic expression.

A consequence of the definition of an inverse is that $f^{-1}(f(x))=x$. In English: The function *f* sends *x* to some element in *B*, say *y*. The inverse function takes this element *y* in *B* and sends it back to *x* in *A*. Thus the inverse function *undoes* the operation of the function.

As an example, consider a function *f* from the real numbers to the real numbers, such that $f(x)=2x$. This function takes the input, multiplies it by two, and then outputs the result. The function turns out to be invertible (as we will see in the next lesson), and its inverse function is $f^{-1}(x)=\frac{1}{2}x$. Evaluating $f^{-1}(f(x))$, we get back *x* again, as expected. You should play around with this function to get a feel for how inverse functions behave.

Conclusion

So what is the big deal with functions? In fact, why should we even care about them as game developers? Alas, a partial answer to this question will have to wait for our next lesson, when we delve into a wide range of mathematical functions that have numerous uses in game development. In the next lesson we will veer away from the general-purpose definition of functions and focus our attention on algebraic functions. We will explore how some of our definitions in this chapter carry over to algebra and cover a few examples of algebraic functions. Lastly, we will cover a fantastic way of visualizing these functions, one which turns out to have quite a few applications in game development. Until then, please make sure that you try your hand at the exercises. When learning mathematics, there is nothing more important than practice.

Exercises

1. Determine which of the following are sets:

   a. $F = \{ p \mid p \text{ is a prime} \}$

   b. $M = (1, 2, 3, 4, \ldots)$

   c. $R = \{\}$

   *d. $Q = \{ \{x, y\} \mid x \text{ is an integer}, y \text{ is an integer}, x > y \}$

2. Translate the following sets from English into set builder notation:

   a. A is the set that contains the names of all electronic components.

   b. L is a set that contains all integers that are not less than the square root of five.

   c. W is a set that contains all other sets.

   d. Z is a set that contains everything that is not contained in the universal set.

3. Translate the following sets from set builder notation to English:

   a. $G = \{ y \mid y \in \mathbf{R}, y > 0 \}$

   b. $D = \{ s \mid s \text{ is a string of letters} \}$

   c. $P_S = \{ x \mid x \subset S \}$

   d. $N = \{ x \in \mathbf{Z}^+, x \in \mathbf{Z}^- \}$

4. Prove that if A and B are sets, and $A = B$, then A is a subset of B, and B is a subset of A.

5. *Prove that the intersection operator is distributive with respect to the union operator, or draw a Venn diagram to illustrate this.

6. Calculate the Cartesian products of the following sets (list the ordered pairs explicitly if the resulting set is finite):

a. $A = \{ 1, 2, 3 \}$, $B = \{ a, b, c \}$

b. $A = \{ !, @, \# \}$, $B = \{$ alpha, beta, gamma, delta $\}$

c. $A = \{ x \mid x$ is a real number $\}$, $B = \{ y \mid y$ is a letter of the alphabet $\}$

d. $A = \mathbf{R}^2 \times \mathbf{Z}$

7. Determine whether the following mappings are functions:

a. $1 \to 2$

$3 \to 4$

$5 \to 2$

$4 \to 1$

b. $3 \to 2$

$1 \to 1$

$5 \to 2$

$1 \to 3$

8. Determine which of the following functions are invertible:

a. $f(1) \to 2$

$f(3) \to 4$

$f(4) \to 5$

$f(2) \to 3$

b. $f(5) \to 2$

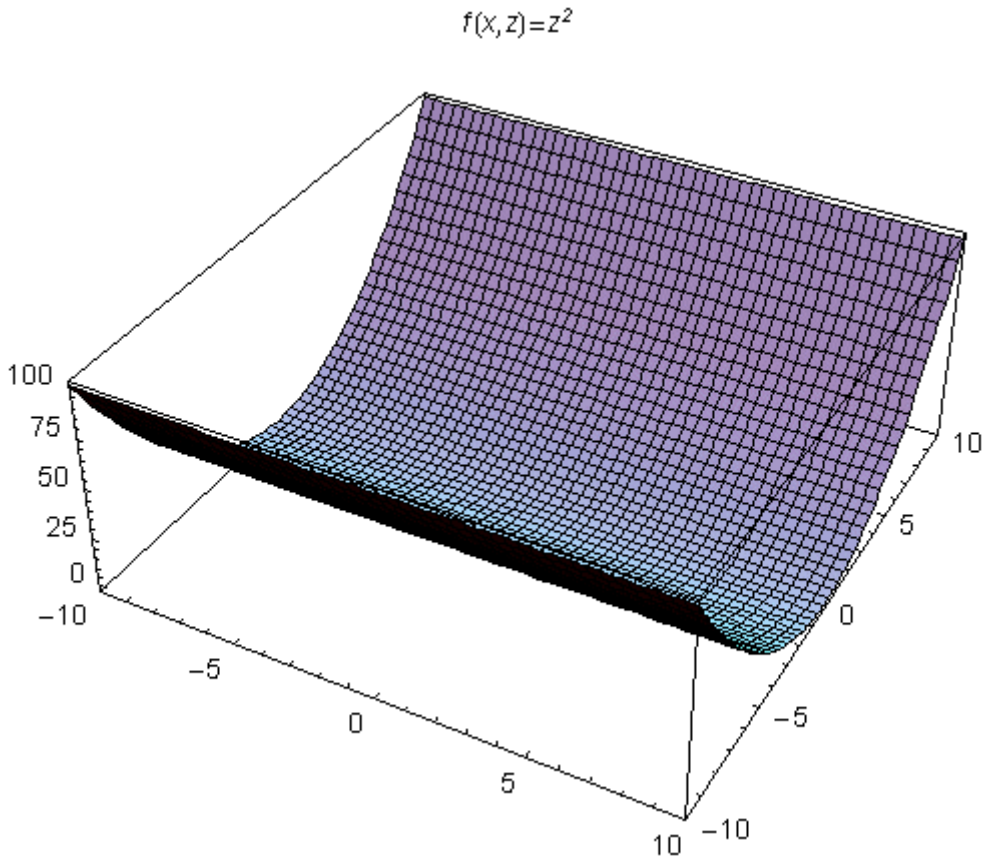
$f(1) \to 1$

$f(6) \to 2$

$f(2) \to 3$

Chapter Two

Mathematical Functions



Introduction

In our last lesson we painted a rather broad picture of functions using set theory as our base. A function, we said, is a mapping from a set *A* (the domain) to a set *B* (the range) such that with every element in *A*, the function associates exactly one item in set *B*. As this definition suggests, the concept of a function is extremely broad, since there are not any restrictions on what kinds of sets you can use for the domain and range. For this reason, the vast majority of possible functions have no applications in computer science or game development. So in this lesson, we will narrow our attention to *mathematical functions*, and attempt to get a feel for what they look like and how they work. We will also learn how to visualize these functions and cover a few of the more common ones used extensively in game development.

2.1 Mathematical Functions

Mathematical functions are functions *from* mathematical sets *to* mathematical sets. We are already very familiar with the sets of real numbers, positive numbers, rational numbers, and integers, but there are other sets as well, such as sets of ordered *n*-tuples (which we briefly mentioned in the last chapter), sets of matrices, sets of vectors, sets of quaternions, sets of imaginary numbers, and much more. Mathematical functions map from one of these sets to another (or possibly the same set).

Perhaps the most common type of function we will encounter is one that maps from some subset of the real numbers *to* another subset of the real numbers. (Keep in mind that the set of all real numbers, just like any other set, is a subset of itself, so the domain and range of these functions may very well be the set of *all* real numbers.) These kinds of functions are sometimes called *real-valued functions of a real value*. The "real-valued" part indicates the range of the functions is a subset of the real numbers. The "of a real value" part indicates the domain is a subset of the real numbers.

We saw a few examples of real-valued functions of a real value in the last lesson, such as $f(x) = x^2$. This particular function takes any real number, squares it, and outputs the result -- itself a real number.

There are infinitely many ways we can create functions like this one using just high school algebra. For example, all of the following are perfectly valid functions that map from real numbers to real numbers:

$$f(x) = 3x^4 - 2x + 9$$
$$f(x) = x$$

Of course, we are not limited to using the symbol "*f*" for the function and "*x*" for the element in the domain (although these are quite common choices). All of these examples are equally valid as well:

$$g(q) = q^4 - 3$$
$$\Phi(\theta) = \frac{1}{5}\theta^2$$

Suppose we do not want to pass just a single variable to a function, but want to pass a whole bunch. How do we fit this into the definition of a function? Simple: we just map from the set of ordered *n*-tuples (where *n* is the number of variables we want to pass to the function) to whatever set we are interested in. Here are a few examples that map from the set of ordered 3-tuples to the set of real numbers:

$$f(x,y,z) = x + y + z$$
$$d(x,y,z) = \sqrt{x^2 + y^2 + z^2}$$

As you can see, the definition of a function gives us quite a bit of flexibility. The only rule we need to satisfy is that our function associates exactly one item in the range with each item in the domain.

Are there any algebraic constructs that do not satisfy this constraint? Absolutely. It is quite possible to accidentally create an equation that we *want* to map from the set of all real numbers to the set of all real numbers, but which fail to do so for specific choices of elements in the domain (see Figure 2.1).

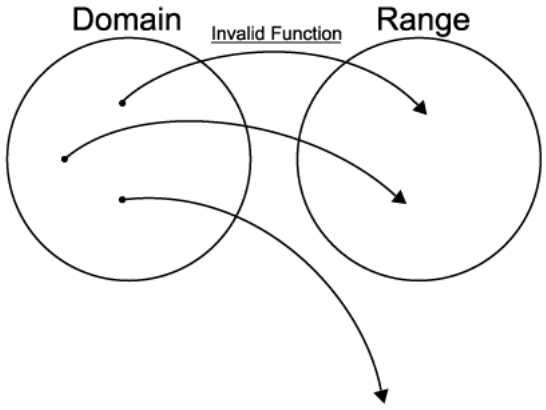


Figure 2.1: A mapping that fails to satisfy the criteria for a function.

Consider, for example, the following "function":

$$f(x) = \frac{1}{x}$$

Now if we specify the domain and range for this "function" as the set of all real numbers, then our mapping is not a function at all. Why is this so? Well, 0 is a real number, so if the equation truly did specify a function from the set of all real numbers to the set of all real numbers, then $f(0)$ would also have to be a real number, but $f(0)$ is just $1/0$, which is undefined.

However, if we restrict our domain to the set of all real numbers *except* 0, then *f* instantly becomes a function. Another way to make *f* a function is to define it *piecewise* -- that is, define *f* using more than one equation, and specify for what elements in the domain each equation applies. We could, for

example, define $f(x)$ to be $\frac{1}{x}$, except when $x = 0$, and then specify that for this sole case, $f(x) = 1$. This "composite" function is called a *piecewise-defined function*. Piecewise-defined functions are designated with the following notation:

$$f(x) = \begin{cases} \frac{1}{x}, & x \neq 0 \\ 1, & \textit{otherwise} \end{cases}$$

Division by zero is not the only way to create a mapping that is not a function. Suppose we define a mapping $f(x) = \sqrt{x}$ from the set of all real numbers to the set of all real numbers. Then f is not a function because if you evaluate it for any negative number, the result is not a real number. Take, for example, $f(-2) = \sqrt{-2}$. There is no real number such that, when multiplied by itself, is equal to -2, since whether the number is negative or positive, when you multiply it by itself the result will be positive.

> **Note:** Recall that the product of two negative numbers is a positive number, and of course, the product of two positive numbers is a positive number. So the square of any number is positive. That is, until we get to chapter 10!

2.2 Functions and Graphs

I can remember overhearing a teacher carefully explain the concept of a tensor to one of his more inquisitive students: "Essentially," he said, "a second-order tensor is a list of 9 numbers that satisfy certain properties." The student listened patiently while the professor went on to describe what those properties were. "Make sense now?" the professor asked. "Yeah, only what does a tensor *look* like?"

Unfortunately, the student was not to have his wish fulfilled, since you cannot really picture a tensor. But the point is that as human beings, we are very comfortable with tangible things; things we can hear, touch, taste, and see. When it comes to the intangible, we often try to find ways to understand these things through analogies or representations. In the last chapter's material, for example, we explored the use of Venn diagrams to picture relationships between sets, elements, and other sets, and used pictures to represent functions. These illustrations give a sense of concreteness to otherwise abstract subjects.

Not surprisingly, mathematical functions are no exception. There is a way to visualize all real-valued functions of real values. This technique is called *graphing*, and it is the topic of the next section.

2.2.1 Visualizing Single-Variable Functions with Graphs

The formal definition of a *graph* is quite concise: a *graph* of a one-variable function $f : A \to B$ (where *A* and *B* are both subsets of the real numbers) is a set *G* of ordered pairs such that $G = \{(x, f(x)) \mid x \in A\}$. How does this rather terse definition help people understand functions better? Well the great thing about a graph is not its cryptic definition, which serves only as setup, but rather what you can do with it -- you can draw a picture of a graph that visually represents the function.

In order to understand how this is done, we must first cover the concept of a Cartesian coordinate system (which you may already be familiar with from your high school days). Essentially, a Cartesian coordinate system is a mechanism that allows you to specify the location of points using lists of numbers.

Suppose you were told to draw a point on a piece of paper. This can be done with two pieces of information: first, you are going to be given some measurements relative to a fixed point on the paper, say the exact center; this point is called the *origin* of the *coordinate system*. Then you will be told to place your pencil at the origin, and move it left or right, then up or down a certain number of inches. I can describe this movement with two numbers: one that tells you how far you should move left or right (say positive numbers are right of the origin, and negative numbers are left), and another that tells you how far you should move up or down (say with positive numbers being up, and negative numbers being down). Together, these two numbers are called the *coordinate* of the point. There are unique coordinates for every point on the piece of paper.

Figure 2.2 is an illustration of how this setup might work.

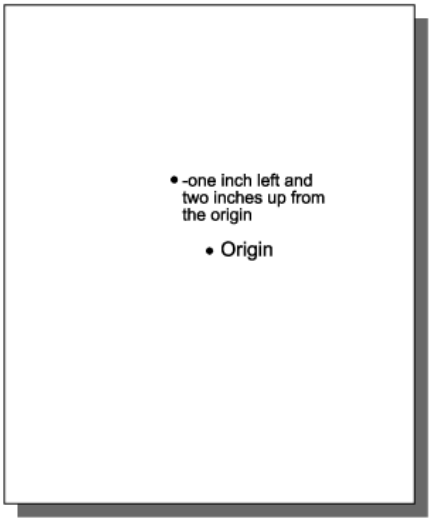


Figure 2.2: A coordinate system on a piece of paper.

What we have just described is a very simple two-dimensional Cartesian coordinate system. Mathematicians add some formality to the preceding illustration by introducing the concept of an *axis*, an infinitely long imaginary ruler of sorts. For the two-dimensional Cartesian coordinate system, there

are two axes: the horizontal one, often referred to as the *x-axis*, and the vertical one, usually called the *y-axis*. The origin is usually taken to be the place where these two axes intersect.

You can describe all points in the Cartesian coordinate system by using an ordered pair. The first number in the ordered pair tells you how far along to go on the *x*-axis, and by convention, positive numbers are taken to be rightward, and negative numbers, leftward. The second number in the ordered pair tells you how far to go on the *y*-axis -- most often, positive numbers are upward and negative numbers are downward.

Figure 2.3 shows how of all this looks with a visual representation of the *x*- and *y*-axes and a point plotted in the coordinate system. Notice how finite lines represent the axes, since it is impossible to draw an infinitely long line in a finite space. But you should still think of the axes as going on forever. Also, little marks are drawn on the axes and numbers are drawn next to the marks: the numbers indicate how far along the axes the marks are; this lets you know which numbers correspond to which positions without having to use real-world measurements such as inches or centimeters. Lastly, the direction of positive movement is indicated with a plus sign -- here the standard conventions are chosen, so there is a plus sign to the right of the origin for the *x*-axis, and a plus sign to the top of the origin for the *y*-axis.

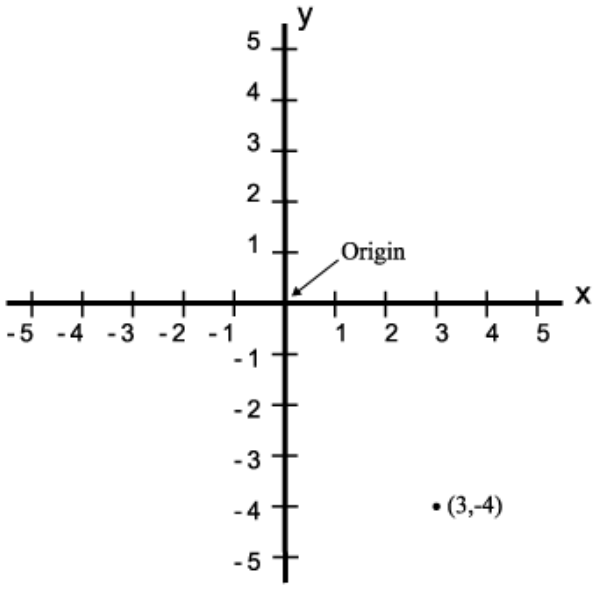


Figure 2.3: A point on a two-dimensional Cartesian coordinate system.

Recall we said that a *graph* of a one-variable function function $f : A \to B$ is a set *G* of ordered pairs such that $G = \{(x, f(x)) \mid x \in A\}$. How do you visualize a function given its graph *G*? Well *G* is a set of ordered pairs, and we just learned that an ordered pair can represent a point, so the answer is staring us right in the face: simply draw (or *plot*) all the points in *G* on some coordinate system. The process of drawing all these points is called *graphing* or *plotting* the function, and as we shall soon see, it makes visualizing functions very easy.

You may be thinking to yourself that for most functions, *G* will contain infinitely many points. After all, we can pass infinitely many real numbers to most functions, and the "number" of points is equal to the "number" of things we can pass to the function. Indeed this is so, which creates a bit of a problem: how

can we display infinitely many points? The answer is that we cannot, due to both time and physical constraints. It would take infinitely long for us (or even a fast computer) to display infinitely many points. Even if we could do it in a finite amount of time, the best we could hope for is to use one subatomic particle to represent each point, and though the universe contains a great many subatomic particles, the actual number is finite.

So can we ever truly graph a function, then? Not in the general case. But what we can do is focus on a particular subset of the function's domain (include only a specific range of real numbers, say the real numbers from -10 to 10) and then from this subset, choose a finite number of points to draw. We can then play "connect the dots" and draw lines between the points we have plotted. Assuming the function is "smooth" (a concept given more precision in calculus), and assuming we choose enough points, our graph of the function should approximate what the function really looks like.

Now that we have defined the concept of a graph, and discussed how to display it (or display some wisely chosen part of that graph), it is time for a few examples. Figure 2.4 shows the graph of the function $f(x) = x^2$ for all *x* in the interval [-1, 1].

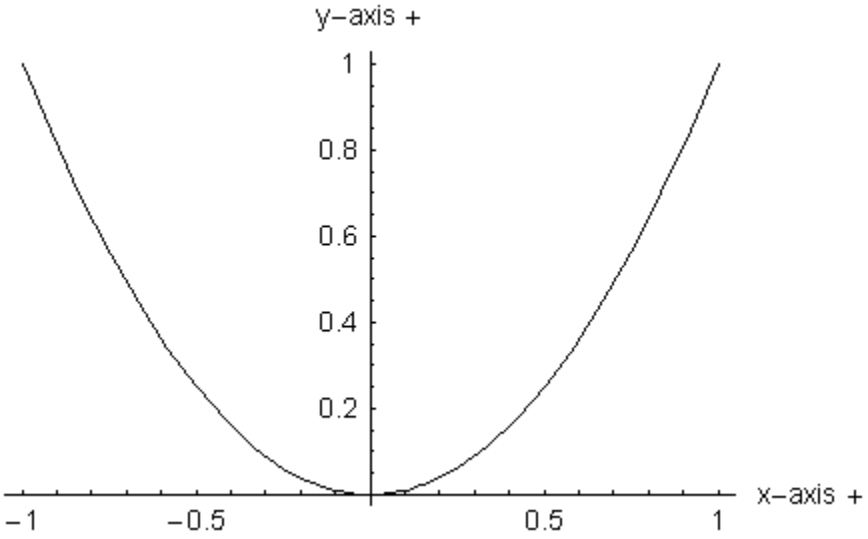


Figure 2.4 The graph of the function $f(x) = x^2$

A computer program called Mathematica™ was used to generate this graph. This is not a numerical analysis course, so we will not be delving into exactly how the program works its magic. But suffice it to say that Mathematica™ is one of the best math programs around, and it does an excellent job at graphing functions (as well as just about everything else).

How should you interpret the graph? Well if you see some point plotted, say (x, y), then you know that this point is in the graph, which means $y = f(x)$. For example, in Figure 2.4 you can discern that the point (1, 1) is plotted, which means that $1 = f(1)$. Let us check to see if this is true. Recall that $f(x) = x^2$, so $f(1) = 1^2 = 1$. So (1, 1) really *is* in the graph of the function. Before proceeding you should take a few minutes to study Figure 2.4. It is even recommended that you graph the function yourself: just choose some *x*'s, evaluate the function for these values, and plot the resulting ordered pairs

on a piece of graph paper. Then connect the dots and frame your masterpiece in the living room for all to behold!

Just to give you a feel for how some other functions look, a more complex function is graphed in Figure 2.5. Once you have become more familiar with how graphs correspond to functions, you will be ready to proceed to the next section, in which we cover three-dimensional graphs.

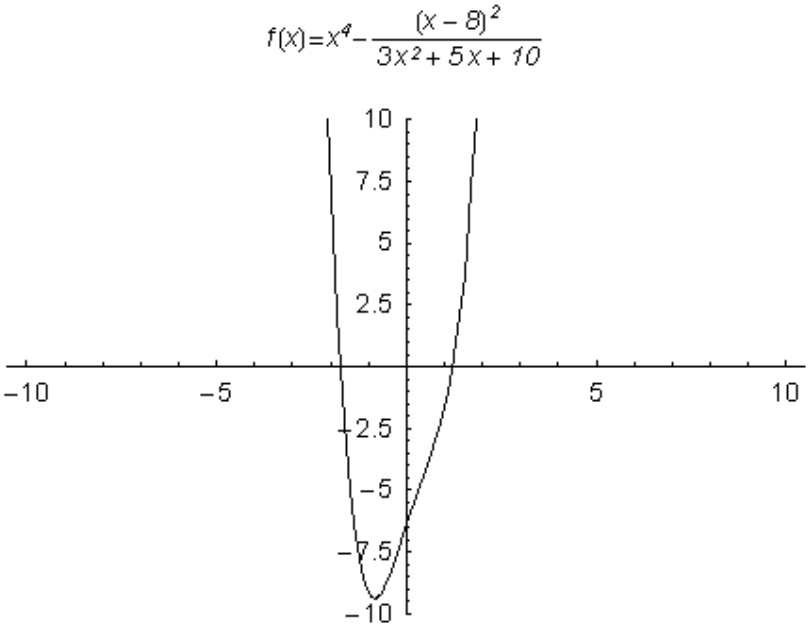


Figure 2.5 The graph of a more complex mathematical function.

2.2.2 Visualizing Two-Variable Functions with Graphs

Just as we can visualize a real-valued function of one real variable with a two-dimensional image, we can visualize a real-valued function of *two* real variables with a *three-dimensional* image. Actually, the image is not 3D in the strict sense of the word; rather it is a 2D image with the illusion of depth (very similar to a photograph of a 3D object).

To do this, we still make use of the concept of a graph, but we extend it as follows: the *graph* of a two-variable function $f : A \to B$ (where A is a set of ordered pairs, and B is a subset of the real numbers) is a set of ordered triples G such that $G = \{(x, f(x,z), z) \mid (x,z) \in A\}$.

As with the one-variable case, to visualize the graph, all we have to do is plot some subset of the elements in G. Note though, that now the elements in G have three components (G is, after all, a set of ordered *triples*), so they actually correspond to three-dimensional points. This means that to plot them, we need a three-dimensional coordinate system.

It is easy enough to cook up such a system. All we have to do is add an additional axis, which we will call the *z-axis*, and make this axis perpendicular to the two we have already covered. Figure 2.6 shows an example of such a coordinate system. The *x*-axis measures how far to the left or right a point is, the *y*-axis measures how far up or down the point is, and the *z*-axis measures the *depth* of the point. By common convention, positive numbers indicate in front of the origin, and negative numbers indicate behind the origin (the side of the origin you are on when you look at Figure 2.6).

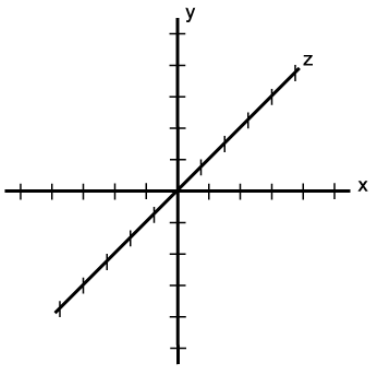


Figure 2.6: A three-dimensional Cartesian coordinate system.

If you were told to plot the point at (1, 3, -2), you could do so by placing your pencil at the origin, then moving 1 unit to the right, 3 units up, and 2 units backward, towards yourself. To display a graph, you have to make use of the same tricks we did when plotting the graphs of one-variable functions: only look at some subset of the points in the graph, and connect those points to form a three-dimensional surface.

That is enough background to introduce our first three-dimensional graph: a plot of the two-variable function $f(x) = x^2 + y^2$ (which actually looks quite similar to the one-variable function $f(x) = x^2$, only in three-dimensions). You can see the Mathematica™-generated plot of this function in Figure 2.7.

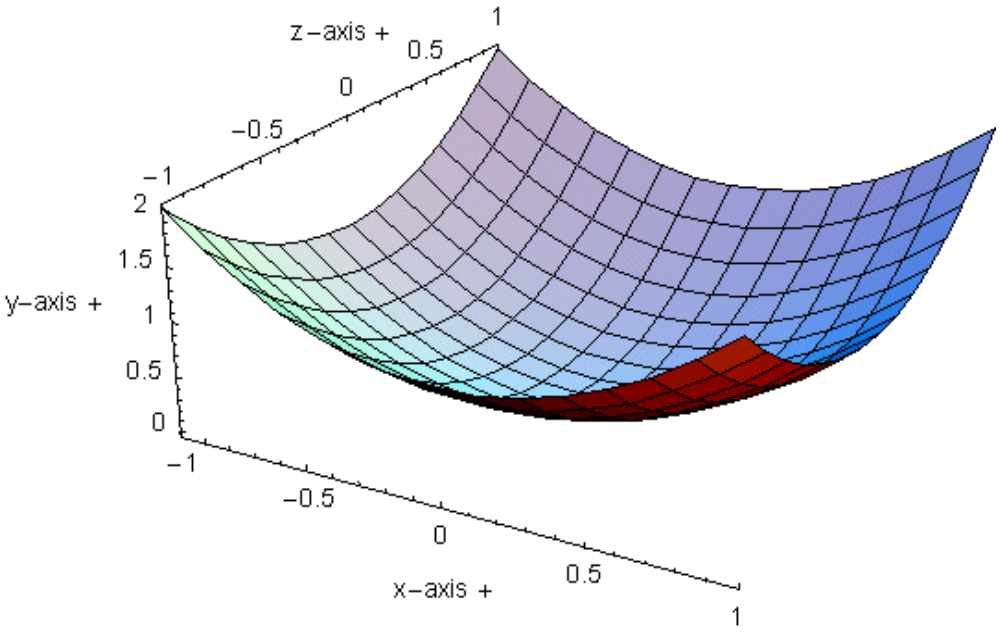


Figure 2.7: A plot of the two-variable function $f(x) = x^2 + y^2$.

The interpretation of the graph is much like that for one-variable functions. If you see some point (*x*, *y*, *z*) plotted, then you know that $y = f(x,z)$. For example, the plot shows that the point (-1, 2, 1) is in the graph, so we would expect that $f(-1,1) = 2$. Sure enough, $f(-1,1) = (-1)^2 + 1^2 = 1+1 = 2$.

Figure 2.8 shows some other real-valued functions. You should spend some time relating their equations to how their graphs look.

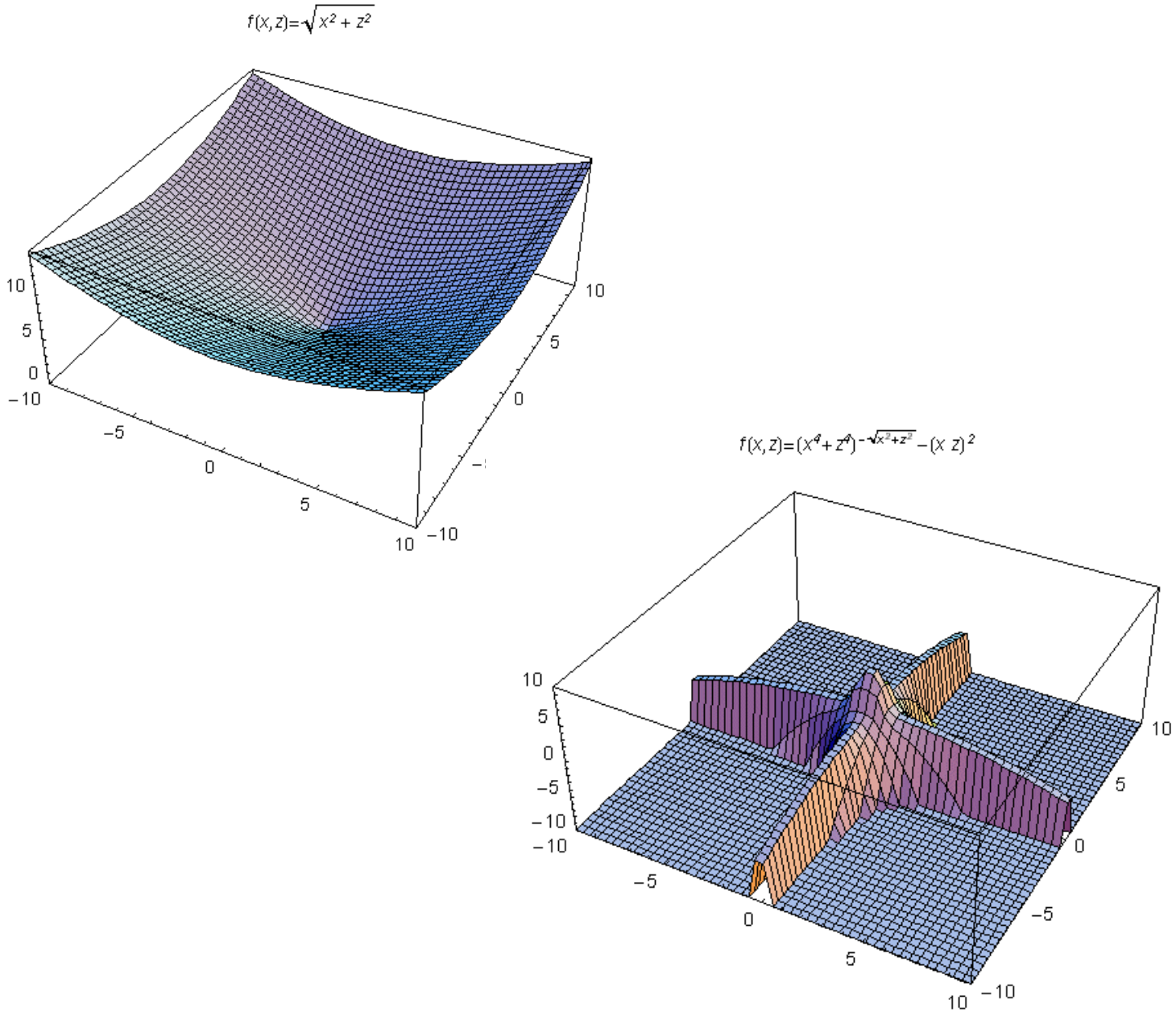


Figure 2.8: Real-valued functions of ordered pairs

Humans live in a world of three spatial dimensions, so they have no problems visualizing two-dimensional and three-dimensional graphs. Unfortunately, we have no capacity to visualize higher dimensions. So if you run across a function of three or more variables, you are essentially out of luck. There are ways to visualize these functions, but only by breaking them down into one- or two-variable functions by holding some of the other variables constant (We will look at this technique later on in this lesson's material). The good news is that, as game programmers, we are mostly going to be concerned with functions of one- or two-variables.

Now that you have a better grasp of what functions are, and can visualize them with the aid of graphs, it is time to start covering some of the ones that come up all the time in game development.

2.3 Families of Functions

With the rich variety of mathematical functions, you might wonder if any attempt to classify them could possibly succeed. As fate would have it, however, certain kinds of functions come up quite frequently in math and computer science -- often enough to group them into categories. For example, mathematicians have defined *polynomial functions*, *rational functions*, *exponential functions*, *logarithmic functions*, *trigonometric functions*, and other classes of functions as well.

These functions are important enough to game development that they warrant some study. Exponential functions, for example, are used to model fog, as well as certain kinds of light sources. Rational functions are used indirectly in texture mapping. Polynomial functions are used to approximate other, more complicated functions, and to create smooth surfaces and fluid paths. Trigonometric functions are used a million different ways in everything from animating 3D geometry to collision detection.

In the remainder of this lesson's material, we are going to look at several of the easiest types of functions to understand: the absolute value function, exponential functions, and logarithmic functions.

2.3.1 Absolute Value Function

The absolute value function is a strange little function: you feed it a number, and if that number is positive, it will return that number; if the number is negative, it will return the negative of that number (which is itself a positive number).

The absolute value function is designated by the symbols '|' and '|', which -- in departure from the normal functional notation -- enclose the expression being sent to the function. For example, if you want to write down the absolute value of 3, you write |3|. Similarly, the absolute value of -8 is |-8|.

You can express the absolute value function using mathematical operators as follows:

$$|x| = \sqrt[+]{x^2}$$

In words, this says that the absolute value of *x* is the positive square root of the square of *x*. Since a negative number times a negative number is a positive number, and a positive number times a positive number is a positive number, the square of *x* is always positive. The square root of this positive number will itself be positive, so you can see the absolute value function returns positive numbers unchanged and inverts the sign of negative numbers.

2.3.2 Exponential Functions

Perhaps you have heard the word "exponential" in a sentence before, such as "the value of Amazon.com stock rose exponentially in its early years," or "it is exponentially more difficult to find a parking spot downtown than it is on the fringes of the city." These sentences should illustrate that the word "exponential" is often associated with "very much" or "greatly". Does this have any bearing to the mathematical meaning of "exponential"? The answer is a definite "yes".

Mathematicians are often concerned with how quickly functions increase or decrease. To understand what this means, think back to the graphs of the functions you have seen so far. It is not too difficult to picture the wavy line in such graphs as the silhouette of a mountain range. Wherever the slope of the range is very steep, the function is increasing or decreasing rapidly (increasing if, when walking on it from left to right, you have to walk uphill, and decreasing otherwise).

Some functions do not increase or decrease at all. Take, for example, the function $f(x)=1$. When you plot this function, as we have in Figure 2.9, you can see that it is just a straight horizontal line, lifted one unit above the origin.

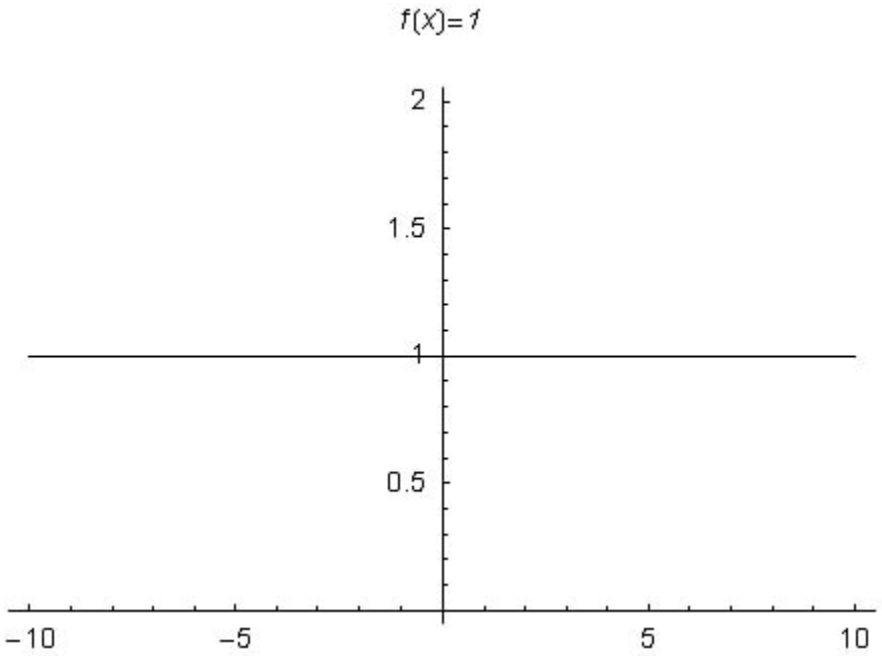


Figure 2.9 The plot of the function $f(x)=1$.

Other functions change comparatively slowly. For example, the plot of the function $f(x) = 2x - 3$ is just a straight line, as shown in Figure 2.10. Sure, the function increases, but not by much -- to return to our mountain range analogy, for every one unit you walk to the right on the range, you have to hike "just" two units up if you want to stay on the plot of the function (that may seem like a lot now, but just wait!).

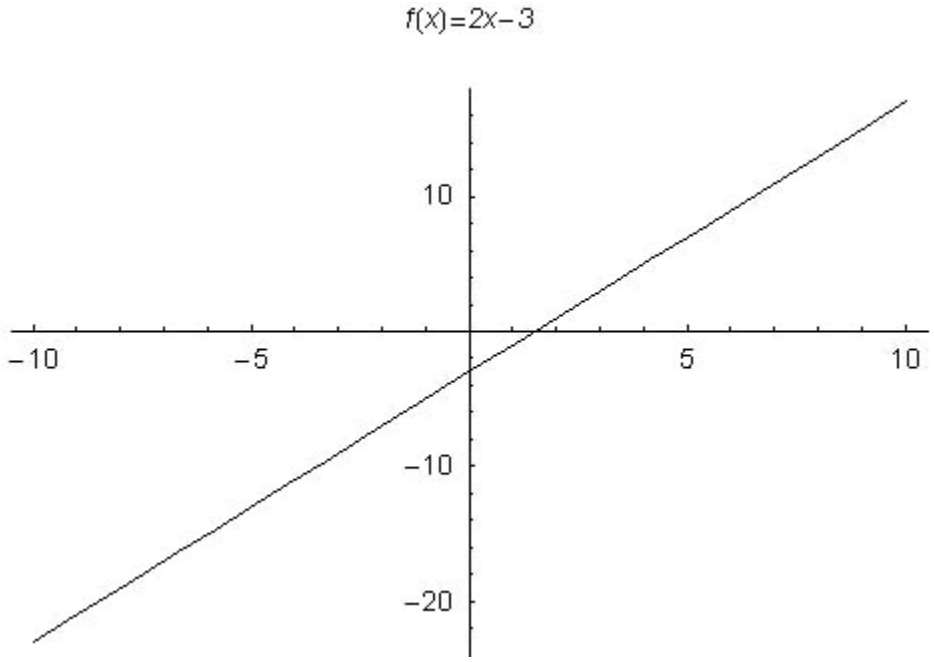


Figure 2.10 :The plot of the function $f(x) = 2x - 3$.

The plot of the function $f(x) = x^2$ (the first function we graphed, visible in Figure 2.4) decreases and increases quite a bit more rapidly than any of the previous ones, especially the farther away from the origin you get. For example, $f(10) = 100$ and $f(11) = 121$. So if you were to walk on the plot of the function from x = 10 to x = 11, you would have to hike 21 units vertically to stay on the plot. And the situation only gets worse: $f(100) = 10{,}000$, and $f(101) = 10{,}201$, so if you walked from x = 100 to x = 101, you'd have to hike a full 201 units straight up to stay on the graph.

Back in Figure 2.4 we saw the plot of x^2. You might be thinking to yourself that other *higher* powers of x (such as x^3, x^4, or x^5) would change even more rapidly than x^2. This is quite true: the greater the power, the faster the function changes (increases or decreases). But there is another class of functions, called *exponential* functions, which make even large powers of x seem like straight lines in comparison!

An exponential function is one in the form:

$$f(x) = a^x \qquad a > 0, \qquad a \neq 1$$

The constant a, called the *base* of the exponential function, is just any real number, subject to the following constraints: $a > 0$, since otherwise, the function would not map to the real numbers; and $a \neq 1$, or else the function would just graph as a horizontal line (which does not fit with the meaning of the word *exponential*).

Does the exponential function live up to our expectations? Take a look at the graphs of the exponential function for a few different base choices in Figure 2.11 and judge for yourself.

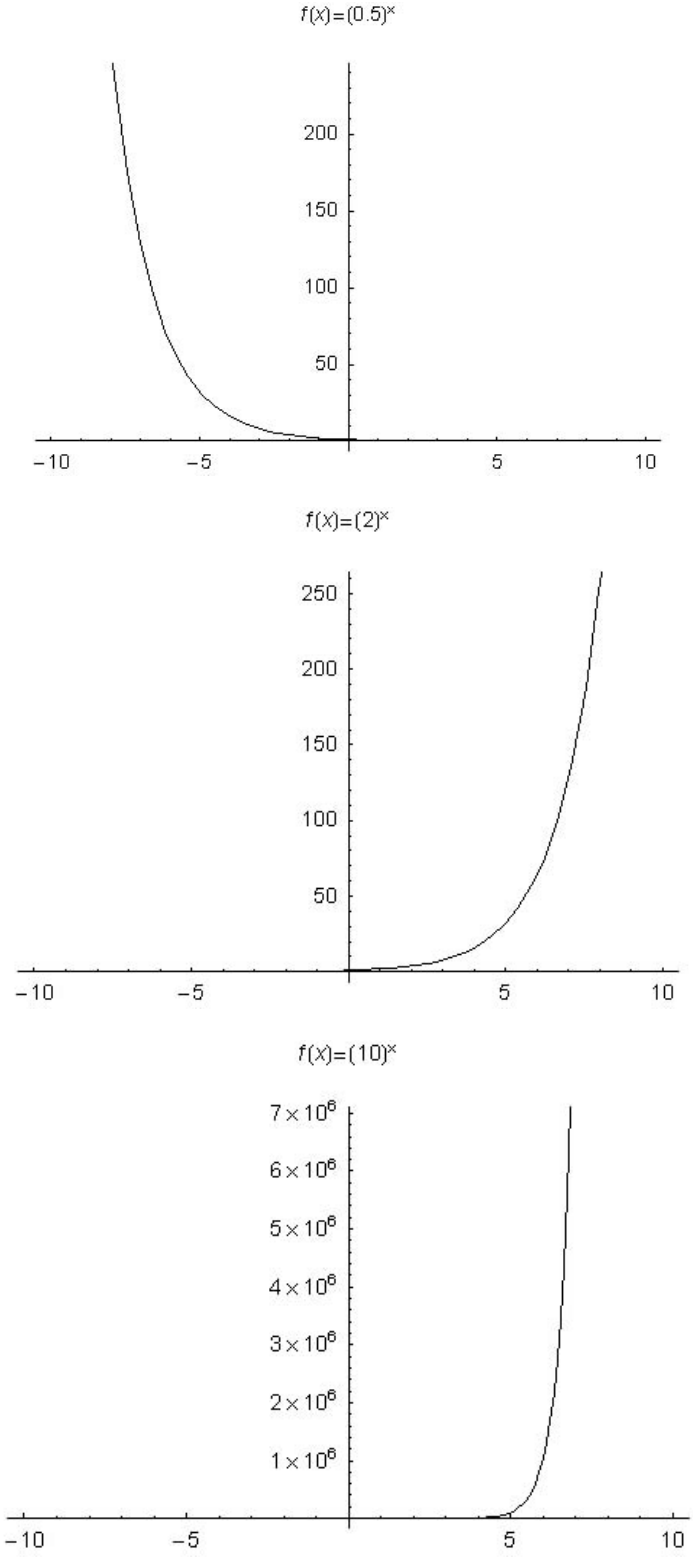


Figure 2.11: The plot of the exponential function for various bases.

As you can see, for fractional bases (those between 0 and 1) with a positive exponent, the function decreases rapidly from left to right; for larger bases with a positive exponent, the function increases rapidly. If you change a positive exponent into a negative exponent, you reverse these results (which should make sense, since a^{-x} is just $\left(\frac{1}{a^x}\right)$).

So just how rapidly does the exponential function change? To answer this question, let us look at the exponential function with base 2. Well, for x^2, if you walked along the function until you got to $x = 100$, and then walked one more unit right, you would have to hike 210 units straight up to remain on the graph. With the base 2 exponential functions, you would have to hike 1,267,650,600,228,229,401,496,703,205,376 units straight up under the exact same circumstances to remain on the graph of the function.

This should tell you that you definitely do not want to be hiking the graphs of exponential functions any time soon! The exponential function changes *exponentially* faster than all the other functions we have looked at.

There is one particular base that distinguishes itself from all others: called the **natural base**, this number is so special it is given the unique symbol '*e*'. Its approximate numerical value is 2.7182818284590452354.

The exponential function with base *e* is called the *natural exponential function*, or sometimes, just *the* exponential function (which reflects its importance).

What is so special about the natural exponential function? For one, it is the only function that is its own derivative (a topic covered in calculus). But perhaps more importantly for our purposes, the function is special because it lends itself to modeling real-world phenomena; for example, the function has been used to model all of the following:

- To predict how long it takes for radioactive isotopes to decay
- To determine what effects various interest rates have on a principal investment
- To predict how fast populations are expanding
- To model how well people remember things
- To model the way fog density increases with distance

There are many applications for games, such as modeling how long it takes for a burning building to reduce to ashes (for which the exponential function is a much better choice than what most games nowadays use), simulating damage to a vehicle (you can make it so the more damaged a vehicle is, the more easily it takes damage, which is a fairly realistic approach), and many more. Rather than cover all possible applications, we are going to focus on just a few, but cover them in sufficient depth so that you will feel comfortable using exponential functions for any application you may dream up.

The most critical application you may run across is fog density, since both OpenGL and Direct3D models both use the exponential function directly.

Exponential Fog Density

Fog is a generic term that refers to any kind of atmospheric suspension of particles (typically water droplets and dust) that is dense enough to cause light to noticeably scatter. The fog we are all most familiar with is the whitish haze that appears whenever moisture is available and there is some sort of localized cooling.

Fog is a nice effect in computer games that has been used both to add realism and to increase performance (you do not have to display anything that is so deeply obscured by fog that it cannot be seen by the user). To implement fog, the basic problem you have to solve is this: given a point at a certain distance from the user, what should the intensity of the fog be at that point?

The simplest of answers to this question involves choosing a start and an end distance for the fog. The fog will not obscure anything closer to the user than the start distance and anything further than the end distance need not even be displayed. Objects that fall in between these points are given a fog intensity (which ranges from 0 to 1) computed by the following *linear* function:

$$f(d) = 1 - \frac{(d_{end} - d)}{(d_{end} - d_{start})}$$

where d_{start} is the distance the fog starts at, and d_{end} is the distance it ends at.

This function is called *linear* because its graph is a straight line for any given values of d_{end} and d_{start}. Figure 2.12 graphs a part of the function, using $d_{end} = 10$ and $d_{start} = 0$.

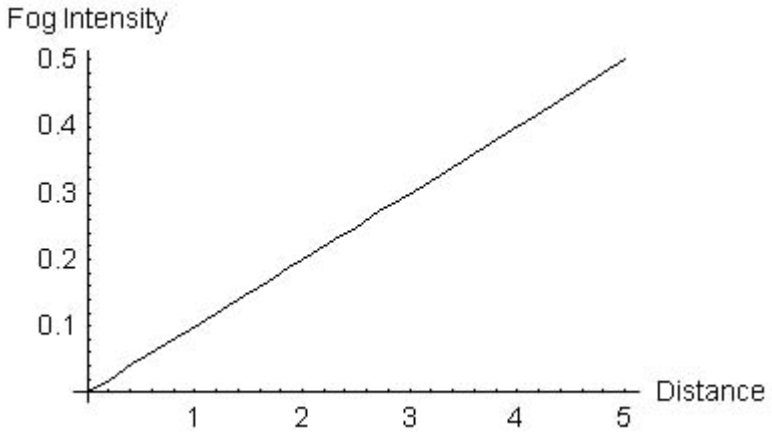


Figure 2.12: The graph of a linear fog function.

The function's range is the set of all real numbers between 0 and 1. A value of 0 indicates no fog at that point, and a value of 1 indicates full fog. Notice in the graph that instead of calling the axes '*x*' and '*y*', we have called them "Distance" and "Fog Intensity" to make things clearer.

This approximation works well enough that the majority of games can use it to simulate fog, but we can certainly improve on the model. It turns out for various physical reasons that the perceived intensity of fog increases exponentially with distance, rather than linearly. Recall that when you are in fog, the objects in your immediate vicinity are clear and sharp and there is little if any fog obscuring them. Then at some further distance, the intensity of the fog increases quickly until you cannot see anything beyond that point. The photograph in Figure 2.13 illustrates this effect.

Figure 2.13: A photograph showing how fog intensity increases with distance.

This is where the exponential function comes in. Rather than invent our own equations using the exponential function, we will look at two models built directly into popular rendering APIs like Direct3D and OpenGL: the *exponential model* and the *squared exponential model*.

Both of these models make use of something called the *fog density*, which is used to tweak just how foggy the environment appears. This value is used instead of the start and end distances in the linear model, but serves much the same purpose.

The fog density can be set to any value between 0 and 1, and hence, is another variable -- just like the distance to the point whose fog intensity we wish to compute. Thus the functions for both the exponential model and the squared exponential model map from a set of ordered pairs (consisting of the distance to the point and the fog density) to the set of real numbers (the fog intensity, between 0 and 1).

In the exponential model, the fog-mapping function is defined as follows:

$$f(d,\rho) = 1 - e^{-d\rho}$$

where ρ is the fog density.

So you can see what this function actually does, it is graphed in Figure 2.14 for all fog densities and for all distances from 0 to 5.

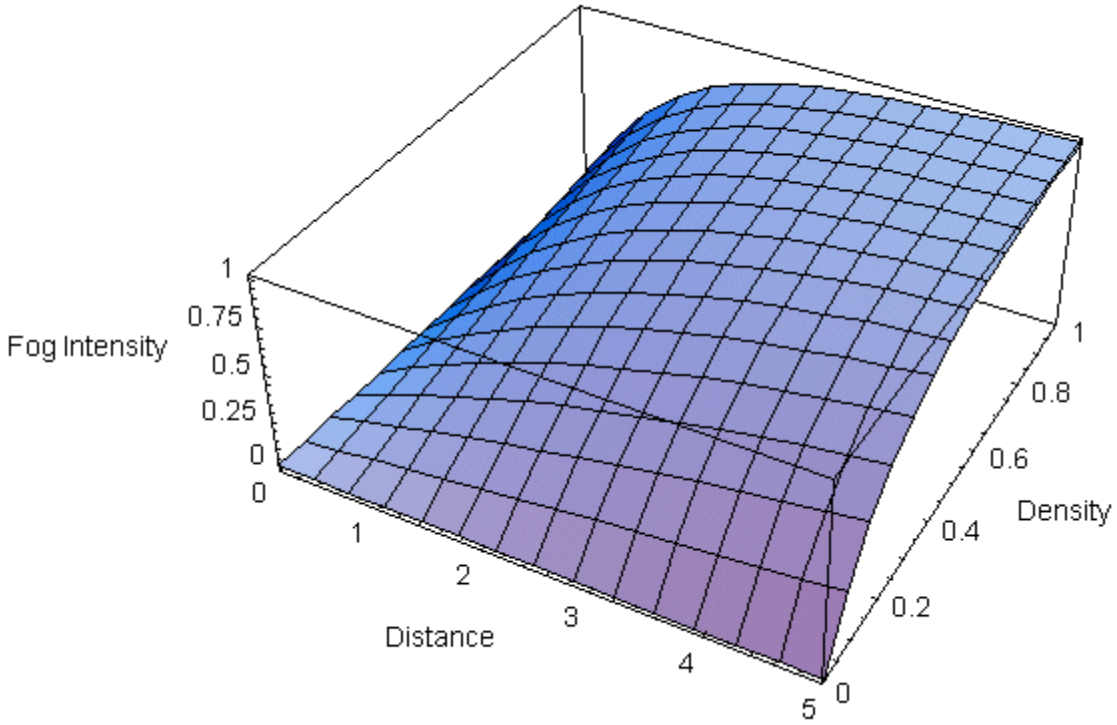


Figure 2.14: A graph of the exponential fog-mapping function.

Notice how that the fog intensity is very low for small distances, and then increases rapidly for larger distances. Exactly when this occurs is governed by the fog density: lower fog densities allow the user to see much more of the immediate vicinity than do higher densities.

The squared exponential model is very similar to the exponential model, as shown below:

$$f(d,\rho) = 1 - e^{-(d\rho)^2}$$

The squaring of the $d\rho$ makes the fog intensity increase much faster than in the exponential model. You can see this for yourself in Figure 2.15.

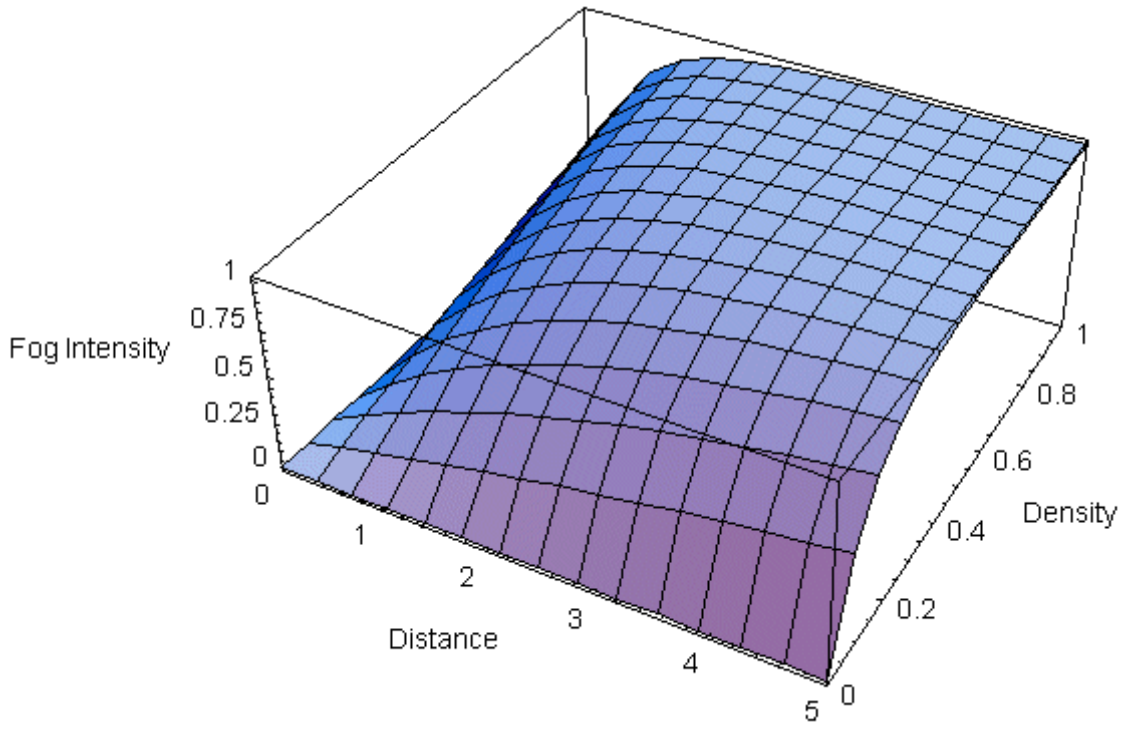


Figure 2.15 A graph of the squared exponential fog-mapping function.

These fog models provide substantially more realistic results than the linear fog equation, and show that even a topic as abstract and mathematical as exponential functions can be extremely useful in game development.

Next we are going to go step-by-step through the process of creating our own model of how an object takes damage, using the exponential function for increased realism.

Taking Damage the Exponential Way

Suppose you are working on a 3D role-playing game where you battle monsters, spirits, and evil wizards, and want to implement a system for how the various items of armor that your character wears take damage. For example, your character probably has a shield, a helmet, mail, gloves, and perhaps boots. When enemies attack your character, you will want to damage these items by some realistic amount.

Certainly, the amount of damage your character's armor takes is going to depend on how *strong* the enemies are and how *good* their weapons are. A strong enemy with a good weapon will do more damage than a weak enemy with a good weapon, for example. At a minimum, then we are going to want to create a function that inputs these two variables and outputs the damage done by the attack.

As our first attempt, we might try a function similar to the following:

$$f(a,b) = a + b$$

where *a* and *b* are the strength of the enemy and the rating of his or her weapon, respectively. This is not a particularly bad attempt, though it may seem arbitrary: why not multiply *a* and *b* together, or sum multiples of *a* and *b*, or throw in a few powers of *a* or *b*? These are all possibilities, and they will affect how your combat system works. Which one you choose ultimately depends on your preferences (unless you can fund some studies to figure out which one is most realistic). For example, if you would like to give preference to strength, under the theory that a really strong opponent with a flimsy weapon will be able to overcome a weaker one with an excellent weapon, then you could use the following function:

$$f(a,b) = 2a + b$$

For our purposes, we will just use the sum of *a* and *b* and work on improving its realism through more objective means.

Rather than increasing the damage done to an item by a fixed amount for a given enemy and weapon, it makes sense to increase it by an amount that varies depending on how badly it has already been damaged. For example, a brand new shield in perfect working order should take less damage than one that is falling apart. So what we really should do is have the item always take some sort of base damage, and then add to this an amount that increases as the overall damage of the item goes up.

We will run through an example to make this clearer. Say our shield is undamaged and has 100 hit points. An opponent comes along who has 10 points for strength, and is carrying a weapon with a rating of 5 points. Thus the base damage the opponent does to the player is 10 + 5 = 15. When the opponent first strikes, this is all the damage he does to the player's shield (since the shield is undamaged), so the shield ends up with 85 hit points.

After the first strike, the shield is damaged; so when the opponent strikes again, we will throw in 2 hit points above base value, bringing the total damage done by the opponent to 17. This reduces the shield's hit points to 68. In the next attack, we will add 4 hit points to the base value, reducing the shield's hit points to 49. In the fourth attack, we will add 8 hit points to the base value, reducing the shield's hit points to 26. In the fifth and last attack, we will add 16 points to the base value, making the shield completely unusable. (Of course, in a more realistic example, we would also reduce the hit points of the enemy's weapon, thus downgrading its ability to deal out damage, and possibly also decrease the strength of the enemy as fatigue set in from the battle.)

You can see that our damage function is going to be a function of *three* variables: the strength of the enemy, the rating of his or her weapon, and the damage already done to the item being attacked. Mathematically, we can express this as follows:

$$f(a,b,c) = a + b + g(c)$$

where *c* is the damage already done to the item and $g(c)$ is a new function that computes how much *additional* damage should be done to the item based on the damage it has already sustained. For convenience, we will let *c* range between 0 and 1, where 0 represents no damage, and 1 represents full damage.

Our task now is to figure out exactly what $g(c)$ looks like. We know immediately it is going to look like an exponential function, since that is what this section is all about! But if we did not already know this,

how would we come to this conclusion? One way would be to graph a function based on the example described with the shield, as in Figure 2.16.

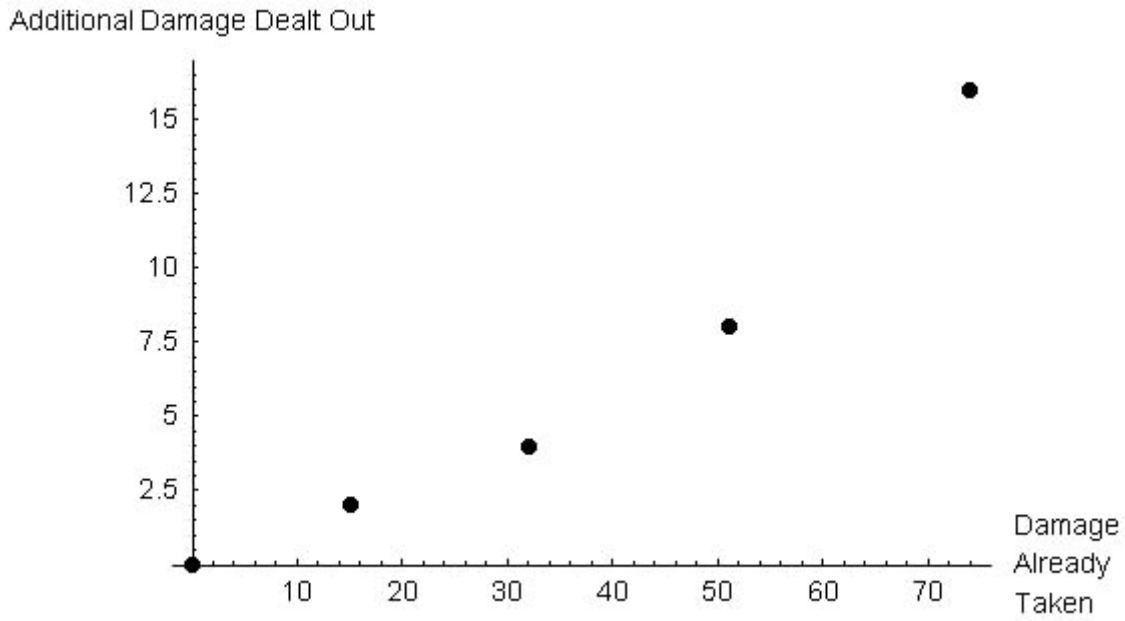


Figure 2.16: A graph of damage already taken versus additional damage dealt out.

As you can see, the function is definitely exponential.

The form we will choose for $g(c)$ is shown below:

$$g(c) = k_1 e^{k_2 c} + k_3$$

Why add k_1, k_2, and k_3 to the basic exponential function? Because we want to be able to have some degree of control over how $g(c)$ operates -- and these new constants will let us do exactly that. Otherwise, we would just have to accept whatever the exponential function gave us, which is not what we want.

In order to determine what the values of k_1, k_2, and k_3 are, we are going to have to be more precise in describing how $g(c)$ works. Certainly, we want $g(0) = 0$ (that is, when the item is not damaged, no additional damage should be added to the base damage). This, in turn, implies that $k_1 e^0 + k_3 = 0$, but any number raised to the power of 0 is just one, so we have $k_1 + k_3 = 0$. This is good because it tells us that k_1 and k_3 are related in a very specific way -- one is the negative of the other. We will need this information later on.

What else can we say about $g(c)$? Here we enter murky waters. We can describe $g(c)$ more, but only by making subjective choices about how the damage dealing should work. Somewhat arbitrarily, we will

choose to add 1 hit point of damage if the item is 10% damaged. We can express this mathematically as follows:

$g(0.1) = 1$

This gives us the following relationship:

$1 = k_1 e^{k_2 (0.1)} + k_3$

Our preference will be for $g(1)$ to be a very high value, since an item that is nearly completely damaged should not be able to withstand any more. So we will choose $g(1) = 100$, which although tending toward the high side, should work fine for most items. Mathematically, we can express this additional constraint as shown below:

$100 = k_1 e^{k_2 (1)} + k_3$

These two equations, combined with third equation $k_1 + k_3 = 0$, enable us to determine what the values of k_1, k_2, and k_3 are. Mathematica™ gives the following results:

$k_1 \approx 2.10999, \quad k_2 \approx 3.87937, \quad k_3 \approx -2.10999$

The squiggly equals sign means "is approximately equal to", which is a roundabout way of saying the terms are irrational numbers and therefore do not have a finite decimal expansion.

Plugging all of the values back into the original equation, we finally get what we are after:

$g(c) = 2.10999(e^{3.87937c} - 1)$

The relevant part of this function is graphed in Figure 2.17. As you can see, the function satisfies all of our three conditions: $f(0) = 0$, $f(0.1) = 1$, and $f(1) = 100$. Plug it into our expression for *f*, and it is ready to use in your game.

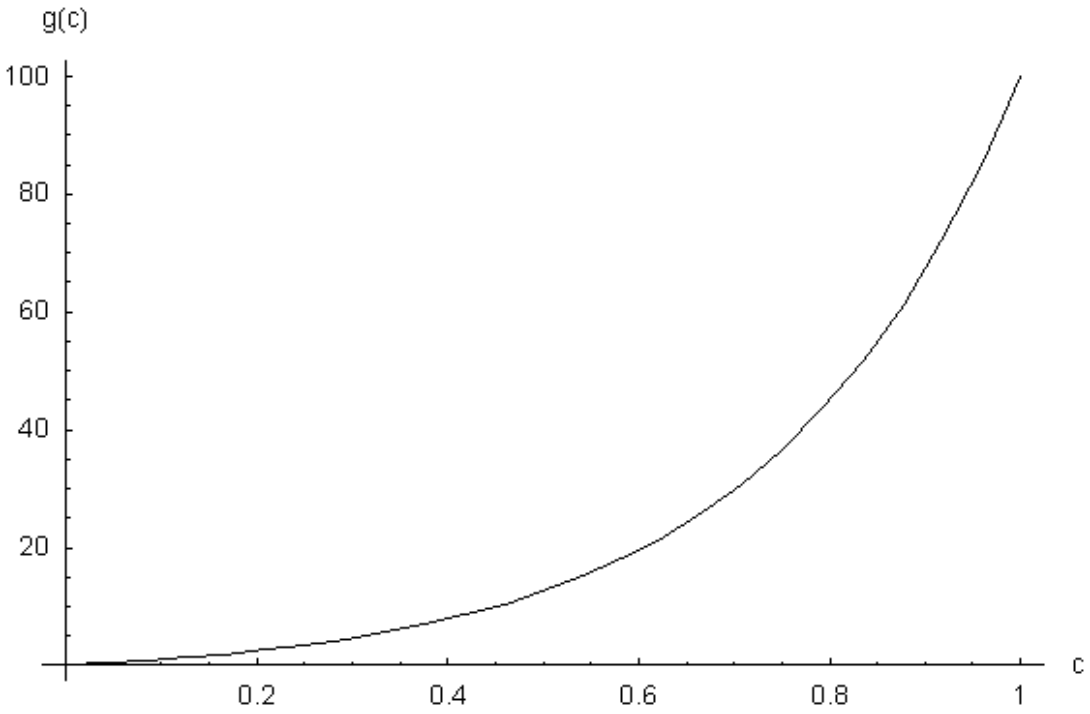


Figure 2.17: The graph of $g(c) = 2.10999(e^{3.87937c} - 1)$ for all *c* in [0,1].

We have covered exponential functions enough that you should be familiar with how they work and have a rough idea of how to use them to model things in your games. Now we will move on to logarithmic functions, which are closely related to exponential functions.

2.3.3 Logarithmic Functions

Logarithmic functions are very similar to exponential functions (and with good reason, as we will see shortly). Recall that there is no *single* exponential function -- rather, there is a *family* of exponential functions, each one with a different base. The same is true for logarithmic functions (or log functions, as we will call them for short): there are many different log functions, each one with its own base.

Log functions are represented by the notation $\log_a(x)$, where *a* is the positive base of the function. Just as with exponential functions, the base of a log function is a positive real number -- zero or negative bases are not allowed. Unlike exponential functions, however, there is a restriction on the *domain* of the log function: the domain is strictly the set of all positive real numbers.

So what is it that log functions do, anyway? To be precise, $\log_a(x)$ is a real number *y* such that $a^y = x$; in other words, $\log_a(x)$ is a number such that if you raise *a* to this power, you get *x*. This may seem like a complicated definition (and it is!), so we will examine some examples:

$\log_{10} 100 = 2$, since if you raise 10 (the base) to the power of 2, you get 100.
$\log_2 8 = 3$, since if you raise 2 to the power of 3, you get 8.
$\log_a 1 = 0$, since if you raise any base to the power of 0, you get 1.
$\log_8 64 = 2$, since if you raise 8 to the power of 2, you get 64.

These examples should hopefully clarify what it is that log functions do: they give you a real number such that, if you raise the base to this number, you get whatever real number you sent to the function.

Exponential functions are fast changing functions, as we saw in the last section. What about log functions? Log functions, as it turns out, have a reputation for growing extremely *slowly* for the vast majority of the graph (they do grow quickly in the beginning, however). Walking along the graphs of log functions is not too much more difficult than walking on a level plane.

So you can see this for yourself, we have graphed some log functions with a number of different bases in Figure 2.18. Remember that since the log's domain is the set of all *positive* real numbers, you will not see anything to the left of the origin. This may seem a bit odd at first, since the previous functions we have graphed are visible everywhere, but we will see many more functions in the coming lessons whose domain is restricted in like fashion.

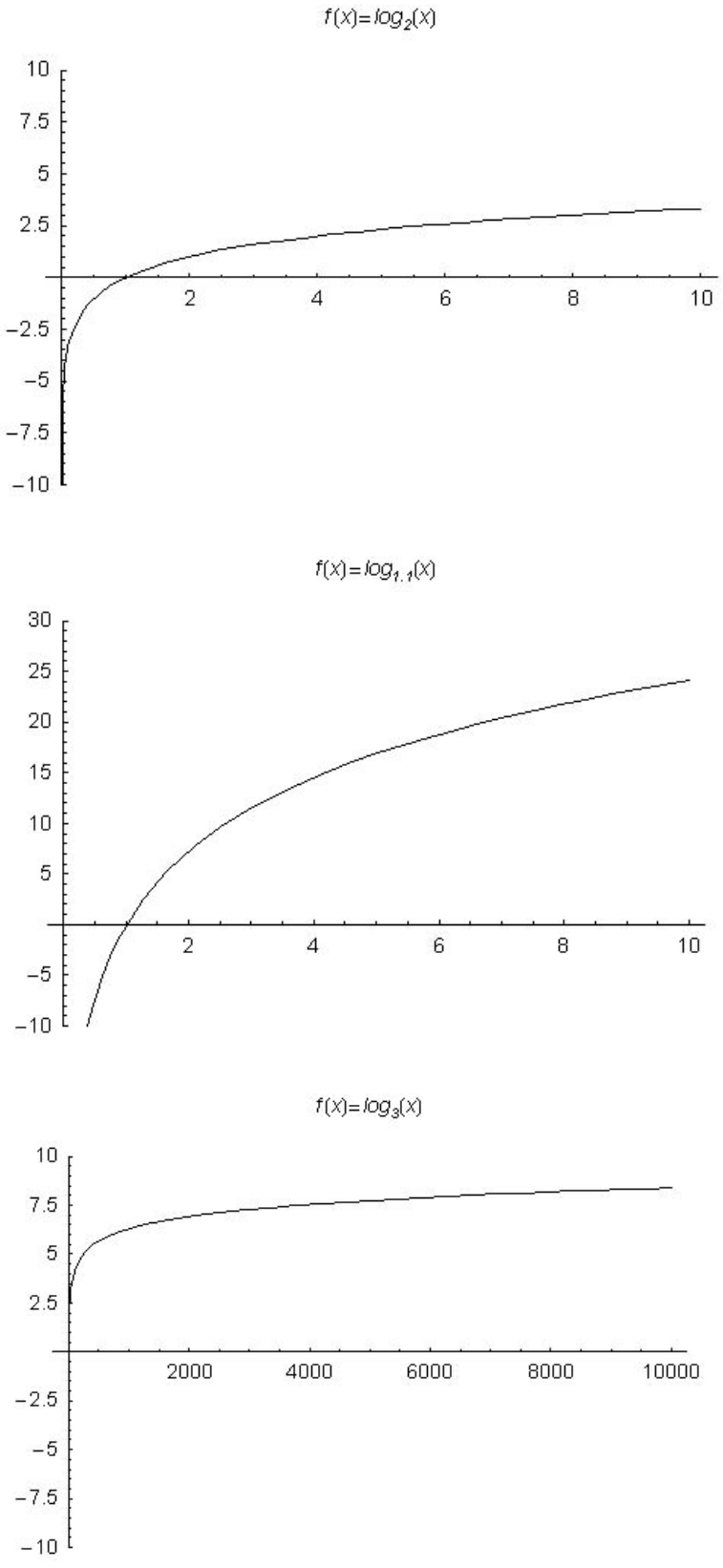


Figure 2.18: Graphs of some various log functions.

Perhaps the most interesting property of log functions is that $\log_a(x)$ is the inverse function for the exponential a^x; or to put it another way, $\log_a(a^x) = x$. Recall from our last lesson that an inverse function *undoes* the operation of the function: if a function *f* sends an element *x* in *A* to *y* in *B*, then the inverse of *f* (denoted f^{-1}) will send *y* in *B* to *x* in *A*.

Just like the natural exponential function is the exponential function with base *e*, the *natural log* is the log function with base *e*. Instead of denoting this function with $\log_e$, mathematicians often shorten it to **ln**, which stands for natural log (in reverse word order!).

If you just see $\log$ in a textbook with no base, then you are pretty safe in assuming the implied base is 10, unless it is a computer science book, and then the implied base is probably 2. Also, sometimes mathematicians get lazy and drop the parentheses from the log function, so it is quite common for textbooks to write something like $\log 5$ instead of $\log(5)$.

Log functions have a number of useful properties, which are summarized for you in Table 2.1.

Table 1: Useful properties of log functions.

1. $\log_a(uv) = \log_a(u) + \log_a(v)$	$a \neq 0, u > 0, v > 0$
2. $\log_a(\frac{u}{v}) = \log_a(u) - \log_a(v)$	$a \neq 0, u > 0, v > 0$
3. $\log_a(u^n) = n\log_a(u)$	$a \neq 0, u > 0$
4. $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$ (Change of Base Formula)	$a \neq 1, b \neq 1, x > 0$

The Change of Base Formula listed in Table 2.1 is quite handy, since unless you have a really nice calculator, the one you have *will not* support bases other than 10 or *e*. If you want to use other bases, you will first have to convert from these bases to one that the calculator supports.

The exercises ask you to prove several of the properties listed in Table 2.1. To give you a feel for how you should go about this, we will prove the first and last properties now:

Proof of Log Property (1): By definition, $\log_a(u) + \log_a(v) = y + z$, where $u = a^y$ and $v = a^z$. Here we have a representation for *u* and *v*. Multiplying these together, we get $uv = a^y a^z = a^{y+z}$. Taking the log of both sides of this equation (with base *a*), we find that $\log_a(uv) = \log_a(a^{y+z})$. The right hand side of this equation simplifies to $y + z$, so we have, $\log_a(uv) = y + z$. But $y + z = \log_a(u) + \log_a(v)$, so $\log_a(uv) = \log_a(u) + \log_a(v)$. This completes the proof.

Proof of Log Property (4): Let $y = \log_a(x)$. By definition, this implies that $a^y = x$. Taking the log of both sides (with base *b*), we have $\log_b(a^y) = \log_b(x)$. By Log Property (3), we can write the left hand

side of this equation as $y\log_b(a)$, so the equation becomes $y\log_b(a) = \log_b(x)$. Solving for *y*, we get $y = \frac{\log_b(x)}{\log_b(a)}$. But $y = \log_a(x)$, so we know $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$. This completes the proof.

Right now you probably know more about log functions than you want to! But the knowledge certainly will not go to waste -- the log function is invaluable when you are dealing with exponential functions, since it enables you to solve equations you would not otherwise be able to solve. We will wrap up this chapter's discussion by looking at an example that illustrates this point perfectly.

Using the Log Function for Game Development

Many role-playing games use a system that allows a character to advance, whether in strength, dexterity, or some skill such as thievery. Typically, your character gains experience points by completing quests or killing bad guys. After acquiring a certain number of experience points, your character advances to the next level (a process termed "leveling up"), and you can choose what areas of your character you want to develop more.

With open-ended games that allow you to do the same quests over again (or just kill monsters endlessly to gain experience points), one of the challenges developers face is what to do when the players become *too* powerful, and killing common enemies or completing low-level quests becomes too easy. If developers made it so that killing a Beta Werewolf brought you 10 experience points, and you leveled up every 100 points, you could just spend all your time killing these monsters until your character became mega-powerful.

To avoid this problem, developers often make each new level harder to reach than the last. After reaching level 2, for example, perhaps it would require 200 experience points to make it to level 3, which would mean you would have to kill *twice* as many Beta Werewolves. This motivates players to play at a level more appropriate for their character, since the harder the enemies or quests, the more experience points the characters gain for a victory.

One way to implement this solution is to use an exponential function, which ensures that each new level will be harder to reach than the last. Specifically, we can solve the problem with an exponential function $f(x)$ such that *x* is the current level of the character and $f(x)$ is the number of experience points required for the character to level up.

As before, we will setup a generic function involving the natural exponential, only this time we will use multiplicative constants (this will make the problem solvable with the mathematical tools we have developed thus far). Such a function is shown below:

$$f(x) = k_2 e^{k_1 x}$$

Similarly, we will need some constraints so that we can determine the values of the two constants. Two that we are going to use are $f(0) = 10$ and $f(100) = 10000$. The first one says that a character at level 0 has to acquire 10 experience points in order to level up, and the second one says a character at level 100

has to acquire 10000 experience points to level up. (This should be considered the practical upper limit for character development, since the exponential function will grow so fast beyond this point that higher levels will be virtually unattainable.)

Mathematically, we can express these two constraints as follows:

$$k_2 e^0 = k_2 = 10$$
$$k_2 e^{100k_1} = 10000$$

The first equation tells us that $k_2 = 10$. Substituting this value into the second equation, we get,

$$10e^{100k_1} = 10000$$

Dividing both sides of the equation by 10, we find that,

$$e^{100k_1} = 1000$$

Now what do we do? Without the log function or a computer algebra system like Mathematica™, we would be reduced to trying different values for k_1 in hopes of stumbling on the one that made the equation true (there are actually more sophisticated guessing methods than this, but we have not covered them). With an infinite number of real numbers to try, our odds of guessing the right one are not very good.

Fortunately we do not have to guess. The log function acts as an inverse for the exponential function, so all we really have to do is take the natural log of both sides. This will give us the equation shown below:

$$\ln(e^{100k_1}) = \ln(1000)$$

This simplifies to the following:

$$100k_1 = \ln(1000)$$

Dividing both sides by 100, we finally isolate k_1:

$$k_1 = \frac{\ln(1000)}{100} \approx 0.0690776$$

So our final expression for the function *f* is as follows:

$$f(x) = 10e^{0.0690776x}$$

The graph of this function from 0 to 100 is shown in Figure 2.19.

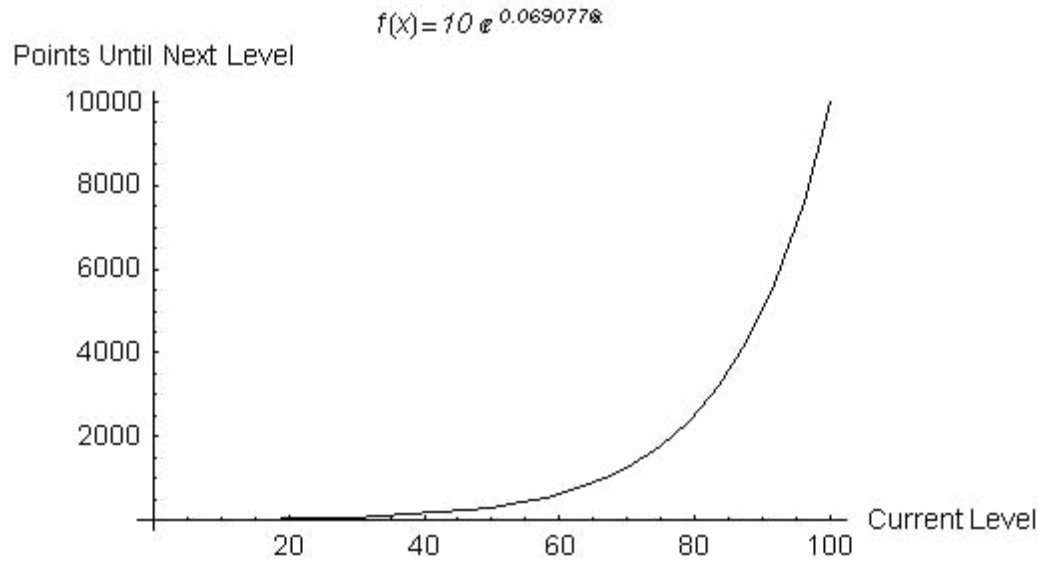


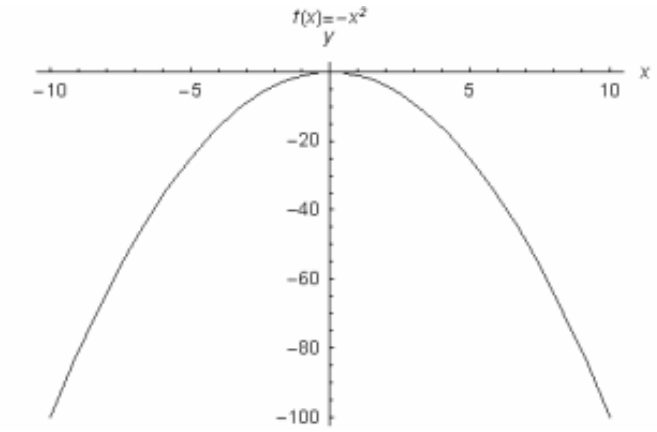
Figure 2.19: The graph of the function $f(x) = 10e^{0.0690776x}$

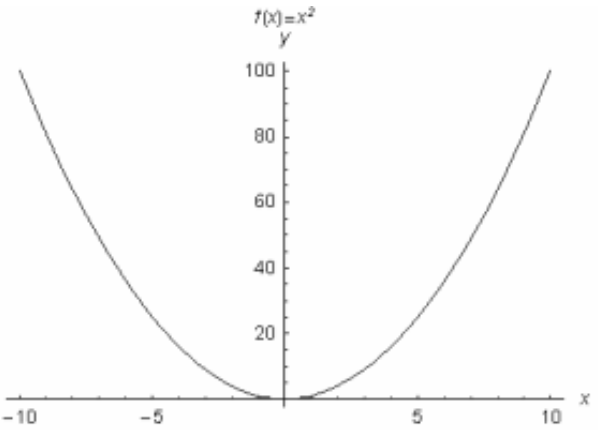
Conclusion

In our next lesson, we are going to look at an altogether different class of functions: the class of *polynomial* functions. These handy functions, which are simply sums of powers of one or more variables, can be used to draw polygons or lines, approximate functions, predict the future, and much more.

Chapter Three

Polynomials





Introduction

The basement was a complete mess. Sawdust and a fine white powder covered the floor, boxes of packed goods were clustered in the corners, and slabs of wood and drywall were stacked in each room. The basement was not being remodeled -- it was being finished. And as I could tell from my monthly inspection, it had a long way to go.

I was at my friend's house to visit and as we chatted about current world affairs and various philosophical topics, I helped him drywall the ceiling in one of the rooms. This arduous process consisted mainly of cutting pieces of drywall and screwing them to the overhead wooden beams.

The job went smoothly enough, until we came to the ceiling light fixture in the center of the room. This protrusion required a custom cut in exactly the right place and in precisely the right shape. The first part was easy: we just used a measuring tape to locate the fixture relative to where we knew the drywall slab would be placed. Once measured, we drew a corresponding 'X' on the slab itself to indicate where we would need to cut the hole.

The hard part was the cutout itself. Fortunately, the base of the light fixture was circular, so we measured the diameter of the base and drew a square of matching dimensions on the drywall slab, centered on the 'X'. Our task, we reasoned, was now to draw the largest possible circle that would fit in that square. We of course did what all men would do, and that is search around for some sort of can in our immediate vicinity that we could use to trace our circle with. Alas, the cans we found were either far too small or too large, and being the professionals we were, we could not accept anything less than a perfect fit.

My friend began sketching a circle, but the resulting shape was too crude for our needs. So we pondered our situation for a few moments. We could always go upstairs and search for a compass, but that seemed like such a drastic step to take for something that *real* men should be able to do without assistance. No, a compass was not the solution, we concluded. There had to be another way.

As my friend started drawing little equidistant marks on the sides of the square, suddenly a brilliant solution to our problem hit me: "We can use this square as a piece of graph paper, derive polynomial approximations to the sine and cosine functions, and then use these functions to plot the graph of a perfect circle!" This idea would have worked perfectly, too. The only problem was we did not have a calculator, and so we would have had to do the calculations manually (and there would have been hundreds of them to do).

While I lamented the fact that mathematics was not able to rescue us, my friend solved the problem by drawing a series of lines from marks on one side of the square to marks on an adjoining side, creating a near-perfect circle in the process (a trick he says he picked up when he was quite young). So much for all those years in college studying math!

Still, you may be wondering if it is possible to draw a perfect circle using polynomials. The answer is a definite yes. Polynomials can be used to approximate a large number of functions; indeed, they are so

good at it that various branches of mathematics study polynomials exclusively, knowing that the results of their study are applicable to many other kinds of functions as well.

Approximating other functions is just one use for polynomials: like exponential and logarithmic functions, polynomials can be used to model various phenomena -- phenomena even more complicated than what you can model with exponential and log functions. They can also be used to predict future events based on past events (a useful feature for multiplayer games), to represent three-dimensional curved surfaces, and to direct the paths of computer characters.

So polynomials must be pretty important, right? You bet. In fact, they are so important that we will spend this *entire* lesson looking at nothing but polynomials. By the end of the material, you will be able to astound your friends and amaze even yourself with all of the cool stuff you will be able to do. And you will also have taken your first steps toward being able to finish or remodel your *own* basement without a compass -- just do not forget to bring a calculator!

3.1 Polynomials

A polynomial is a simple algebraic construction involving sums of *polynomial terms*. A *polynomial term* is just a real number times the product of one or more variables, where each variable is raised to some non-negative integer power. A polynomial that uses certain variables is said to be a polynomial *in* those variables; for example, a polynomial that uses the variable *x* is said to be a polynomial *in x*.

The following are all examples of polynomial terms: $-2x$, $4x^3y^2$, $3xyz$, and x^2z^{10}. Each of these polynomial terms is itself a polynomial. Here are some examples of polynomials that have more than one polynomial term: $-5x^3+2x$, x^5+y, and $x^2+y^2+z^2-x$.

Every polynomial has a *degree*, which is defined differently depending on how many variables appear in the polynomial. For polynomials of a single variable, the degree is the largest power the variable is raised to. The degree of the polynomial $-x^3+x$, for example, is 3, since 3 is the largest power that *x* is raised to.

For polynomials of several variables, the degree is the largest *combined* exponent of the terms. By combined, we mean that you sum the exponents of the variables in the terms (e.g. the combined exponent of the term xy is 2 -- the exponent of *x* plus the exponent of *y*). The degree of the polynomial $xy-2x^2+y^2z$, for example, is 3, since this is the largest combined exponent.

Symbolically, we can express a polynomial in *x* of degree *n* with the following notation:

$$a_nx^n+a_{n-1}x^{n-1}+a_{n-2}x^{n-2}+\cdots+a_2x^2+a_1x^1+a_0x^0$$

The *a*'s can be any real numbers at all, as long as $a_n \neq 0$. These real numbers are called the *coefficients* of the terms. The coefficient of x_i, where *i* is any number between 0 and *n*, is simply a_i; for example, the coefficient of x^2 is a_2.

The coefficient a_0 is called the *constant term*, and the coefficient a_n, the *leading coefficient* (since it comes first in the expression when you order the terms as shown).

The symbolic notation for a polynomial in *x* of degree *n* can actually be simplified somewhat, since x^0 is 1, and x^1 is just *x*. We can therefore rewrite the definition as follows:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

There are special names given to polynomials of certain degrees: a polynomial of degree 0 is called a *monomial* (because it has one term) or a *constant polynomial*; a polynomial of degree 1 is called a *binomial* (because it has two terms); a polynomial of degree 2 is called *trinomial* (because it has three terms) or, more commonly, a *quadratic*.

As a point of interest, a *rational expression* is a fraction where the numerator and the denominator are both polynomials. A *rational function* is just a function that maps from real numbers (or ordered *n*-tuples) to the real numbers by *using* a rational expression.

That is about all there is to the definition and representation of polynomials. There is no great mystery; in fact you have seen polynomials in previous lessons. Next we are going to discuss the algebra of polynomials -- all the stuff you can do with them -- but we will restrict our attention to polynomials of a single variable, which are the most important for our purposes. Keep in mind however, that fundamentally, a polynomial can have as many variables as you want.

3.1.1 The Algebra of Polynomials of a Single Variable

The operations defined for polynomials are pretty much the same as for real numbers: you can add them, subtract them, multiply them, divide them, and raise them to powers. You can even multiply and divide polynomials *in more than one way*.

The unfortunate part about this richness of polynomial algebra is that the set of polynomials is not *closed* under the last two operations: that is, if you divide one polynomial by another polynomial, the result may not be a polynomial (in general, it is a rational expression); similarly, if you raise a polynomial to fractional or negative powers, the result will generally not be a polynomial either. But despite these inconveniences, the operations do come up often enough that we will cover them here.

Adding and Subtracting Polynomials

Adding or subtracting two polynomials follows the rules of arithmetic you learned in high school. The new polynomial will have a degree equal to the largest of the degrees of the polynomials you are adding or subtracting. To compute the coefficients for this new polynomial, simply add or subtract the corresponding coefficients of the polynomials you are working with (some of them may be zero).

For example, let us say we have the following two polynomials we want to both add and subtract:

$x^4 - 3x + 5$
$4x^3 + 2x^2 - x + 1$

The first step we might take is to format the polynomials so their powers of *x* line up, putting zeros wherever certain powers are missing:

x^4	$0x^3$	$0x^2$	$-3x$	5
$0x^4$	$4x^3$	$2x^2$	$-x$	1

Now to add or subtract them, all we have to do is add or subtract the coefficients in each column. For example, the addition of the two polynomials is the new polynomial $x^4 + 4x^3 + 2x^2 - 4x + 6$. Similarly, the subtraction of the two polynomials is the new polynomial $x^4 - 4x^3 - 2x^2 - 2x + 4$.

Scaling Polynomials

You can multiply polynomials or divide them by any real number (except you cannot divide by zero). The result will still be a polynomial. To do so, you use the normal rules of algebra, treating the entire polynomial as if it were enclosed by parentheses (thus you multiply or divide each term individually by the real number). Let us look at some examples. If you multiply the polynomial $5x^7 + 10x$ by 2, then the resulting polynomial is $10x^7 + 20x$. If you divide the polynomial $3x^2 - 1$ by 2 (which is the same as multiplying it by 1/2), you get $1.5x^2 - \frac{1}{2}$.

Multiplying Polynomials

To multiply two polynomials together, you treat them as if they were both enclosed by parentheses, and then multiply the expressions together using the distributive property for real numbers. As mentioned briefly in lesson one, the distributive property for real numbers says that $a(b + c) = ab + ac$. This property provides a way of *distributing* multiplication inside of parentheses.

When you are multiplying a monomial by a binomial, the distributive law applies easily and directly: for example, the monomial 5 times the binomial $x - 2$ is just $(5)(x\text{-}2) = 5x - 10$, by the distributive law. If you are multiplying two binomials, the situation is a bit more complicated.

In general, a binomial times a binomial will look like $(ax + b)(cx + d)$, where a, b, c, and d are constant real numbers. To compute the result of this multiplication, we are going to let $e = (ax + b)$; that is, we are going to rename the quantity $(ax + b)$, and the new name we are going to use is e. Then the multiplication becomes $e(cx + d)$. By the distributive property, we know this is just $ecx + ed$. Now we will substitute the quantity $(ax + b)$ wherever we see e, since that is what we defined e to be. This gives us $(ax+b)cx+(ax+b)d$. We can apply the distributive property *two more times* to this expression, giving us $acx^2+bcx+axd+bd$. By factoring out an x, we can simplify this to $acx^2+x(bc+ad)+bd$. Notice the degree of the resulting polynomial is equal to the sum of the degrees of its constituent products -- *this is always true*, no matter the polynomials involved.

The lesson to learn from all this is that multiplying polynomials together is in general an unpleasant process: the more terms in the polynomials, the more unpleasant it becomes. Fortunately, there is an easier way than using the distributive property that can be extended to multiply any number of sums of terms by other sums of terms, whether or not those sums are polynomials.

Let us say we have n sums $S_1 = a_{11} + a_{12} + \cdots + a_{1m}$, $S_2 = a_{21} + a_{22} + \cdots + a_{2m}$, and so on, with the nth sum being $S_n = a_{n1} + a_{n2} + \cdots + a_{nm}$. To compute the product $S_1 \cdot S_2 \cdot \cdots \cdot S_n$, here is what you need to do: choose a term from each sum, and then multiply the terms together. Repeat the process until you have chosen all possible combinations. Add the resulting products, and you are done! Easier said than done, of course, but this way of doing it is much simpler than using the distributive property a thousand times.

Dividing Polynomials

Division of one polynomial by another is the most complicated of polynomial operations. Worse still, the end result is usually not a polynomial at all, but a rational expression. But despite these drawbacks, polynomial division is an invaluable tool for anyone who works a lot with polynomials (and that is, or *will be*, you!).

The method we are going to cover for polynomial division is called *long division*. It parallels long division for real numbers. The best way to explain this process is to look at a few examples.

Suppose we want to divide the polynomial $-4x^5+17x^4+11x^3-11x^2-6x$ (called the *dividend*) by the trinomial $4x^2+3x$ (called the *divisor*). The steps involved in this process are explained below:

1. Write the polynomials in long division form.

$$4x^2+3x\overline{)-4x^5+17x^4+11x^3-11x^2-6x}$$

2. Above the dividend, write down a polynomial term such that, when multiplied by the divisor, the first term of the resulting polynomial is identical to the first polynomial term in the dividend. Here the polynomial we want to write down is $-x^3$, since if you multiply this by the divisor, you get $-4x^5-3x^4$, and the first term of this resulting polynomial is equal to the first polynomial term in the dividend (namely, $-4x^5$).

$$4x^2+3x\overline{)-4x^5+17x^4+11x^3-11x^2-6x}\quad -x^3$$

3. Multiply the polynomial term you just wrote down by the divisor, and then write the resulting polynomial directly beneath the dividend, lining up the powers of *x*.

$$\begin{array}{l} 4x^2+3x\overline{)-4x^5+17x^4+11x^3-11x^2-6x}\quad -x^3 \\ \qquad\quad -4x^5\ \ -3x^4 \end{array}$$

4. Subtract the polynomial you just wrote down from the dividend, and write the resulting remainder directly below the polynomial (taking care again to align the powers of *x*).

$$\begin{array}{l} 4x^2+3x\overline{)-4x^5+17x^4+11x^3-11x^2-6x}\quad -x^3 \\ \qquad\quad \underline{-(-4x^5\ \ -3x^4)} \\ \qquad\qquad\quad\ 20x^4 \end{array}$$

5. Drop down a term from the dividend and add it to the remainder you wrote down in the last step. In this case, we drop down the $11x^3$.

$$\begin{array}{l} 4x^2+3x\overline{)-4x^5+17x^4\ +11x^3-11x^2-6x}\quad -x^3 \\ \qquad\quad \underline{-(-4x^5\ \ -3x^4)}\ \ \downarrow \\ \qquad\qquad\quad\ 20x^4\ +11x^3 \end{array}$$

6. This step is a repeat of step 2: above the dividend, write down a polynomial term such that, when multiplied by the divisor, the first term of the resulting polynomial is equal to the first term of the "remainder" polynomial at the bottom. In our case, the sought-after polynomial term is $5x^2$, since $5x^2(4x^2+3x)=20x^4+15x^3$, and the first term of this new polynomial is $20x^4$, which is the same as the first term in our remainder polynomial.

$$
\begin{array}{r}
 & -x^3 & +5x^2 & & \\
\hline
4x^2+3x\,\big) & -4x^5 & +17x^4 & +11x^3 & -11x^2-6x \\
 & \underline{-(-4x^5} & \underline{-3x^4)} & & \\
 & & 20x^4 & +11x^3 &
\end{array}
$$

7. This step is a repeat of step 3. Multiply the polynomial written above the dividend by the divisor then write the result directly below the remainder polynomial, aligning the matching powers of *x*.

$$
\begin{array}{r}
 & -x^3 & +5x^2 & & \\
\hline
4x^2+3x\,\big) & -4x^5 & +17x^4 & +11x^3 & -11x^2-6x \\
 & \underline{-(-4x^5} & \underline{-3x^4)} & & \\
 & & 20x^4 & +11x^3 & \\
 & & 20x^4 & +15x^3 &
\end{array}
$$

8. This step is a repeat of step 4: from our remainder polynomial, subtract the polynomial you just wrote down, and write the resulting polynomial at the bottom. This is the new remainder.

$$
\begin{array}{r}
 & -x^3 & +5x^2 & & \\
\hline
4x^2+3x\,\big) & -4x^5 & +17x^4 & +11x^3 & -11x^2-6x \\
 & \underline{-(-4x^5} & \underline{-3x^4)} & & \\
 & & 20x^4 & +11x^3 & \\
 & & \underline{-(20x^4} & \underline{+15x^3)} & \\
 & & & -4x^3 &
\end{array}
$$

9. This step is a repeat of step 5: drop down a term, adding it to the remainder calculated in the previous step.

$$
\begin{array}{r}
 & -x^3 & +5x^2 & & \\
\hline
4x^2+3x\,\big) & -4x^5 & +17x^4 & +11x^3 & -11x^2-6x \\
 & \underline{-(-4x^5} & \underline{-3x^4)} & & \downarrow \\
 & & 20x^4 & +11x^3 & \\
 & & \underline{-(20x^4} & \underline{+15x^3)} & \\
 & & & -4x^3 & -11x^2
\end{array}
$$

10. By now you should know exactly what is coming next. To spare you the painful details, we will skip to the end result, which is listed below:

$$
\begin{array}{r}
 -x^3 \;+5x^2 \;\; -x \;\; -2 \\
\hline
4x^2+3x \;\big)\; -4x^5+17x^4+11x^3-11x^2-6x \\
\underline{-(-4x^5-3x^4)} \\
\qquad\quad 20x^4+11x^3 \\
\qquad\quad \underline{-(20x^4+15x^3)} \\
\qquad\qquad\quad -4x^3-11x^2 \\
\qquad\qquad\quad \underline{-(-4x^3-3x^2)} \\
\qquad\qquad\qquad\quad -8x^2-6x \\
\qquad\qquad\qquad\quad \underline{-(-8x^2-6x)} \\
\qquad\qquad\qquad\qquad\quad 0
\end{array}
$$

11. As you can see, the final remainder is zero, so we can conclude that $4x^2+3x$ exactly divides $-4x^5+17x^4+11x^3-11x^2-6x$. The result of the division is the polynomial listed above the dividend: $-x^3+5x^2-x-2$. You should verify that this polynomial times the divisor is indeed the dividend.

This example went very smoothly: the dividend had all powers of *x* and the remainder of the division was zero. This was not a coincidence, either since we rigged the example to produce these results. The vast majority of polynomial divisions, however, will involve polynomials that are not so well behaved. We will cover one more example now that will show you what to do in these cases.

Suppose we want to divide the polynomial x^3-2 by the polynomial x^2-x. Notice that the dividend here does not have any powers of *x* aside from x^3 and x^0, and yet we will have to subtract such powers from it like we did in the previous example. So what do we do? The trick is noticing that it *does* in fact have the missing powers of *x*, but *with zero coefficients*. We can therefore write the dividend as x^3+0x^2+0x-2.

Now we proceed exactly as before. The final result of the division process is shown below:

$$
\begin{array}{r}
\quad x \;+1 \\
\hline
x^2-x \;\big)\; x^3+0x^2+0x-2 \\
\underline{-(x^3-x^2)} \\
\qquad\quad x^2+0x \\
\qquad\quad \underline{-(x^2-x)} \\
\qquad\qquad\quad x-2
\end{array}
$$

As you can see, the remainder *is not* zero in this case -- it is $x-2$. Remainders are so common that if you pick two polynomials at random, and divide the smaller into the larger, chances are you will get a remainder (in fact, the above two polynomials were selected at random, knowing that there would almost certainly be a remainder after the division).

So how do we interpret the remainder? The exact same way we do in long division: the dividend divided by the divisor is equal to the result of the division process plus the remainder divided by the divisor. In our case, this means that $\frac{x^3-2}{x^2-x} = x+1+\frac{x-2}{x^2-x}$.

3.1.2 Finding the Zeros of a Polynomial

Sometimes you will have a polynomial and will want to know exactly what values of *x* the polynomial sends to 0. You can get a rough idea by just graphing the function and looking for places the graph of the polynomial intersects the *x*-axis (more on this in the next section). But generally, we want an analytical expression that will give us exact values and leave no need for guesswork.

It turns out that this is easy for low degree polynomials and progressively harder for higher ones -- eventually, it is literally impossible to write down an analytical solution for the problem.

The monomial and binomial cases are easy, so we will not cover them here. The trinomial case is a bit more interesting. Essentially, we want to know what values of *x* will make the equation $ax^2+bx+c=0$ true. Mathematicians have discovered that a quadratic in this form always has two solutions, although they may be identical or even non-real (a topic we will cover more in lesson seven). The solutions are given by something called the *quadratic equation*, listed below:

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

That is just one equation -- so where are the two solutions? Simply use the positive square root to get one solution, and the negative square root to get the other (the '$\pm$' symbol just means a plus or a minus sign can go there). You should verify that both of these really are solutions to the equation $ax^2+bx+c=0$ (that is, if you plug these values of *x* into the equation, you get a true equality).

3.2 Visualizing Polynomials

To master polynomials, you must be able to intuitively grasp what the lower degree polynomials *do*. The best way to do this is to construct functions that use polynomials to map from the real numbers to the real numbers, and then to graph these functions and carefully study what the graphs look like.

The simplest polynomial of all is a zeroeth-degree polynomial, which is just a real number. A function that uses this polynomial will map *all* real numbers to a single real number. Thus the graph of such a function will be a perfectly straight, horizontal line (Figure 3.1).

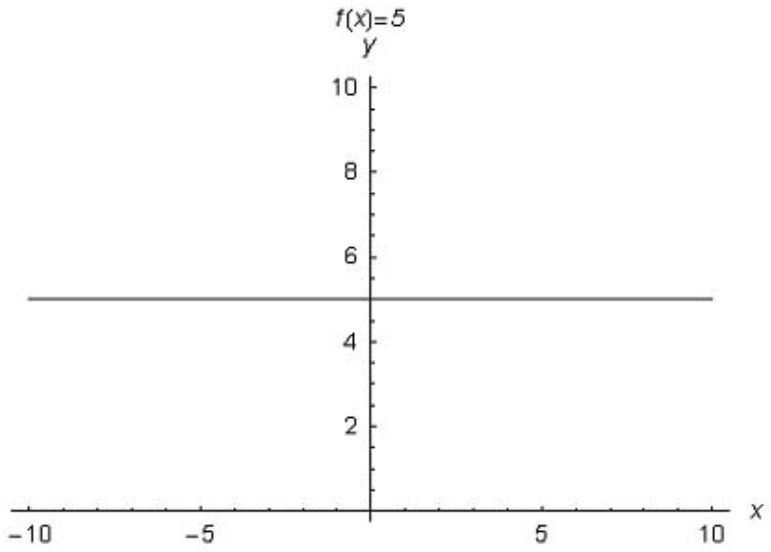


Figure 3.1: The graph of a zeroeth-degree polynomial.

The graph of a function that uses a first-degree polynomial is a straight line too, but this time the line can be oriented in any direction except perfectly vertical (if the graph was a vertical line, the function would have to associate an infinite many elements in the range with a single element in the domain, which violates the definition of a function). A first-degree polynomial is graphed in Figure 3.2.

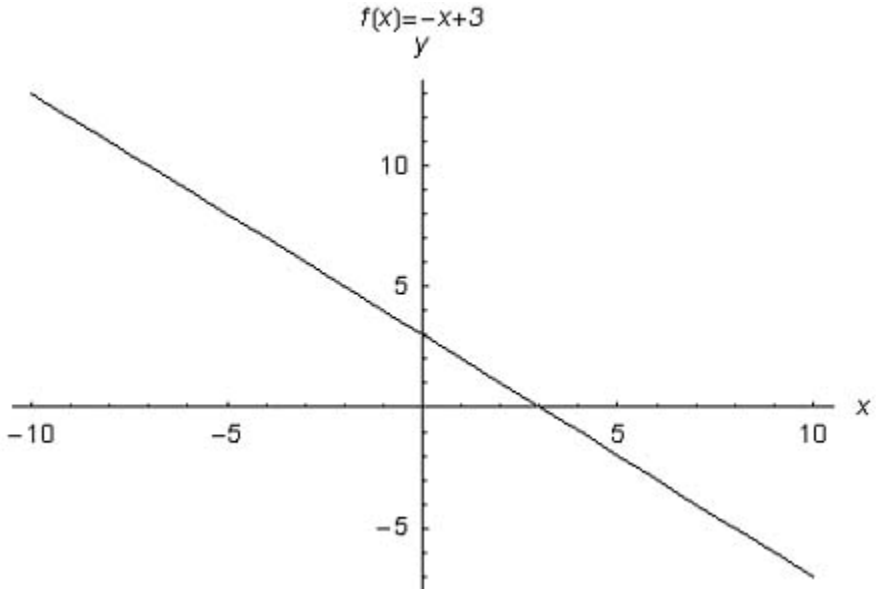


Figure 3.2: The graph of a first degree polynomial.

In our last lesson, we graphed the function $f(x) = x^2$, so you have already seen what a second-degree polynomial looks like. In general, the form of a second-degree polynomial function is $f(x) = ax^2 + bx + c$, where *a*, *b*, and *c* are all real numbers, and *a* is nonzero. If *a* is positive, as it is for $f(x) = x^2$, then the graph of the function resembles a 'U'. If *a* is negative, on the other hand, the graph looks like an *upside down* 'U' (a *parabola*). You can see these distinctions in Figure 3.3.

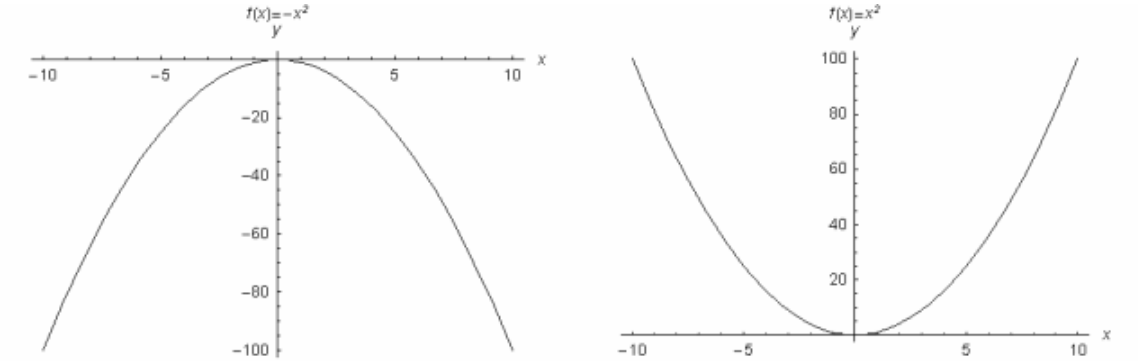


Figure 3.3: The graphs of some second degree polynomials.

A third-degree polynomial resembles a series of hills and valleys, although exactly how it does so depends on the coefficients of the powers of *x*. A few simple trinomial functions are graphed in Fig 3.4.

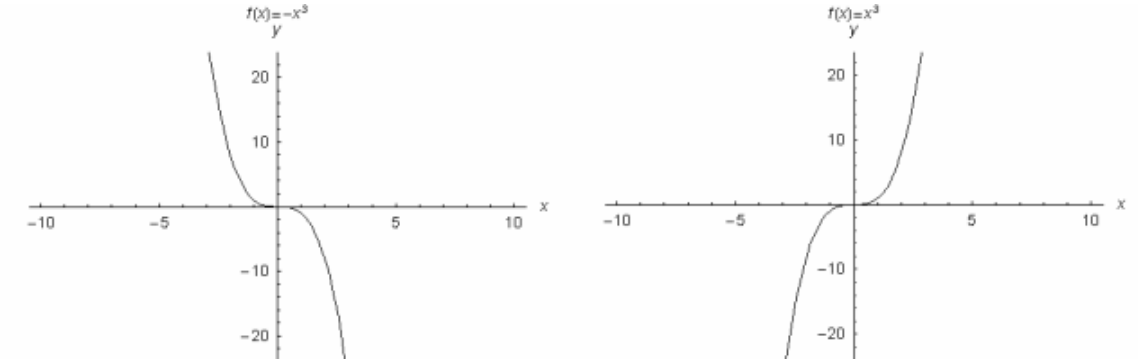


Figure 3.4: The graphs of some third degree polynomials.

The graphs of higher degree polynomials are more erratic, at least when you look around the origin. This is because of the wide variety of choices you have for all the different coefficients: they can be zero, small or huge, and negative or positive. But if you become familiar with the graphs presented here, you will be more than able to handle the applications of polynomials we will look at in the next section.

3.3 Using Polynomials

We have covered just about everything we are going to on the subject of polynomials. You know what they are, what operations are defined for them, and what they look like when graphed. The next step, and the reason we studied all of this material to begin with, is applying polynomials to the real-world problems that game developers face.

Polynomials can be used in a variety of ways -- too many to cover in one lesson's material. So what we will do in the next few sections is select a few of the more interesting applications. But by the time you are done, you will be familiar enough with the process to apply polynomials to anything you want.

3.3.1 Modeling Phenomena with Polynomials

In our last chapter we used exponential and log functions to model various phenomena. Exponential functions work great when you need a steep increase or decrease, and log functions work well if you need a slow increase or decrease (they can also be used to solve exponential equations, as we saw). Some phenomena however, exhibit more complex behavior. For example, let us say that we want to model the intensity of rain in a storm, which starts out softly, then rises to full intensity, and then decreases again to zero. Neither log nor exponential functions can help us here, but polynomial functions can.

Polynomial functions are ideal for modeling phenomena that rise, and then fall (or visa versa). They can also be used to model phenomena that rise, level out, then rise some more (or phenomena that fall, level out, then fall some more). Polynomial functions can do more complex things too, like rise, then level out, then rise some more, then level out, then fall. The possibilities are truly endless, although the more complicated the polynomial, the longer it will take for your computer to process it – there is a point at which it simply is not feasible to model something using a polynomial.

That said, let us take a look at a fairly simple application of polynomials: modeling a flashing light. A flashing light starts out at zero intensity, increases to max intensity, and then decreases to zero intensity again.

It makes sense to have our function map from the time since the start of the flashing cycle to the intensity of the light (say, from 0 to 1, where 0 represents no intensity, and 1 represents full intensity). A rough graph of such a function is shown in Figure 3.5.

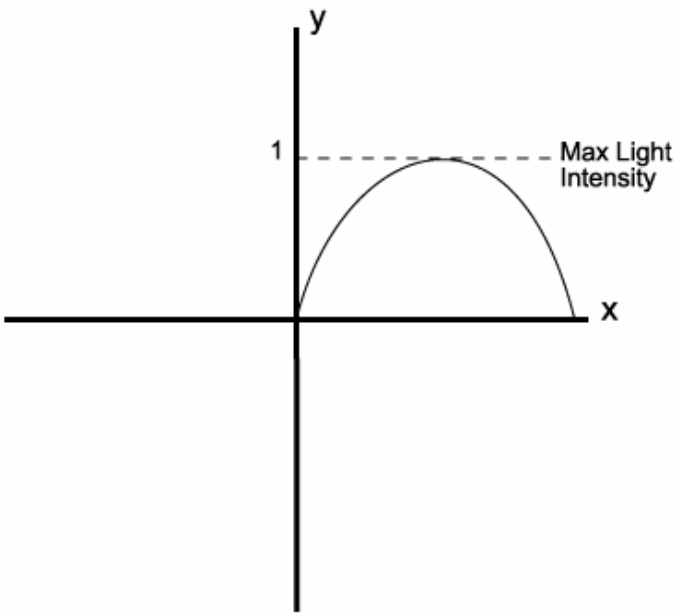


Figure 3.5: The graph of a light intensity function.

The first and most important step to solving this problem is recognizing that we need a quadratic (trinomial) function, (i.e. a function of the form, $f(t) = at^2 + bt + c$). Here *t* is the elapsed time from the start of the flashing cycle, $f(t)$ is the intensity of the light, and *a*, *b*, and *c* are constants. Now we need to choose constraints so we can determine what the values of the constants are.

We will choose the intensity of the light to be zero at $t = 0$, increase to 1 at $t = 0.5$, and then decrease to 0 again at $t = 1$. Now a fact about quadratics that should be fairly obvious from their graphs is that if the graph of one intersects the *x-axis* twice (that is, if the function sends two elements in the domain to 0), then exactly in the center of these intersection points is the *x* value that corresponds to the maximum or minimum of the function (see Figure 3.6). What this means is that since we have specified that the function will intersect the *x*-axis at $t = 0$ and $t = 1$, the maximum or minimum of the function occurs at $t = 0.5$. That is why we said that the function should increase to 1 (our specified maximum intensity) at $t = 0.5$.

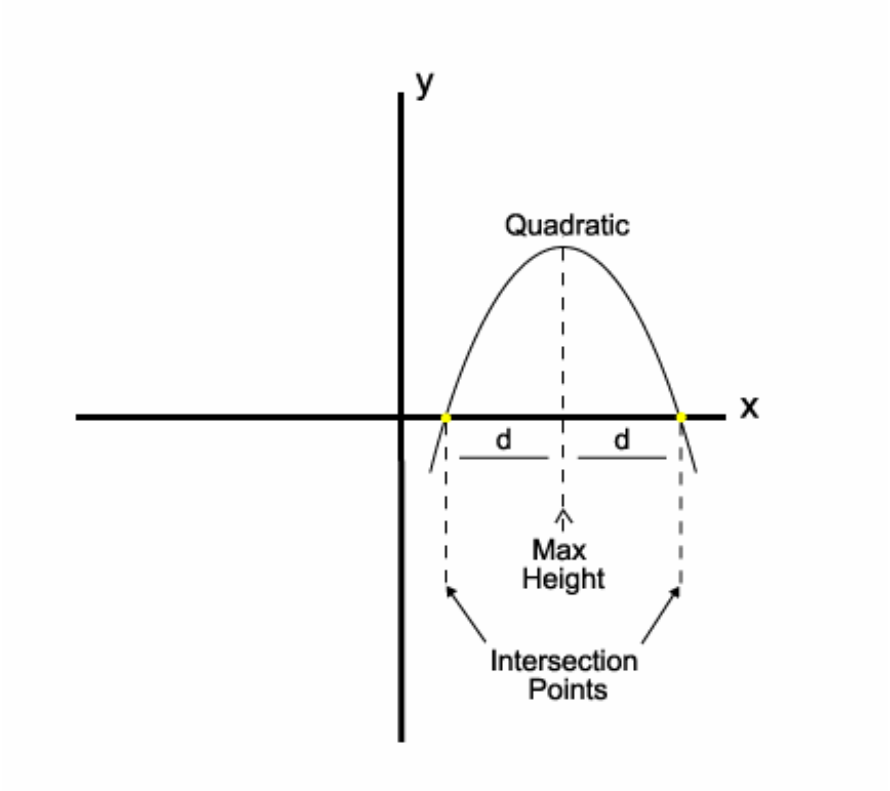


Figure 3.6: The minimum or maximum of quadratic functions.

All of the constraints we have just made up force the function to satisfy the following equalities:

$f(0) = 0$
$f(0.5) = 1$
$f(1) = 0$

These equalities in turn give us the following equations:

1. $a(0)^2 + b(0) + c = c = 0$
2. $a(0.5)^2 + b(0.5) + c = 1$
3. $a(1)^2 + b(1) + c = a + b + c = 0$

Equation (1) tells us that $c = 0$. Substituting this value for *c* into equations (2) and (3), we get the following two derived equations:

4. $a(0.5)^2 + b(0.5) = 1$
5. $a + b = 0$

Equation (5) tells us that $a = -b$. Plugging this into equation (4), we get the following equation:

$-b(0.5)^2 + b(0.5) = b\{0.5 - (0.5)^2\} = b(0.25) = 1$

Thus we see that $b = 1/(0.25) = 4$, and hence, that $a = -4$ (by equation (5)). Our final equation for *f*, our function, is therefore as follows:

$f(t) = -4t^2 + 4t$

Notice that the coefficient of t^2 is negative, which means the graph of the function will look like an upside down 'U' -- exactly what we wanted. You can see the graph for all *t* in between 0 and 1 in Figure 3.6.

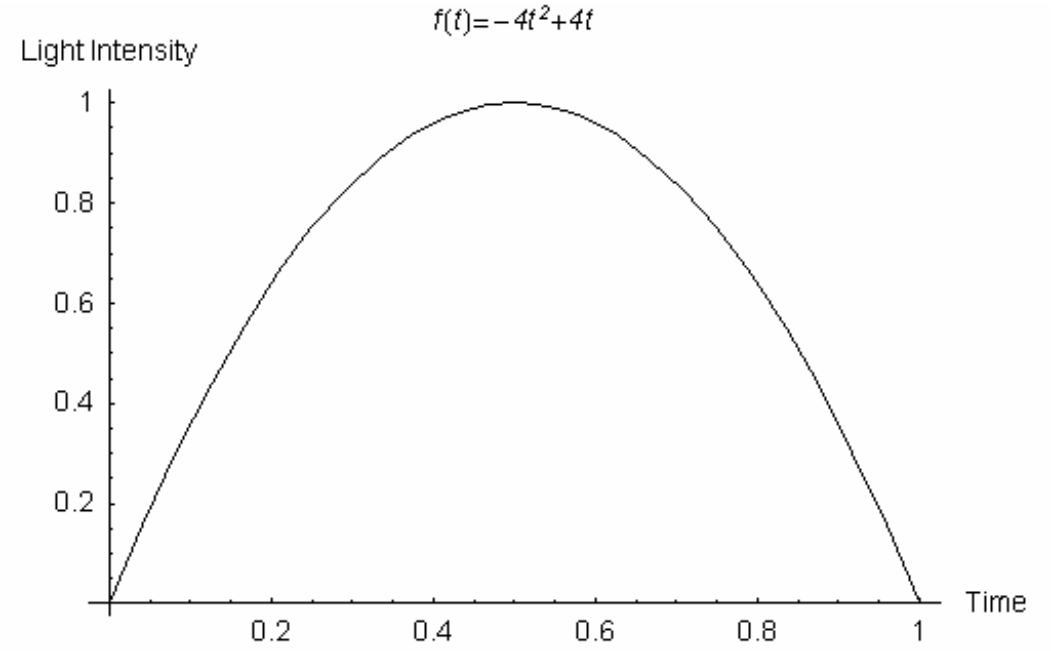


Figure 3.7: The graph of the function $f(t) = -4t^2 + 4t$.

So the process of using a polynomial to model something is not very complex, at least for low degree polynomials. It does get more complex for higher degree ones, but even then, all you have to do is solve

for the unknown constants like we have just done here. To do this, you will want to impose *n* constraints on your polynomial, where *n* is the degree of the polynomial. This will produce *n* equations. Solve one equation for one unknown, and substitute it into the other equations. Repeat the process until you have figured out what all the unknowns are.

3.3.2 Linear Interpolation

Another popular application of polynomials is *linear interpolation*. Linear interpolation refers to passing a variable through a range of values in a linear fashion. That is, we use a binomial function, whose graph is a line, to do it.

Say, for example, that you are manually drawing and filling a polygon on a computer screen. One way to do this is to find the top of the polygon, and then travel down the polygon one row at a time, filling whatever is inside the left and right edges of the polygon. The essential task here is finding out where the left and right edges of the polygon are -- that is, what their *x* (horizontal) components are.

Suppose we are displaying a row -- call it *y*. We need to figure out where the edges of the polygon are on this row. The edges of the polygon are defined by lines, so our task is going to involve lines. Specifically, if we could figure out the *x* coordinate of the line at row *y*, we could solve our problem.

This may seem a bit abstract, so take a look at Figure 3.8 to see what we are talking about.

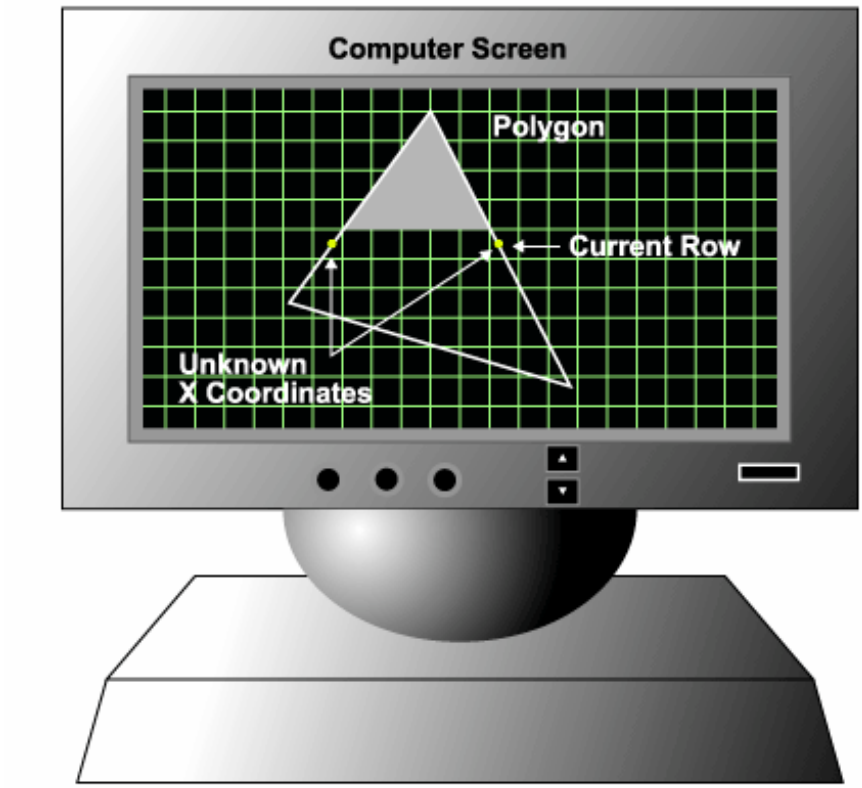


Figure 3.8: Displaying a row of a polygon.

Binomials can easily solve such a problem. The generic form for a binomial function is $f(y) = by + c$. Each edge of the polygon must go through the two points that define that edge. If we call these points (x_1, y_1) and (x_2, y_2), then this imposes the following constraints on the binomial function:

$f(y_1) = x_1$
$f(y_2) = x_2$

These constraints give us the following two equations:

$x_1 = by_1 + c$
$x_2 = by_2 + c$

Solving the first equation for c, we find that $c = y_1 - bx_1$. Substituting this into the second equation, and solving for b, we find that $b = (x_2 - x_1)/(y_2 - y_1)$. Now we can feed this into our equation $c = x_1 - by_1$, which then becomes $c = x_1 - y_1(x_2 - x_1)/(y_2 - y_1)$. Our final expression for the binomial expression is therefore as follows:

$$f(y) = y(x_2 - x_1)/(y_2 - y_1) + x_1 - y_1(x_2 - x_1)/(y_2 - y_1)$$

This gives us the x component for any given y.

Drawing polygons is not the only thing interpolation can be used for. We might associate a color with each vertex of our polygon, and then want to interpolate these colors down the rows of the polygon and then across the columns, displaying each point on the polygon with the interpolated color. This process, known as *Gouraud shading*, would actually require that we interpolate the red, green, blue, and alpha components of the color separately (red, green, and blue together define all possible colors the human eye can perceive, and the alpha component defines how transparent the color is). You could also associate an ordered pair with each vertex, and then interpolate the components of this pair separately, again down rows and across columns, and then look into an image and grab a point from there that corresponds to the interpolated point. This technique is known as *linear texture mapping*, and it was used to increase realism before better methods of texture mapping became practical. (See the Graphics Programming course series for more information on shading and texture mapping.)

There are numerous other applications for linear interpolation. The technique is so easy to apply and so fast, in fact, that it has been applied a bit *too* widely. Most things in real-life are not linear. Drawing polygons, shading, and texture mapping are operations where linear interpolation works, but that is not the most realistic way to do these things. But nonetheless, linear interpolation is an excellent choice whenever it does a good job approximating the underlying phenomena, or when the alternatives are just too slow.

3.3.3 Approximating Functions

Once in a while you will cook up a mathematical function (or perhaps find it in some book on graphics, physics, mathematics, or game development) that does exactly what you need, but when you put it in your game, you find that it slows the application down way too much -- either because the function is

really complex, or because your program uses it quite frequently. Sometimes you can solve this problem by using a polynomial to *approximate* the function. By approximate, we mean that the graph of the polynomial resembles the graph of the function over some region of interest.

Unfortunately, the optimal method for producing such polynomials requires an understanding of calculus, which is not assumed in this course. However, there is a 'cheat' that often produces good results. This method involves calculating a couple of points on the graph of the original function, and then generating a polynomial that passes through these points. The good news is that this technique does not involve learning any concepts we have not already covered.

We will look at one example to see how it can be done.

Suppose we come up with the following function, whose domain is the set of all positive real numbers:

$$f(x) = \left(\frac{e}{\ln(x+1)} \right)^{x+1}$$

This is a pretty complex function, as you can tell by looking at Figure 3.9, and it is going to take a while for the computer to process it.

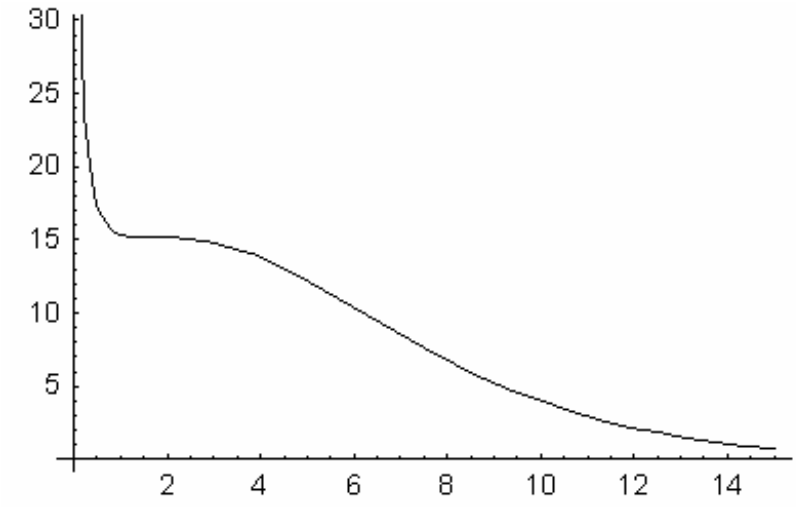


Figure 3.9: The graph of the function $f(x) = \left(\frac{e}{\ln(x+1)} \right)^{x+1}$.

Now let us take a look at how we could code this function in C/C++:

```
float f ( float x )
{
   return pow ( 2.71828F / log ( 1.0F + x ), 1.0F + x );
}
```

The code involves two function calls, each of which is very slow, plus a division and an addition (addition is quick, but division is not). If you did not need to use this function very often, then it would

probably not be much of a problem, but if you had to call it hundreds of times a second, it could slow down your program considerably.

We cannot approximate the entire function (especially not without calculus), but we can approximate a section of it with a low degree polynomial. This may not sound terribly useful, but it turns out that more often than not, we will only be interested in a small region of a function anyway. For example, if you are using a function to simulate the swing of a grandfather clock pendulum, you really only need to concern yourself with low angles. Similarly, instead of using a realistic but cumbersome lighting function for every point you draw on the computer screen, you can calculate the true lighting for just a few points, and then use a polynomial to approximate the lighting for the in-between areas; in this way, you need approximate only a small subset of the range of the lighting function.

For this example, let us assume that we are interested in all x values in between 2 and 4. A blown-up section of the function over this region is shown in Figure 3.10.

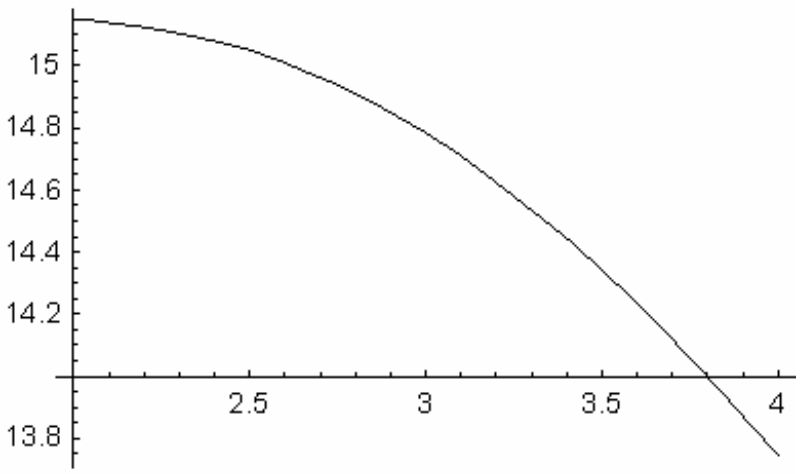


Figure 3.10: A magnified view of the relevant portion of the function to be approximated.

This portion of the function looks remarkably similar to a quadratic, so we will define our polynomial function to be $g(x) = ax^2 + bx + c$.

We want our polynomial to approximate the region from $x = 2$ to $x = 4$. One way to do this is to make sure the points (2, f(2)), (3, f(3)) and (4, f(4)) are on the polynomial, since these same points are also on the function we are approximating. This forces our polynomial to satisfy the following equalities:

$$g(2) = f(2)$$
$$g(3) = f(3)$$
$$g(4) = f(4)$$

If you go through the math (we will avoid the gory details here), then you end up getting the following expression for the function g:

$$g(x) = -0.337058x^2 + 1.32025x + 13.8555$$

This function is graphed in Figure 3.11. The polynomial approximates the function *f* amazingly well over the interval we are interested in. In fact, if you were to graph both of them together, you would not be able to tell the difference between them -- that is how good the approximation is.

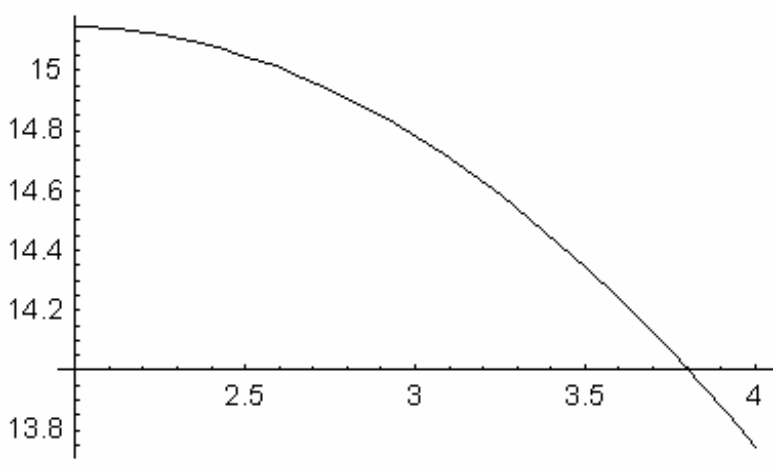


Figure 3.11: The graph of the approximation function.

The polynomial function requires only a few multiplications and additions, which is a tremendous savings. And since the approximation is nearly flawless over the specified interval, there would be no reason not to use the polynomial instead.

Of course, polynomials will not always save you. Sometimes you will be interested in so wide a region or so complicated a function that a low degree polynomial will not be of any service. Higher degree polynomials can approximate most any smooth function very well over most any interval, but at great cost: all of those powers of *x* take time to compute, possibly even more time than the function that you are trying to approximate takes. The lesson to learn here is that polynomials are useful when approximating functions, but only when you can do so using a low degree polynomial.

3.3.4 Predicting the Future

Another way to use polynomials is for predicting the next number in a sequence of numbers -- such a polynomial is called a *predictor*.

There are many ways you can use predictors in games. Say, for example, you are writing a multiplayer racing car game. Your computer needs to know where everyone else is and what they are doing, but due to network latency problems, sometimes your computer will not receive any information about where the other players are for a long time (anywhere from a half a second to several seconds). During this period of time, what should you do? You could stop their cars at their last known positions, and then start them up again the next time you receive information, but this would make the cars jump from position to position, which is not very realistic. A much better way is to look at what the cars were doing the last time you received information, and then *guess* what they are doing now -- when you receive the next update, you can always correct any errors in your guess.

You can also use predictors in artificial intelligence for games. In action games, the player's enemies often have weapons (such as crossbows or plasma guns) that shoot out projectiles that take time to travel from the enemies to the player. If the computer just aimed at the player and fired, then whenever the player was moving, the projectile would miss. The solution to this problem is to anticipate where the player will be and then aim for *that* location instead.

Implementing a predictor is very straightforward: decide how many past numbers you want to analyze and then create a polynomial with that many terms (if you wanted to analyze 2 numbers, for example, then you would create a binomial). Then create a series of points based on the numbers you record; for example, if the recorded numbers were labeled a_0, a_1, ..., a_n (where *n* is one more than the degree of the polynomial), then you could generate the points $(0, a_0)$, $(1, a_1)$, ..., (n, a_n). Lastly, force the polynomial to contain these points, and solve for the unknown coefficients. To predict the next number in the sequence, just evaluate the polynomial at the next *x* value -- in our setup, $n+1$.

You will not need to write code to implement these ideas, since it has already been done for you as part of the Game Math SDK included with this course. To use the source code, just add eIMathLib.cpp to your project and include the header eIMathLib.h in your files.

The class that implements prediction is called **eIPredictor**. This class has just three member functions: **AnalyzeValue()**, **PredictValue()**, and **Reset()**.

The function **AnalyzeValue()** stores a real number in memory for use in predicting future values. A maximum of three values are stored. If you have the function analyze more values, then the oldest of the values stored in memory is tossed out.

The function **PredictValue()** uses the stored real numbers to predict the next number. Exactly how it does this depends on how many values have already been stored. If one value has been stored, the function returns that value. If two values have been stored, then a linear predictor is used to predict the next value. If three values have been stored, then a quadratic predictor is used to predict the next value.

The function **Reset()** clears all the stored numbers.

The following code example shows you how you might use this class:

```
eIPredictor Pred;
Pred.AnalyzeValue ( 1.0F );
float Value = Pred.PredictValue (); // Value is now equal to 1.0F
Pred.AnalyzeValue ( 2.0F );
Value = Pred.PredictValue ();       // Value is now equal to 3.0F
Pred.AnalyzeValue ( 1.0F );
Value = Pred.PredictValue ();       // Value is now equal to -2.0F
```

If you were using the predictor in a multiplayer game to predict the new position of players, then you might want to use the **Reset()** function every time you received an information update. (Note that positions require two to three numbers to represent, depending on the dimension of the coordinate system, so predicting a position would require two to three predictors.)

3.3.5 Using Polynomials in Code

To make polynomials as easy to incorporate into your games as possible, a simple C++ math source code library is included with this course. As with the **eIPredictor** class, to use this source code, simply add eIMathLib.cpp to your project and include the header eIMathLib.h in your files.

The source code hides the complexity of using polynomials in the **eIPolynomial** class, which represents a polynomial. You can set and retrieve the degree and coefficients of polynomials, evaluate a polynomial for a given real number, add and subtract polynomials, multiply and divide polynomials by real numbers, compare polynomials, and create a polynomial that passes through a series of points.

You can create an **eIPolynomial** object as you would any other object, as shown with the following code snippet:

```
eIPolynomial MyPoly;
```

If you want to specify the degree of the polynomial when you create the object, you can use the alternate **eIPolynomial** constructor, which accepts an integer (the degree of the polynomial). The following code creates an **eIPolynomial** object of degree 2.

```
eIPolynomial Quadratic ( 2 );
```

If you do not use this constructor, then before you can use the other member functions of **eIPolynomial**, you will need to set the degree of the polynomial with the **SetDegree ()** member function. You can use this function on any polynomial object, even if its degree has already been set.

The following code sets the degree of a polynomial object to 4:

```
MyPoly.SetDegree ( 4 );
```

To set and retrieve the coefficients of an **eIPolynomial**, simply call the **SetCoeff()** and **GetCoeff()** member functions. The first parameter of these functions is the power of *x* of the term you are interested in. The second parameter of **SetCoeff()** is the new value of the coefficient of that term (**GetCoeff()** does not accept any more parameters).

The following code example creates a second-degree polynomial and then sets its coefficients to that of the polynomial x^2.

```
eIPolynomial Poly ( 2 );
Poly.SetCoeff ( 0, 0.0F ); // Sets the coeff. for the x^0 term
Poly.SetCoeff ( 1, 0.0F ); // Sets the coeff. for the x^1 term
Poly.SetCoeff ( 2, 1.0F ); // Sets the coeff. for the x^2 term
```

Adding, subtracting, and multiplying or dividing polynomials by scalars are performed using the standard C++ operators for these functions. You can compare polynomials with the **==** and **!=** operators.

To evaluate the polynomial for a given real number, use the **Evaluate ()** function as shown below:

```
ElementInRange = Poly.Evaluate ( 1.0 );
```

The last feature of the **eIPolynomial** class is the ability to create a polynomial that passes through a number of points. The function that does this, **Interpolate()**, is a static member function. It accepts a list of points and an integer that specifies how many points are in that list. The list itself is specified as a series of **eIVector2D** classes, which we will learn more about in Chapter Seven. For now, just think of them as points. The **eIVector2D** class has two member variables you should be familiar with: **x1** and **x2**. The first designates how far the point is along the *x-axis*, and the second designates how far the point is along the *y-axis*.

If you wanted to generate a polynomial that passed through the points in the list **Pts**, and there were **n** of them in that list, then you could do so with the following code:

```
eIPolynomial NewPoly = eIPolynomial::Interpolate ( Pts, n );
```

The method by which this function works its magic is somewhat beyond the scope of this chapter's topic. However, please note that it is a very slow process (especially the more points you use) and it is subject to inaccuracies. In general, you should work out the polynomials by hand whenever you can, rather than relying on this function to generate them for you.

Conclusion

In our next lesson, we will continue our tour of mathematical functions by covering the basics of trigonometry. “Trig”, as it is commonly abbreviated, is largely concerned with the definitions and properties of the so-called *trigonometric functions*. These functions relate angles to the lengths of the sides of a right triangle in ways that spawn a multitude of applications.

Exercises

1. Calculate the following:
   a. $(2x^2 - 4x^3) + (-2x^2 - 2x^3 + 9x)$
   b. $(x + x^2 + x^3) \times (-2 + 3x + 5x^5)$
   c. $(x^3 + 2x - 5) \div (x + 1)$

*2. Suppose you are incorporating rain effects into your game and you want to make the size of a raindrop proportional to the distance between the viewer and the raindrop. If the position of the raindrop is (x, y, z), and the position of the viewer is (v_x, v_y, v_z), then the distance between the two points is $d = \sqrt{(v_x - x)^2 + (v_y - y)^2 + (v_z - z)^2}$. If you let $q = (v_x - x)^2 + (v_y - y)^2 + (v_z - z)^2$, then you can rewrite this equation as $d = \sqrt{q}$ -- a function of a single variable q. Since there are hundreds or thousands of raindrops, and it is computationally expensive to evaluate the square root function, it makes sense to approximate the square root function with a polynomial. Develop a quadratic approximation over the interval [0, 10] (on the assumption that raindrops further away than 10 units will not be visible at all). (Hint: You will have to use the quadratic formula to solve for the constants, unless you have a calculator or software that can solve the equations for you.)

An example of this kind of approximation is shown in Figure E3.1.

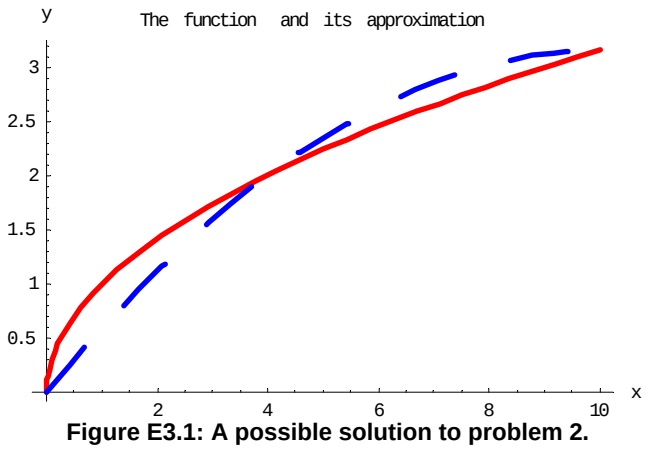


Figure E3.1: A possible solution to problem 2.

3. Find the general form of a monomial that passes through the point (x, y).

4. Find the general form of a binomial that passes through the points (x_0, y_0) and (x_1, y_1).

*5. Texture mapping is the process of mapping a bitmap onto a polygon. You are given a two-dimensional polygon described by n points (ordered pairs). With each point, you are also given another ordered pair that specifies a location on a bitmap corresponding to that point (see Figure E3.2). In order to map the texture onto the polygon, as you display each pixel of the polygon you must find out which location in the bitmap corresponds to that pixel. Describe how you might use binomials to solve this problem.

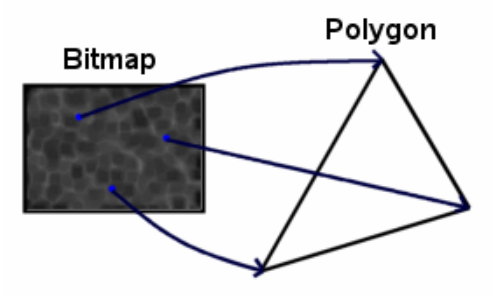


Figure E3.2: Linear texture mapping.

*6. Suppose the angle between a surface and a light source is denoted by *x*, measured in degrees, from 0 to 180 (see Figure E3.3). Develop a quadratic function to model the intensity of the surface from 0 to 1 (assuming that at *x* = 90 degrees, the surface is at its brightest, and at *x* = 0 = 180 degrees, the surface is at its dimmest).

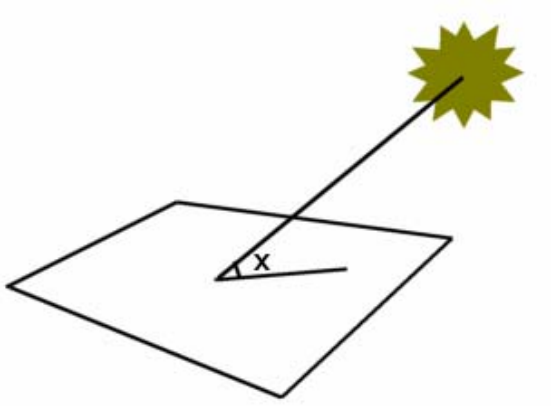


Figure E3.3: The angle between a surface and a light source.

*7. A product of *n* terms can only be zero if one of the terms is zero. Thus, the roots of the 4th degree polynomial $(x+2)(x+1)(x-1)(x-2)$ are at *x* = -2, -1, 1, and 2. Create a polynomial whose roots are at *x* = 0, 1, 2, and 3, and then scale the polynomial so that its maximum is 2. (Hint: The scaling factor will be negative.) The end result is shown in Figure E3.4.

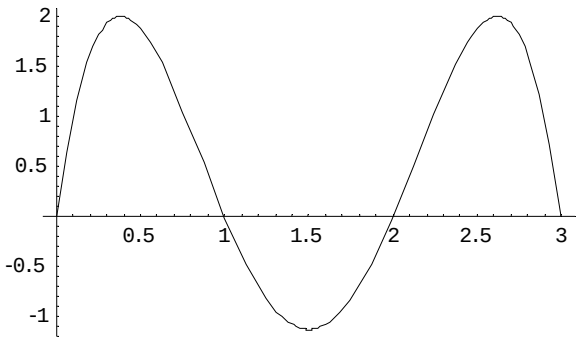
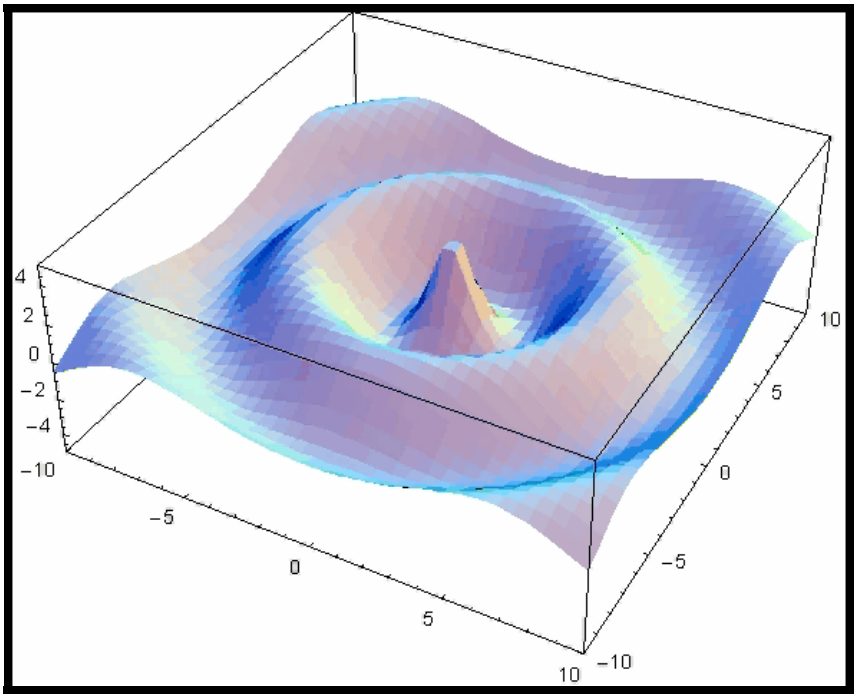


Figure E3.4: The solution to problem 7.

Chapter Four

Basic Trigonometry I



Introduction

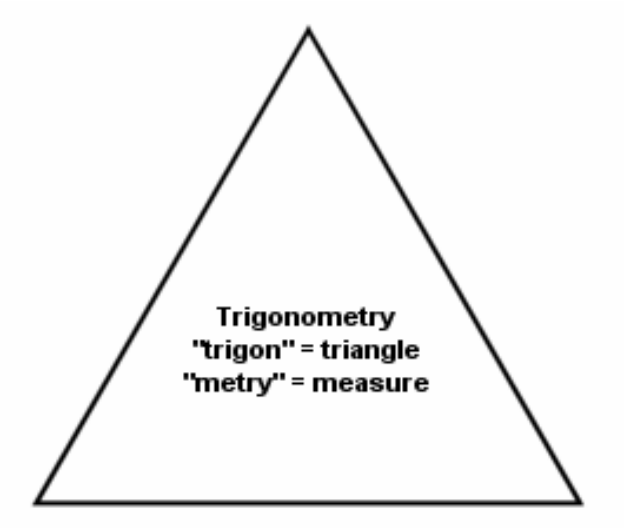


Figure 4.1: A triangle.

Trigonometry is a subject whose roots go all the way back to Hipparchus, a Greek philosopher who lived in the 2nd century BC, although its foundations go back even further, to the heyday of the Babylonians. From those early times to the Swiss mathematician Leonhard Euler's work in the 18th century, trigonometry has continued to evolve and grow into a subject of considerable depth and richness.

The reason for humanity's special preoccupation with trigonometry is that it is eminently applicable to real-world problems. The Greeks used it for surveying, navigation, and astronomy. Modern physicists use it in virtually all of their work, from astrophysics to quantum physics. And game developers use it to manipulate and display 3D geometry, to perform physics calculations, and to optimize their games for blazing fast performance.

Trigonometry is often one of the most misunderstood of all subjects. Even students who have gone through trigonometry in high school are often baffled by what they covered, and can barely recall only its most basic principles years later (perhaps due to the emotional trauma of studying it!).

But trigonometry need not be so intimidating or difficult to master. At its heart are some very simple ideas that involve one of the simplest shapes of all -- the triangle. In this chapter, we will introduce (or perhaps *re*introduce) the subject of trigonometry and cover it in a slow, methodical manner so that you will be on top every step of the way. When you are done, your skills as a game developer will have increased *exponentially* (to borrow a term from lesson two!).

4.1 Angles

Central to trigonometry is the concept of *angles*. Formally, an angle is a real number that measures the amount of divergence between two rays that share a common origin.

You have probably used angles before back in your high school geometry lessons. Unfortunately, the system you learned there was likely based on degrees (e.g. 360 degrees in a complete revolution), and that system has been out of fashion with mathematicians for quite some time.

In this course we will use the concept of *radian measure*, which is a far more elegant and useful way to express angles.

Radian measure relies on a circle of radius 1 (that is, the distance from the center of the circle to its perimeter is 1 unit) centered on the origin of a Cartesian coordinate system. Suppose we draw two rays that emanate from the origin. Then the angle between the rays is the length of the arc delineated by the intersection of the rays with the perimeter of the circle. If this makes no sense to you, check out Figure 4.2, because -- as they say -- a picture is worth a thousand words.

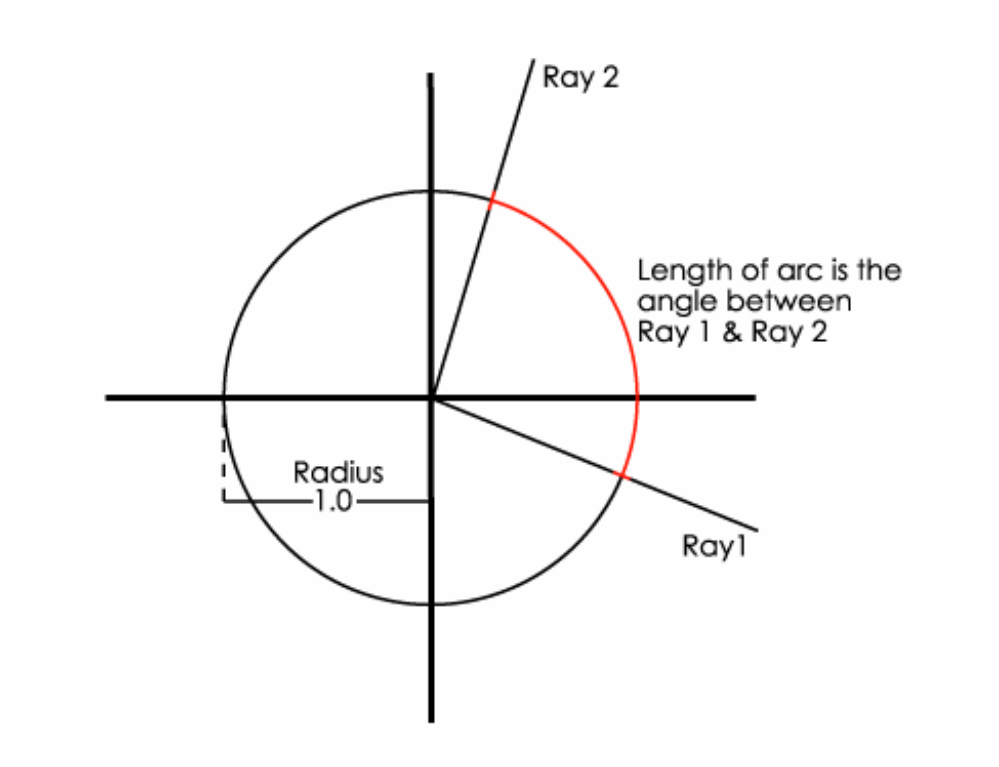


Figure 4.2: The definition of an angle.

Figure 4.2 depicts a circle of radius 1 (a *unit circle*) centered on the origin, and two rays that emanate from the origin. The two rays intersect the perimeter of the circle at points *A* and *B*, respectively. The angle θ is simply the length of the arc from *A* to *B* -- i.e., the distance you would travel if you walked the perimeter of the circle from *A* to *B*. There is only one caveat: you can actually draw your arc (or

travel from *A* to *B*) in an infinite many ways by going around the circle repeatedly, or in a different direction. This means that you can measure any given degree of divergence in infinitely many ways.

Angles are usually designated with lower-case Greek letters, the most common letter being θ (theta - pronounced "THAY-tuh"). They are identified by the word "radian" (plural: radians), so if you see a measurement such as "2.31 radians", you know the number is an angle. This is where the term *radian measure* gets its name.

4.1.1 Common Angles

Suppose we wanted to rotate a ray all the way around the perimeter of a circle and back to its starting point. By how many radians would we have to rotate the ray to do this? You may recall from high school the formula $p = 2\pi r$, which relates the perimeter p of a circle to its radius r. The perimeter of a circle, as you probably remember, is just the distance you would travel if you walked all the way around it once.

The radius of the measuring circle is 1, so we have $p = 2\pi(1) = 2\pi$ radians. Rotate a ray by this many radians, and it is back where it started. Of course, you could also rotate it by 0, -2π, or 4π radians to get the same effect (try to think of a few more angles that would do this).

Figure 4.3 shows this and some other angles that arise often in trigonometry.

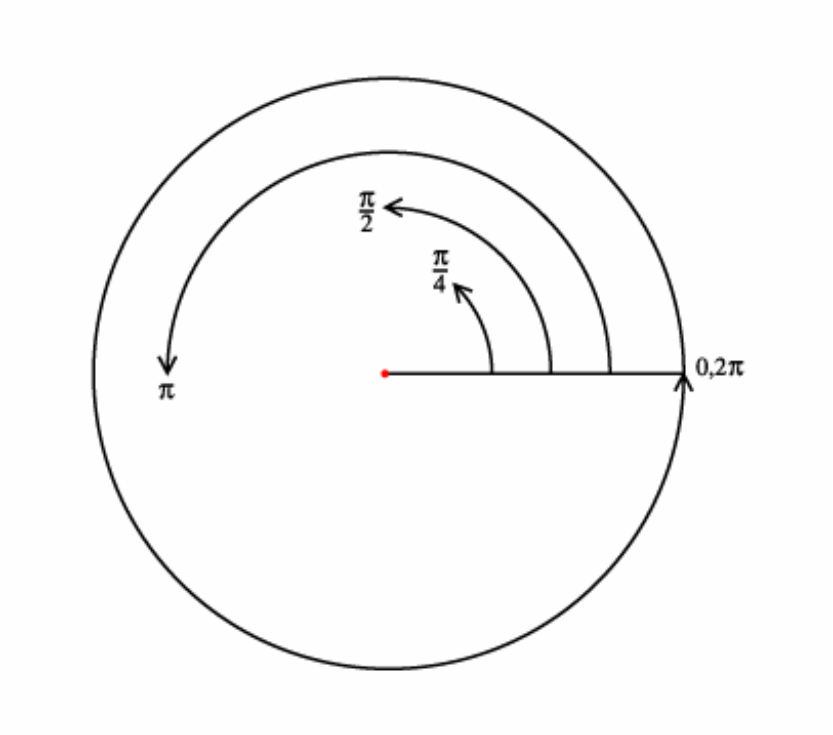


Figure 4.3: Common angles.

The angle $\pi / 2$ (which corresponds to 90 degrees) is called a *right angle*. Any angle less than this angle is classified as an *acute angle*, and any angle greater than this is classified as an *obtuse angle*.

You can use angles to measure the divergence of more than just rays, too, or else they'd be pretty useless. For example, you can measure the angle of a ramp by using two rays, one parallel to the ramp and the other parallel to the ground. The concept of rays only gives the definition of angles precision.

4.1.2 The Polar Coordinate System

The concept of the angle allows mathematicians to use a different coordinate system. It is different from, although similar to, the Cartesian system we studied earlier. It is called a *polar coordinate system*. Under this system, points are still measured by ordered pairs, but the interpretation of the numbers has changed. The first number of the ordered pair specifies the distance from the origin of the coordinate system to the point. The second number of the ordered pair tells you the angle a ray drawn from the origin through the point makes with the positive *x-axis*. By convention, positive angles indicate a counterclockwise measure with respect to the *x-axis*, and negative angles indicate a clockwise measure from the positive *x-axis*.

You can see a few points plotted in the polar coordinate system in Figure 4.4.

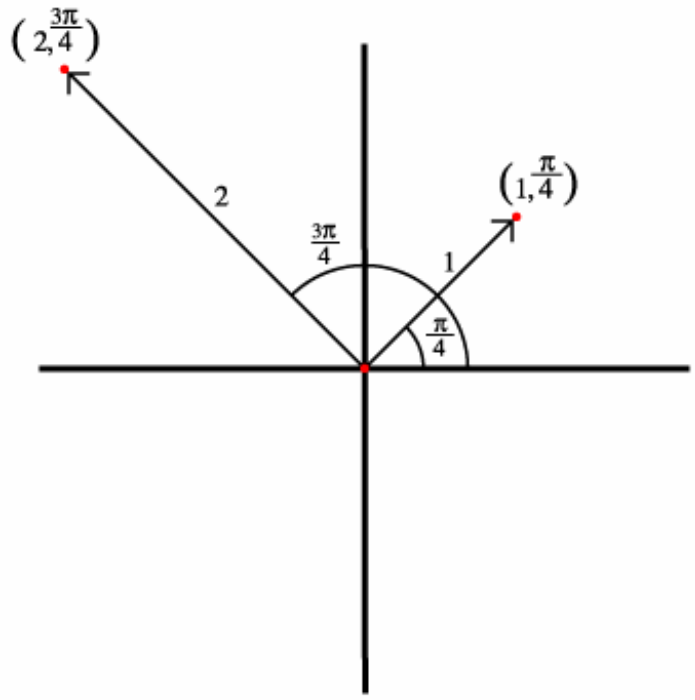


Figure 4.4: Points in the polar coordinate system.

Note that in the polar coordinate system, unlike the Cartesian coordinate system, each point does not have a unique representation.

Although we will not be using it in this course, there is a three-dimensional analog to the polar coordinate system called the *spherical coordinate system*. In this system, each point is specified by an ordered 3-tuple: the first number of the 3-tuple describes the distance from the origin to the point (as with polar coordinates). The second number tells you the rotation of the point around the *y*-axis from the positive *x*-axis. The third number tells you the angle the point makes with the positive *y*-axis.

4.2 The Triangle

The triangle is a geometric primitive (building block) that consists of all the points in the closed region formed by the intersection of three non-parallel lines. The angles between these lines are referred to as the angles of the triangle (see Figure 4.5). If the vertices (i.e. corner points) of the triangle are labeled, then these letters also symbolically represent the angles of the triangle.

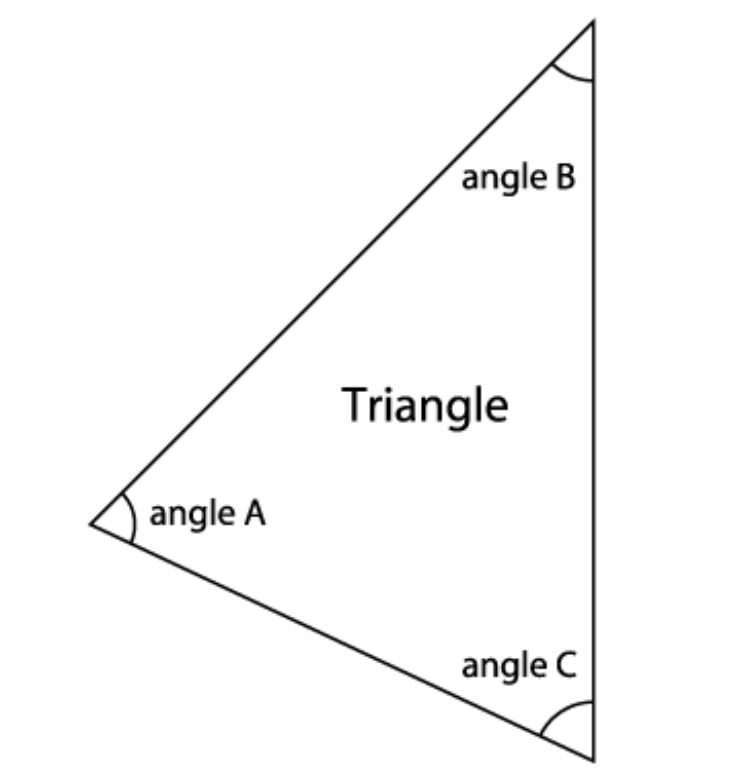


Figure 4.5: An example of a triangle.

Triangles come in a variety of different flavors: an acute triangle is a triangle whose angles are all acute. An obtuse triangle is a triangle that has an angle that is obtuse. An equiangular triangle is a triangle whose angles are equal. A right triangle is a triangle that has a right angle. Figure 4.6 depicts all of the different types of triangles.

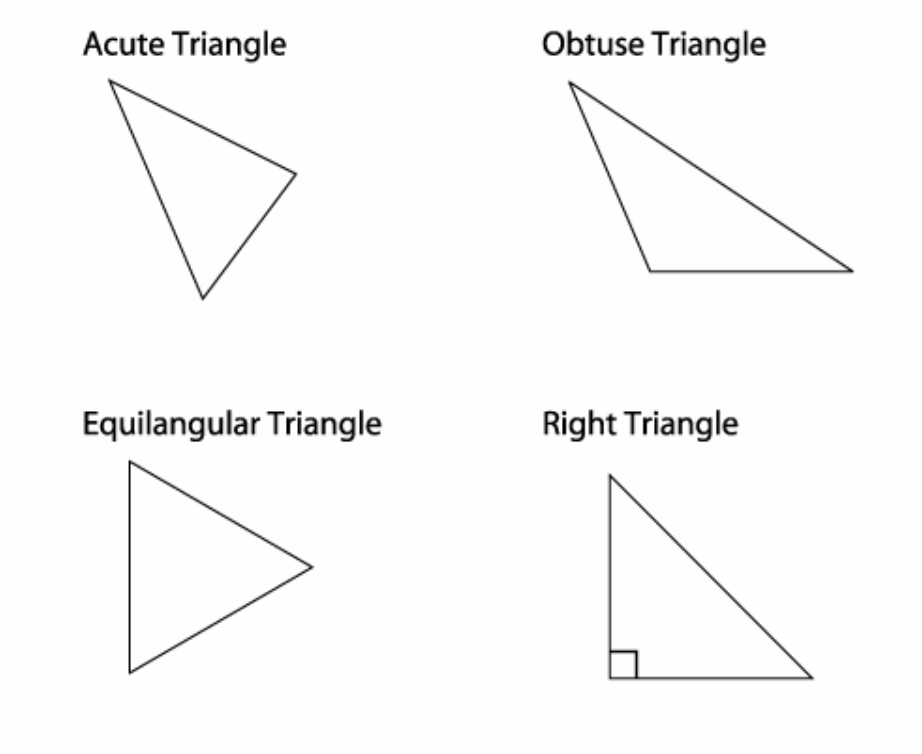


Figure 4.6: The types of triangles and their labels.

4.2.1 Properties of Triangles

The sum of the angles of any triangle is always equal to π radians -- no exceptions. The proof behind this theorem is quite trivial, more of an observation than a proof, as you can see from looking at Figure 4.7.

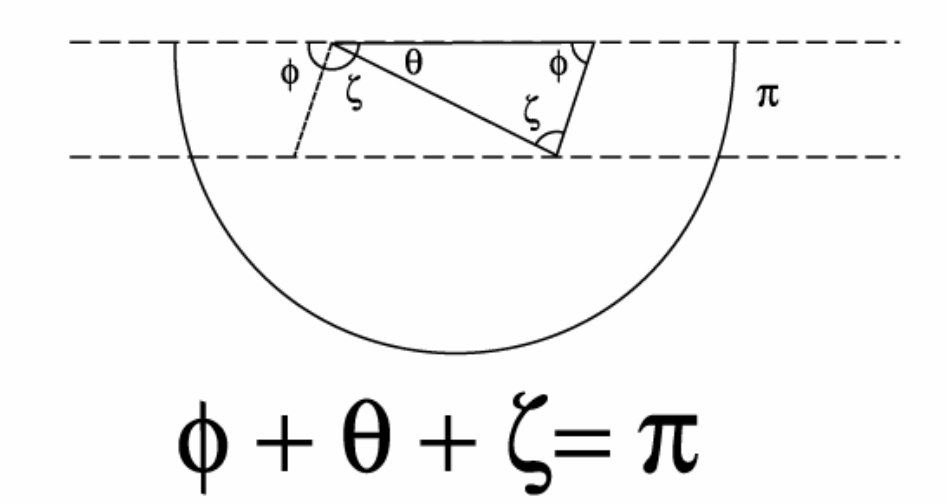


Figure 4.7: Proof that the sum of the angles of any triangle is equal to π radians.

An important concept in trigonometry is that of *similar triangles*. Two triangles are regarded as similar if their corresponding angles are equal, as shown in Figure 4.8.

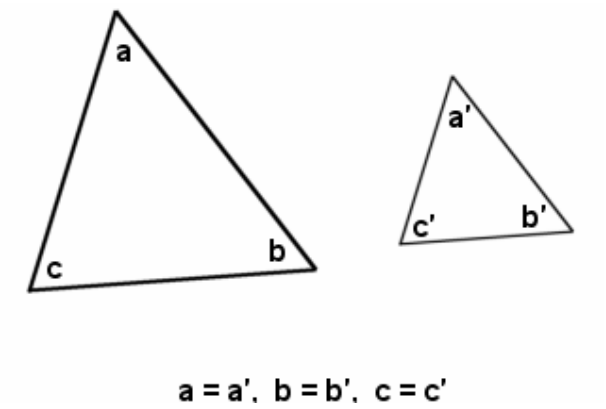


Figure 4.8: The corresponding angles of similar triangles are equal.

Similar triangles have a very interesting property. Say you label the sides of a triangle a_1, a_2, and a_3, then label the corresponding sides of a similar triangle A_1, A_2, and A_3. It turns out that $a_i / a_j = A_i / A_j$, for all i and $j \in \{1,2,3\}$. In English, this says that the ratio of two sides of a triangle is equal to the ratio of the corresponding sides of a similar triangle. Check out Figure 4.9 to see what this looks like.

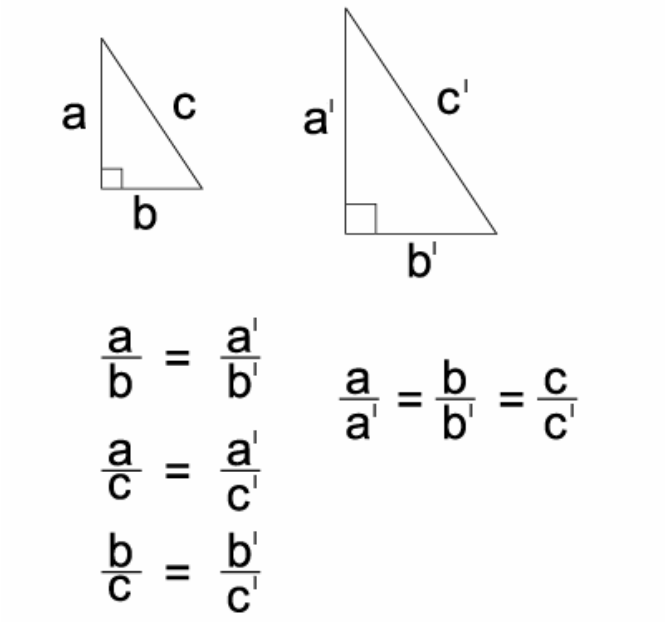


Figure 4.9: Similar triangles.

A consequence of this property of triangles is that a triangle can be enlarged or reduced to match the dimensions of any similar triangle. To be more precise, there is a certain real number, call it α, such that the length of any given side of the triangle is equal to α times the length of the corresponding side of the similar triangle.

The proof of this is straightforward: first, we label the sides of the triangle a_1, a_2, and a_3, and then label the corresponding sides of the similar triangle A_1, A_2, and A_3, as before. Second, we solve the equation $a_i / a_j = A_i / A_j$ for a side of the triangle, say a_i, giving us $a_i = a_j A_i / A_j$. We want $a_i = \alpha A_i$, so substituting this value into our equation for a_i, we find that $\alpha A_i = a_j A_i / A_j$. Solving this equation for α, we see that $\alpha = a_j / A_j$.

We have found an expression for α that has the properties we want, but we are not quite done yet. It is conceivable that a_j / A_j is not constant -- that is, the ratio changes depending on what value we use for *j*. This would be unfortunate, since it would mean we really had *three different versions of* α. However, we are trying to prove there is a *single* real number that satisfies our properties, so this simply will not do. The solution? Prove that a_j / A_j *is* constant and that no matter what value we use for *j*, we get the same real number. We can do this by proving that $a_i / A_i = a_j / A_j$, regardless of the choices for *i* or *j*.

How are we going to do this? Easy. Since the one triangle is similar to the other, we know that $a_i / a_j = A_i / A_j$, for any *i* and *j*. Dividing both sides of this equation by A_i, and then multiplying both sides by a_j, we get $a_i / A_i = a_j / A_j$, which is exactly what we wanted. Proof complete!

4.2.2 Right Triangles

The most important triangle for our purposes as game developers is the right triangle.

The side opposite of the right angle is called the *hypotenuse*, and the other two sides are called the *legs* of the triangle. If we are talking about a certain angle, then the leg that is adjacent to the angle is called the *adjacent side*, and the side directly opposite the angle is called the *opposite side*. You can see this naming convention in Figure 4.10.

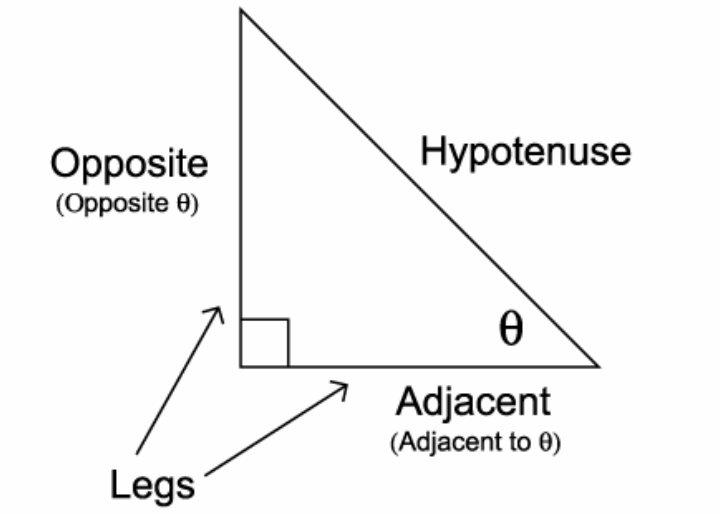


Figure 4.10: The names of the sides of a right triangle.

An ancient theorem states that the square of the length of the hypotenuse is equal to the sum of the squares of the lengths of the legs, for any right triangle. This theorem is called *Pythagorean's theorem*, named after the famous Greek mathematician who first analytically proved it. It is very useful to us because it allows for the calculation of distances between points (among other things).

If the length of the opposite side is *a*, the length of the adjacent side is *b*, and the length of the hypotenuse is *c*, then Pythagorean's theorem can be written symbolically as follows:

$$a^2 + b^2 = c^2$$

4.3 Introduction to Trigonometry

Trigonometry (or *trig*, as it is sometimes called for short) has a whole slew of applications, but one of its most important ones, and the reason it was created in the first place, was to help solve problems involving right triangles. Specifically, if you know a little bit about a right triangle, trigonometry can tell you everything you *do not* know about it.

In this respect, trigonometry is much like a good private detective: you give it some leads, and it will figure out what you want to know. This amazing ability is what makes trigonometry such a powerful tool.

To understand exactly what trigonometry does, consider the right triangle pictured in Figure 4.11.

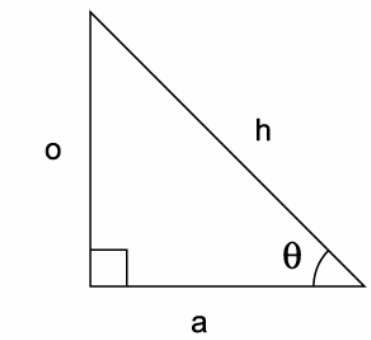


Figure 4.11: A right triangle.

One of the triangle's acute angles is labeled θ, and the length of its sides are labeled *a*, *o*, and *h*, for adjacent, opposite, and hypotenuse, respectively. If we knew θ and the length of one of the sides, could we calculate the lengths of the other two sides and the values of the other two angles? Intuitively, we might say “yes”. After all, we know that the sum of the angles of the triangle must be equal to π radians, and we know that one angle of the right triangle is $\pi/2$ radians, so we know the third angle is just $\pi - \pi/2 - \theta$ radians. Knowing all three angles and the length of one side, we might reason that the lengths of the other two sides are fixed, and that we should be able to calculate what they are.

This intuition turns out to be correct, and it is not terribly hard to see why. Suppose you know an acute angle and the length of one side of a triangle. To figure out what you do not know, first determine the other two angles of the triangle, as described above. Then draw a line segment so that it is as long as the one side you do know, and draw two lines that pass through the end points of this segment and make the appropriate angles with it. These lines will intersect in exactly one place, thus forming a closed region that defines the triangle (Figure 4.12). To figure out the lengths of the other two sides, just measure them.

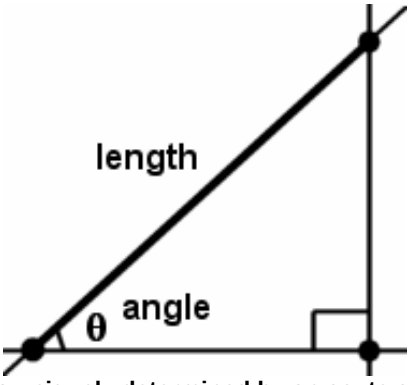


Figure 4.12: A right triangle is uniquely determined by an acute angle and the length of a side.

This recipe is not purely mathematical of course – you would have to use a piece of paper, a pencil, a protractor, and a ruler to compute the results -- but it should help you understand that what we are asking from trigonometry is not what is beyond its capabilities to deliver.

Now that you know something of what trigonometry is all about, it is time to probe a bit deeper. We have already claimed that you can use trigonometry to determine missing information about the angles and lengths of the sides of right triangles. But exactly how do you do this? The answer is found in the *trigonometric functions*.

4.4 The Trigonometric Functions

The three trigonometric functions we are going to introduce in this lesson are called *sine*, *cosine*, and *tangent*, and are denoted with the function names *sin*, *cos*, and *tan*, respectively. All three functions map from the real numbers to the real numbers.

The trigonometric functions are defined with respect to a right triangle. Such a triangle is depicted in Figure 4.13. As before, one of the triangle's acute angles is labeled θ, and the lengths of its sides are labeled *a*, *o*, and *h*, for adjacent, opposite, and hypotenuse, respectively.

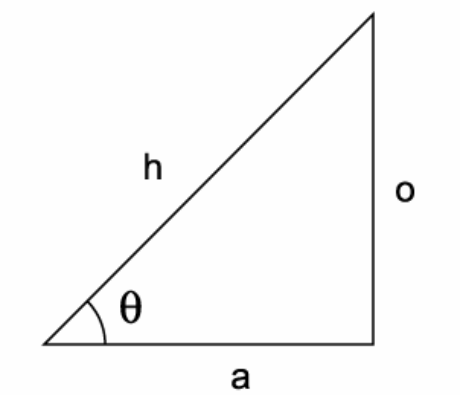


Figure 4.13: The right triangle used to define the trigonometric functions.

With reference to Figure 4.13, the trig functions are defined as follows:

$$\sin(\theta) = \frac{o}{h}$$

$$\cos(\theta) = \frac{a}{h}$$

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} = \frac{o}{a}$$

You will input an angle to the trig functions and they will output a ratio of the sides of the triangle; the nature of the ratio depends on what trig function you are using. For the sine function, the ratio is the length of the opposite side over the length of the hypotenuse. For the cosine function, the ratio is the length of the adjacent side over the length of the hypotenuse. For the tangent function, the ratio is the length of the opposite side over the length of the adjacent side.

With these three trigonometric functions (actually, as you can see from the definition of the tangent function, only sine and cosine are really necessary), you can analytically determine the lengths of the sides of a right triangle knowing only the length of one side and an acute angle. We will look at how to do this shortly, after we cover a more general way to define the trig functions.

4.4.1 An Alternate Definition of the Trigonometric Functions

An alternate way of defining trigonometric functions that proves more useful in many applications involves a circle. This circle is centered at the origin of a Cartesian coordinate system and has radius *r*. Consider a point (*x*, *y*) on the perimeter of the circle in the first quadrant. A ray drawn from the origin to this point makes an angle θ with the positive *x-axis*. This setup is shown in Figure 4.14.

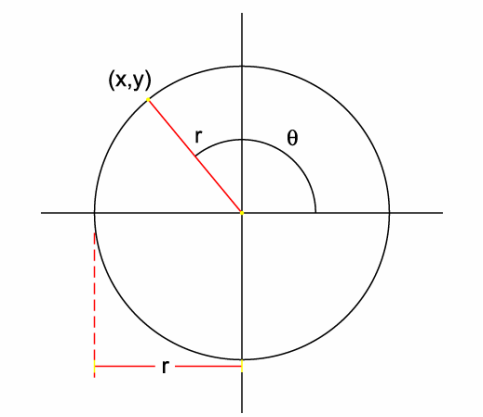


Figure 4.14: The alternate method of defining the trig functions.

With reference to Figure 4.14, the trig functions can be defined as follows:

$$\sin(\theta) = \frac{y}{r}$$

$$\cos(\theta) = \frac{x}{r}$$

$$\tan(\theta) = \frac{\sin(\theta)}{\cos(\theta)} = \frac{y}{x}$$

These equations are, in fact, compatible with the former definitions of the trig functions for all angles in between 0 and $\pi/2$ (which are the only angles that make sense with right triangles). You can see this clearly in Figure 4.15, which shows a right triangle in the first quadrant. The length of the adjacent side is *x*, and the length of the opposite side is *y*. Hence, for any angle in the first quadrant, the above definitions of the trig functions give the same results as the former definitions.

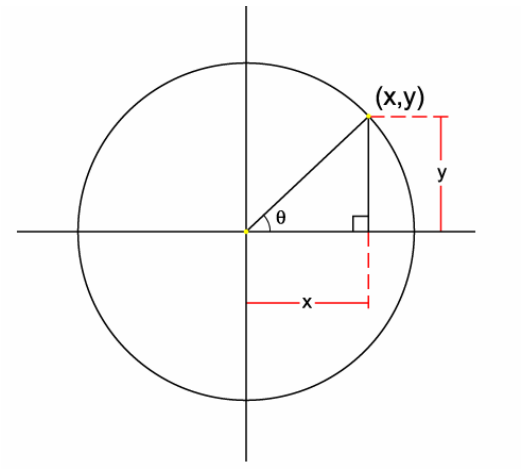


Figure 4.15: A triangle superimposed on a circle.

One thing this alternate definition of the trigonometric functions allows us to do is to convert points from polar coordinates to Cartesian coordinates. If we solve the above definitions of sine and cosine for *x* and *y*, we get the following expressions:

$$y = r\sin(\theta)$$
$$x = r\cos(\theta)$$

This relationship is depicted in Figure 4.16.

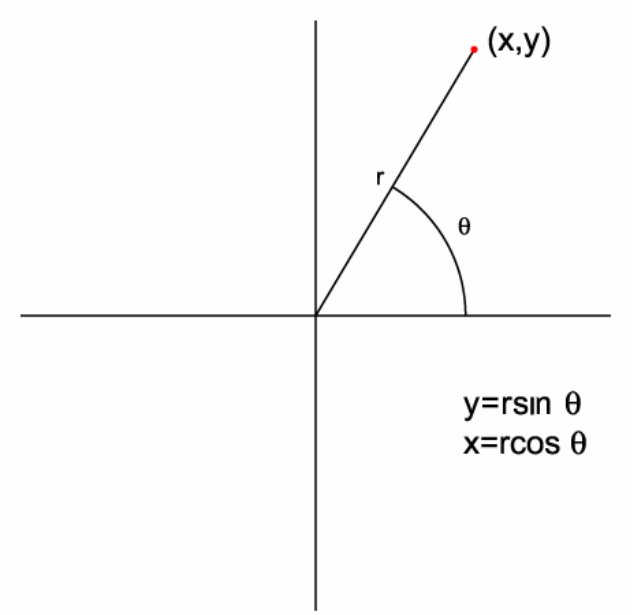


Figure 4.16: A relationship between polar and Cartesian coordinates.

Because the circular method of defining the trig functions is more widely applicable, and is perfectly compatible with the triangular method, this is generally the method programmed into your calculator and computer.

For illustration, the graphs of the sine, cosine, and tangent functions are shown in Figure 4.17.

> **Note:** The domain of the tangent function excludes certain points -- namely, all those points where its definition would otherwise require it to divide by zero. The domain of the other functions, sine and cosine, is the set of all real numbers, thanks to this new way of defining the trig functions.

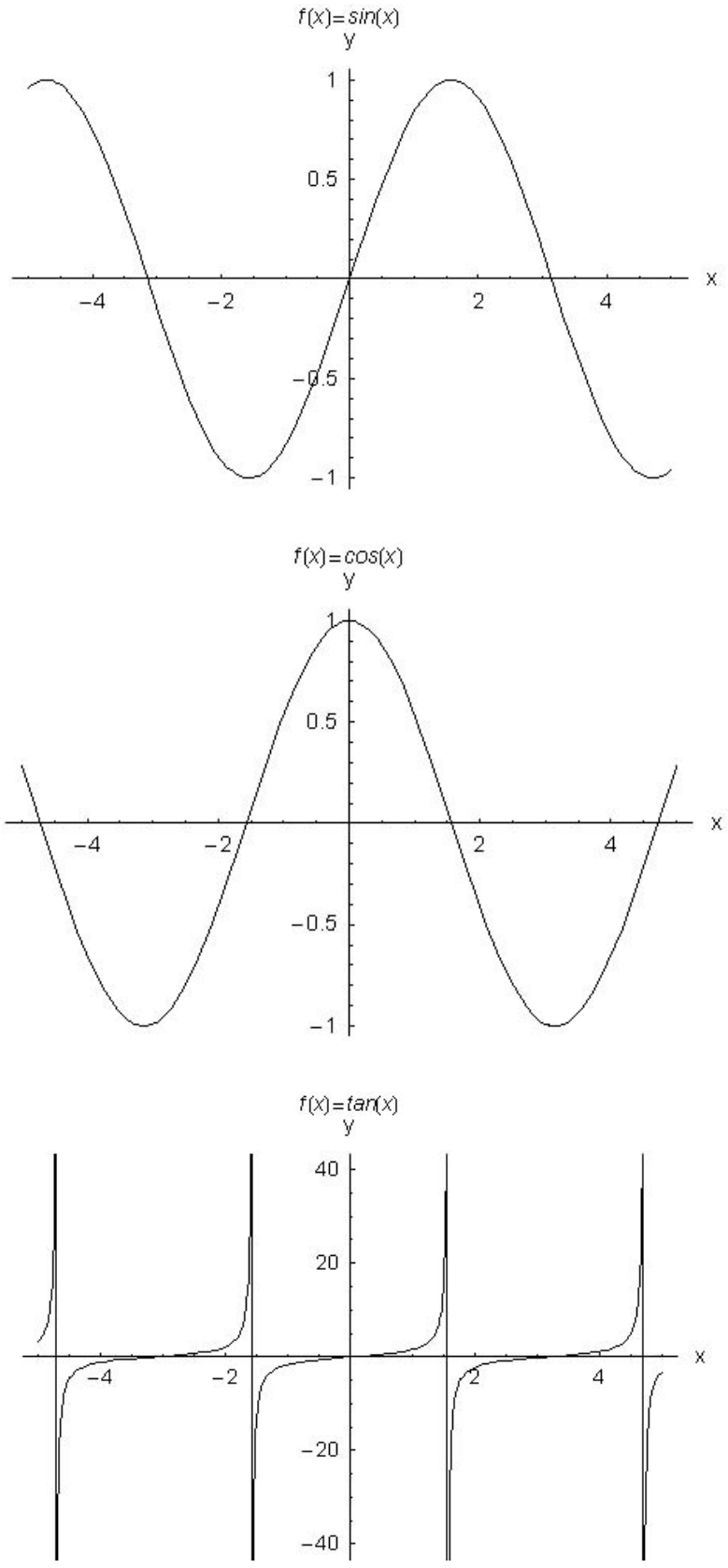


Figure 4.17: The graphs of the trig functions.

4.5 Applications of Basic Trigonometry

Trigonometry pops up *everywhere* in computer science, especially in the field of game development. Some of trigonometry's applications just add dazzle to a 3D world, others are absolutely critical for displaying 3D geometry on a computer screen. In this section we will discuss both kinds of problems, with an emphasis on the ones *essential* to 3D game development.

4.5.1 Solving Triangle Problems

We said before that solving problems involving right triangles was the primary reason trigonometry was invented. But we never really asked why people would want to solve such problems in the first place. As it turns out, there are numerous real-world applications, from navigation, to surveying, to installing your own personal digital satellite system, to applications in computer science (several of which we will cover later on) which take advantage of trigonometry.

So how do we use trigonometry to solve such problems? The simple answer is to just pick a trig function that involves the appropriate quantities and solve for the unknown. For example, let us say you have a triangle whose hypotenuse is 5 units long, and you know an acute angle is $\pi / 4$ radians. Then, from the definition of the sine function, you know the following relation holds:

$$\sin(\pi / 4) = \frac{o}{5}$$

where *o* is the length of the side opposite the angle. Solving for this unknown quantity, we get,

$$o = 5\sin(\pi / 4)$$

If you evaluate the sine function using a calculator or computer, you will find that $o \approx 3.54$.

To find *a*, we can use the definition of the cosine function, which tells us that:

$$\cos(\pi / 4) = \frac{a}{5}$$

Solving this equation for *a*, the length of the side adjacent to the angle, we find that $a = 5\cos(\pi / 4) = 3.54$. So with just the length of one side of the triangle, and one of the acute angles, we were able to determine the length of the other two sides. We could determine the angles too, if we wanted, by the process mentioned previously.

You can also use trigonometry to find out the *angles* of a right triangle given just the *lengths* of two sides, but this requires the more advanced trigonometry we will cover in our next lesson.

4.5.2 Modeling Phenomena

So far we have seen that mathematical functions have a great capacity to model phenomena (whether real-world or otherwise), and the trigonometric functions are no exceptions. Trig functions are ideally suited to modeling phenomena that tends to repeat in a rising and falling pattern. The intensity of a flashing light (modeled in our last lesson with a polynomial), the intensity of a police siren, and the height of waves in a pool of water are all good candidates for a trigonometric-based model.

To learn how to do this, we are going to model the height of waves traveling in water. This example will illustrate many of the techniques you will need when modeling with trigonometric functions.

Modeling Waves

If you drop a stone in still water, then the impact of the stone hitting the water will send out circular waves that travel away from the point of impact. We are going to assume that the waves will instantly propagate from the impact point out as far as the water goes, and then slowly fade as time goes on. If you want to add more realism, you can still use the function we are going to develop, you just have to restrict its use to an ever-increasing number of points around the point of impact.

In our model, the height of a wave at a particular point depends on two things: how many seconds have elapsed since the time of impact and how far the point is away from the point of impact. So our function will take the form $f(t,r)$, where *t* is the elapsed time and *r* is the distance from the point to the place of impact.

We want the height of the waves to vary with time and distance, so a good first attempt would be $f(t,r)=\sin(r+t)$ (you could also use cosine here, too).

To give us some control over exactly *how* the height varies with time and distance, we are going to add three constants in strategic positions to give us the function $f(t,r)=\alpha\sin(\beta r+\gamma t)$ (the Greek letters here are alpha, beta, and gamma, respectively).

The function is nearly finished: with greater distances or greater times, the height of the waves should decrease. We can model this by dividing the whole expression by the product of *t* and *r*, since as these variables grow larger, this will make the value of the function smaller. Or rather, we can divide by *t* and *r* raised to some power, since that will give us more flexibility. Our function then becomes:

$$f(t,r)=\frac{\alpha\sin(\beta r+\gamma t)}{\{t(r+1)\}^{\delta}}$$

(The new Greek letter here is delta).

Notice that because of the division by *t* and *r*, the function is only defined for all $t\neq 0, r\neq -1$. Further, notice that we added 1 to *r* in the denominator, because otherwise, for very small values of *t* and *r*, the

function will fail, since the reciprocal of a tiny fraction is a huge number. This addition allows us to evaluate the function for all $r \geq 0$ without any problems and is a common way to solve this issue.

Now let us say that we want the maximum height of the waves to be 5 units. Since the sine function varies between -1 and 1 (check out its graph in Fig 4.17), if we multiply it by 5, then it will vary between -5 and 5. So a good guess for α is 5 (it is only a guess because we are ignoring the fact that we divide by a power of *t* and *r*; it is a *good* guess because for small *t* and *r*, this will nearly be equal to 1).

The rest of the constants are rather difficult to evaluate analytically, no matter what constraints you impose on the function (you also need inverse trig functions, which we will not cover until the next lesson). For something this complicated, it is recommend that you experiment with the constants until you come up with something that looks nice. Before you do this, however, it is helpful to know that β will affect the spacing of the waves, γ will affect the rate at which the waves travel, and δ will affect the way the height of the waves decreases with time and distance.

Figure 4.18 shows just one frame of an animation for what the water would look like with a version of the function that was derived after a few minutes of experimentation.

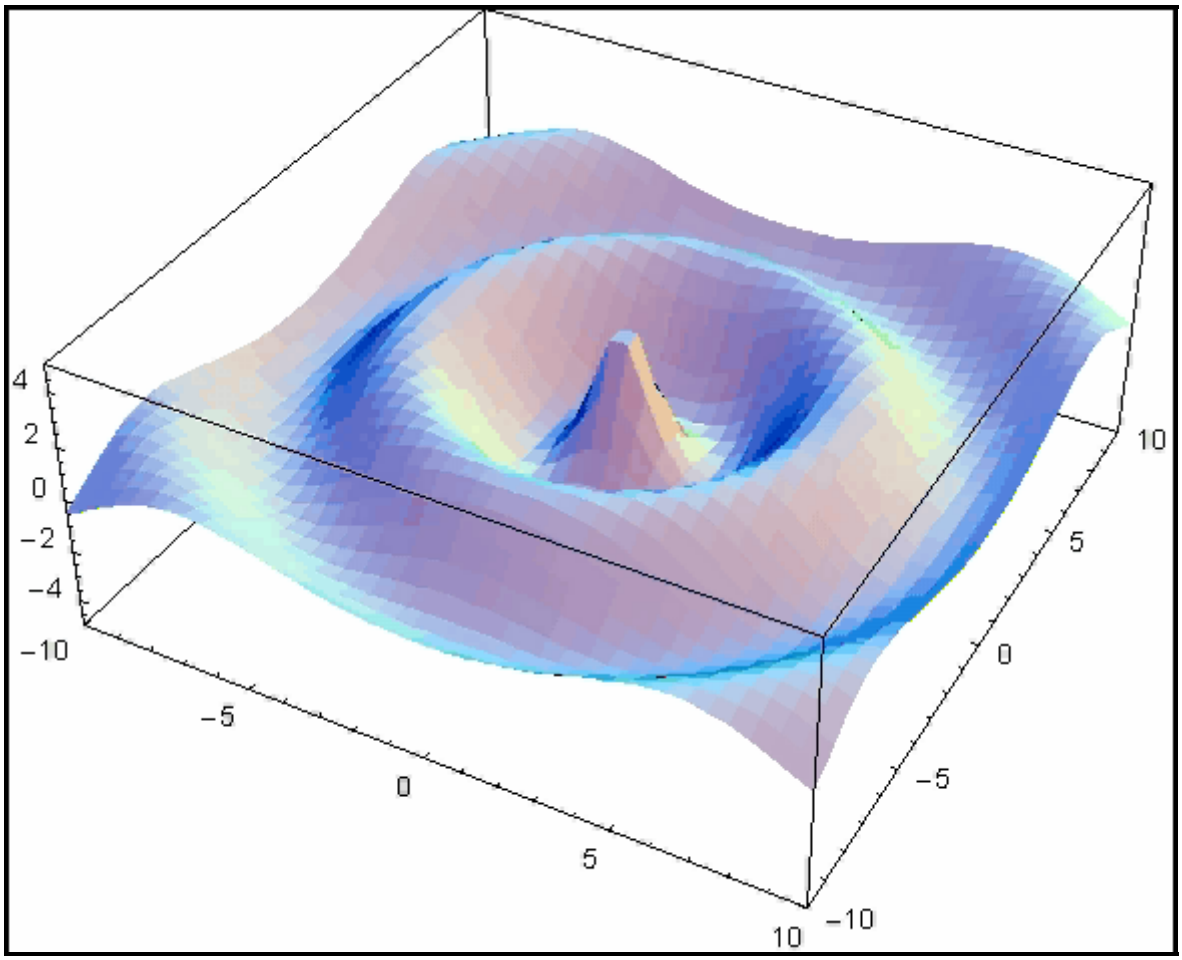


Figure 4.18: Waves traveling through water.

Drawing Circles and Ellipses

Although not particularly efficient, you *can* use the trigonometric functions to draw circles and ellipses on a computer screen, or even generate polygons that approximate these primitives.

To display a circle, just select a set of angles (preferably equidistant from each other) and use these angles and the radius of the circle to form polar coordinate pairs. Then convert these polar coordinates into Cartesian coordinates, and display the resulting points on the screen.

In C++, this whole process might look something like the following:

```
void DisplayCircle (float Radius, int AngleCount)
{
   for ( float Angle = 0.0F; Angle < ( 2.0F * 3.14596F );
         Angle += ( 2.0F * 3.14596F )/( float ) AngleCount ) {
      float X = Radius * cos ( Angle );
      float Y = Radius * sin ( Angle );

      DisplayPoint ( X, Y );
   }
}
```

Exactly *where* the circle is drawn on the screen depends on how the function **DisplayPoint()** interprets the coordinates. The circles will be centered on whatever screen location the function maps the point (0, 0) to (in DirectX, this is typically the upper-left hand corner of the screen).

You could also add fixed integers to **X** and **Y** so that you could control the location of the circle on the screen. The following code does just this:

```
void DisplayCircle(float Radius, int AngleCount, int CenterX, int CenterY )
{
   for ( float Angle = 0.0F; Angle < ( 2.0F * 3.14596F );
         Angle += ( 2.0F * 3.14596F )/( float ) AngleCount ) {
      float X = Radius * cos ( Angle );
      float Y = Radius * sin ( Angle );

      DisplayPoint ( X + CenterX, Y + CenterY );
   }
}
```

Keep in mind that how the values of **CenterX** and **CenterY** correspond to a given screen location is still determined by the **DisplayPoint ()** function.

Ellipses are only slightly more difficult to display than circles. Ellipses have *two* "radii", one equal to half the distance from the top of the ellipse to the bottom, and the other equal to half the distance from the left of the ellipse to the right. These are called the *horizontal semi-radius* and the *vertical semi-radius*, respectively, as shown in Figure 4.19.

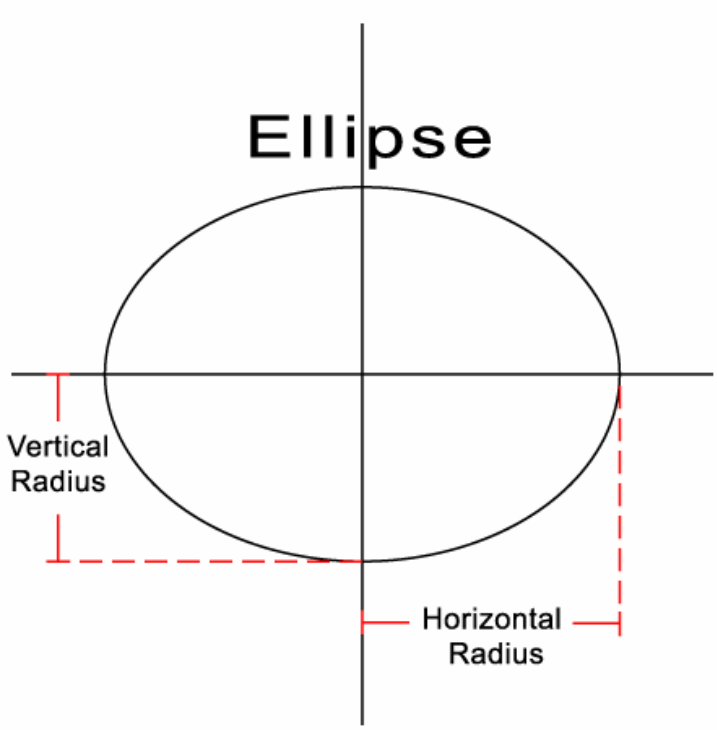


Figure 4.19: The radii of an ellipse determine its shape.

Instead of multiplying the *radius* by the cosine of an angle to get its *x* coordinate, with an ellipse you multiply the horizontal semi-radius by the cosine of the angle to get that coordinate. Similarly, you multiply the vertical semi-radius by the sine of the angle to get the *y* coordinate. You can see why this works by recalling that both the cosine and sine functions map to values in between -1 and 1, so when you multiply them by the major and minor axes, they map to the extents of the ellipse.

The basic code for drawing an ellipse is shown below:

```
void DisplayCircle(float MajorAxis, float MinorAxis, int AngleCount,
                   int CenterX, int CenterY )
{
   for ( float Angle = 0.0F; Angle < ( 2.0F * 3.14596F );
         Angle += ( 2.0F * 3.14596F )/( float ) AngleCount ) {
      float X = MajorAxis * cos ( Angle );
      float Y = MinorAxis * sin ( Angle );

      DisplayPoint ( X + CenterX, Y + CenterY );
   }
}
```

Rotating Points

One of the most important applications of trigonometry is rotating points in 2D and 3D space. Unfortunately this application requires some more advanced trigonometry that we will cover in out next lesson, so we will postpone the topic of rotation until then.

Projecting 3D Geometry onto a 2D Screen

Your computer screen is a two-dimensional surface, yet you play 3D games on your computer all the time. How so? This magic is made possible by a mathematical trick known as *projection*.

The three-dimensional worlds of computer games are composed of polygons (which you can think of as paper cutouts), which are in turn defined by a series of *vertices* (the points that define the shape of the polygons). To understand how projection works, picture a ray sent out from your eye, into the screen, to each point of a polygon. Wherever the ray intersects the screen, *that is* where it is drawn on your screen. This process, illustrated in Figure 4.20, is what makes the illusion of three dimensions possible.

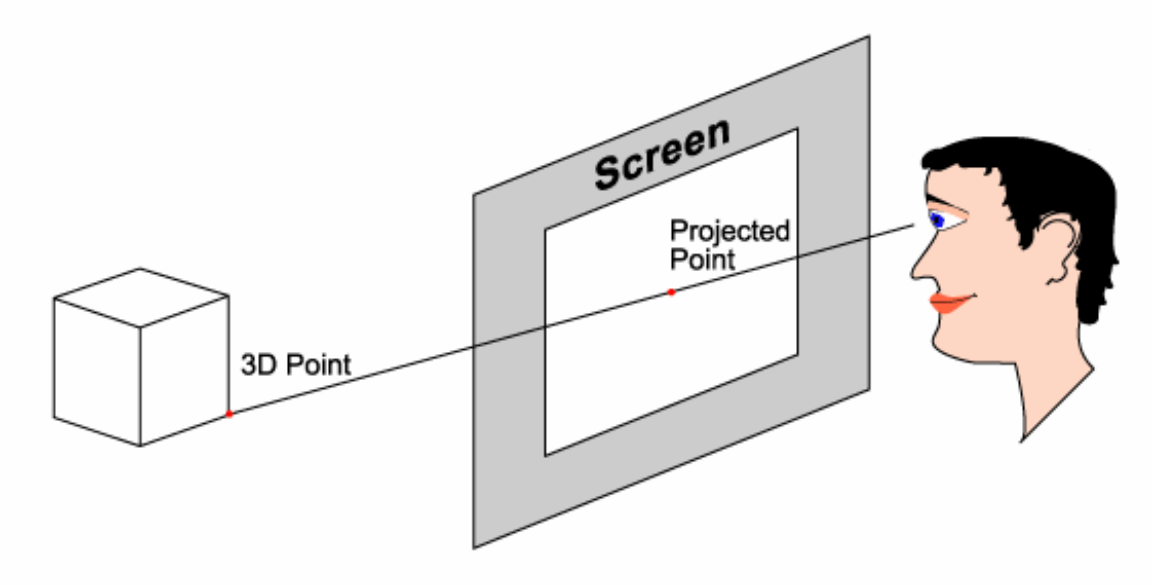


Figure 4.20: The process of geometry projection.

The task of projection is this: given a 3D point, what is its projected location on a 2D computer screen?

The actual math behind projection is quite simple, and requires only the basic trigonometry we have covered in this chapter. Figure 4.21 shows a side view of the viewer sitting in front of the computer screen and a three-dimensional point inside the screen.

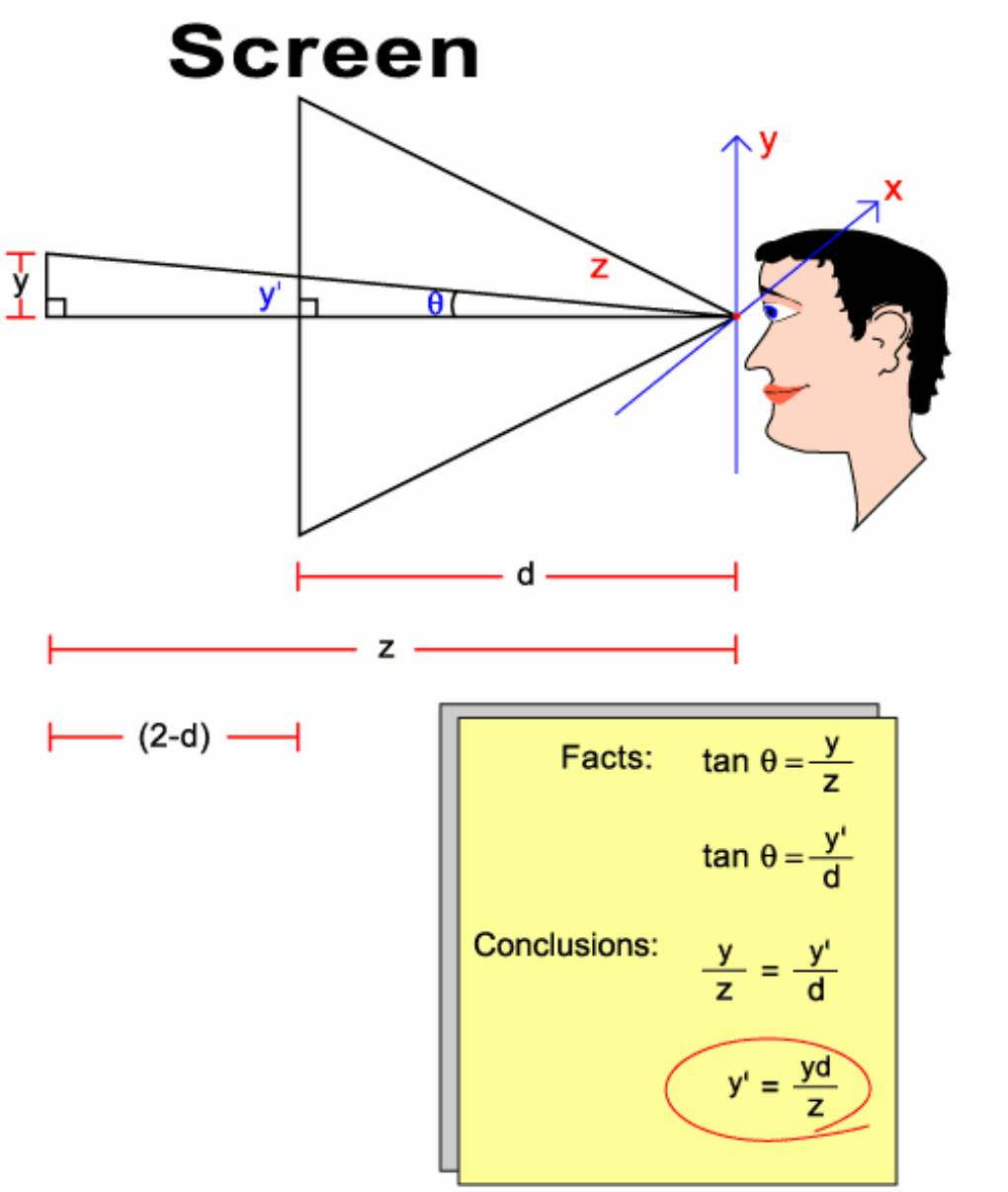


Figure 4.21: The math of geometry projection.

In the figure, a ray is traced from the viewer, through the screen, and to the point. This ray actually forms the hypotenuse of two similar triangles, which are also shown in the figure. To simplify matters, we have placed the origin of the coordinate system at the location of the viewer, who is assumed to be centered both horizontally and vertically in front of the computer screen.

The ray drawn from the viewer to the point being projected makes an angle θ with the *z-axis*. This is also one of the acute angles of the two triangles depicted in the figure.

The point being projected onto the screen is labeled (*x*, *y*, *z*). Since this is a side view of the viewer, you really cannot see the *x* coordinate of the point, but you can see its *y* and *z* coordinates clearly. The *z* coordinate tells you how far in or out of the screen the point is, and the *y* coordinate tells you how far up or down the point is, as mentioned in lesson two.

The vertical location where the ray intersects the computer screen is designated *y'*. The horizontal location, which we would label *x'*, is not pictured, since we are looking at the side view.

The distance from the viewer to the screen (measured in the same units as the coordinate system) is designated *d*; this distance is somewhat arbitrary and determines how much of the world the viewer sees. If the distance is small, then the viewer is right up against the screen and sees a lot of the world; if the distance is larger, then the viewer sees only a small portion of the world.

From the figure and the definition of the tangent function, it is obvious that $\tan(\theta) = y / z$ and $\tan(\theta) = y' / d$, so we can conclude $y / z = y' / d$, and hence, that $y' = yd / z$. You can follow a similar procedure to conclude that $x' = xd / z$ (just view the whole thing from the top -- the only things that change are the labels).

That is basically all there is to geometry projection!

Conclusion

As we have seen in this lesson, trigonometry is a powerful subject. But there is more to trigonometry than just the sine, cosine, and tangent functions we have looked at here. There are other trig functions, pseudo-inverse trig functions, and a multitude of formulas and identities that can all be used to solve more challenging problems. These topics are the subject of our next lesson.

Exercises

1. Fill in the missing information about this right triangle:

Hypotenuse: 10 **Angle**: $\pi/8$ radians **Opposite**: **Adjacent**:

2. Fill in the missing information about this right triangle:

Hypotenuse: **Angle**: $\pi/8$ radians **Opposite**: 3 **Adjacent**:

3. Fill in the missing information about this right triangle:

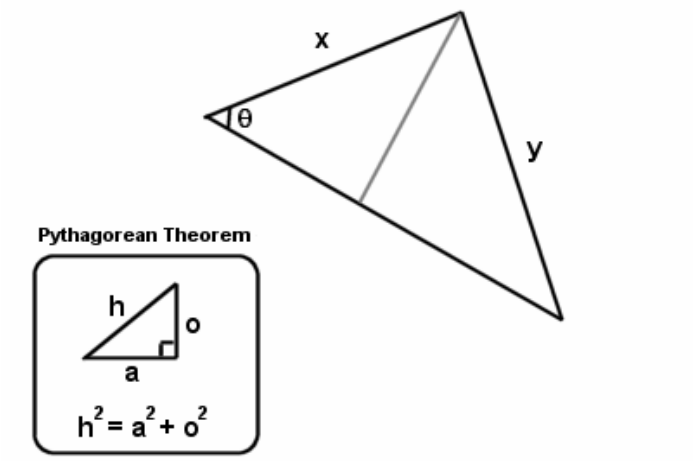
Hypotenuse: **Angle**: $\pi/8$ radians **Opposite**: **Adjacent**: 7

*4. Suppose you have n functions, labeled $f_1(r, t)$, $f_2(r, t)$, ..., $f_n(r, t)$, and that the ith function models the height of a water ripple at time t at a distance r away from the impact point (x_i, y_i). Assuming all of the ripples occur in a single pond, write down an expression describing the height of an arbitrary point (x, y) at time t. (The distance between two points (a, b) and (c, d) is $\sqrt{(a-c)^2+(b-d)^2}$.)

5. Polynomials can be used to represent can be used to approximate trigonometric functions (or even represent them perfectly, for polynomials of infinite degree). Determine which of the trig functions is approximated by the following polynomial:

$$f(x) = 1 - \frac{x^2}{1\times 2} + \frac{x^4}{1\times 2\times 3\times 4} - \frac{x^6}{1\times 2\times 3\times 4\times 5\times 6}$$

*6. Given Pythagorean's theorem and what you know of trigonometry, determine the area of the triangle shown in Figure E4.1 (the area of a triangle is used to compute its mass in some physics simulations).



E4.1: The setup for problem 6.

*7. Suppose you are developing an RPG game and the speed at which a party can travel depends on the temperature (on the assumption that people travel slower in colder weather). Develop a function to model the temperature of the weather assuming trig-like behavior, subject to the following constraints: the time is measured in hours, the temperature, in degrees Celsius; the temperature reaches a low point and a high point exactly once every 24-hours; the minimum temperature is -10 C; the maximum temperature is 30 C.

*8. Improve upon the model you developed in problem 7 by incorporating small weather fluctuations throughout the day (caused by wind, changing locations or altitudes, etc.). The graph of your final function might look something like Figure E4.2. (Hint: You can obtain Figure E4.2 by summing various sine or cosine functions.)

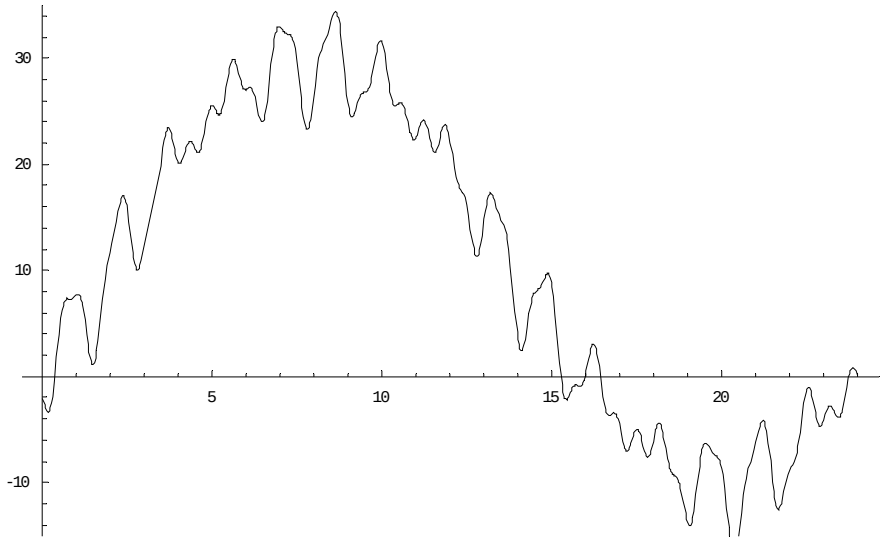
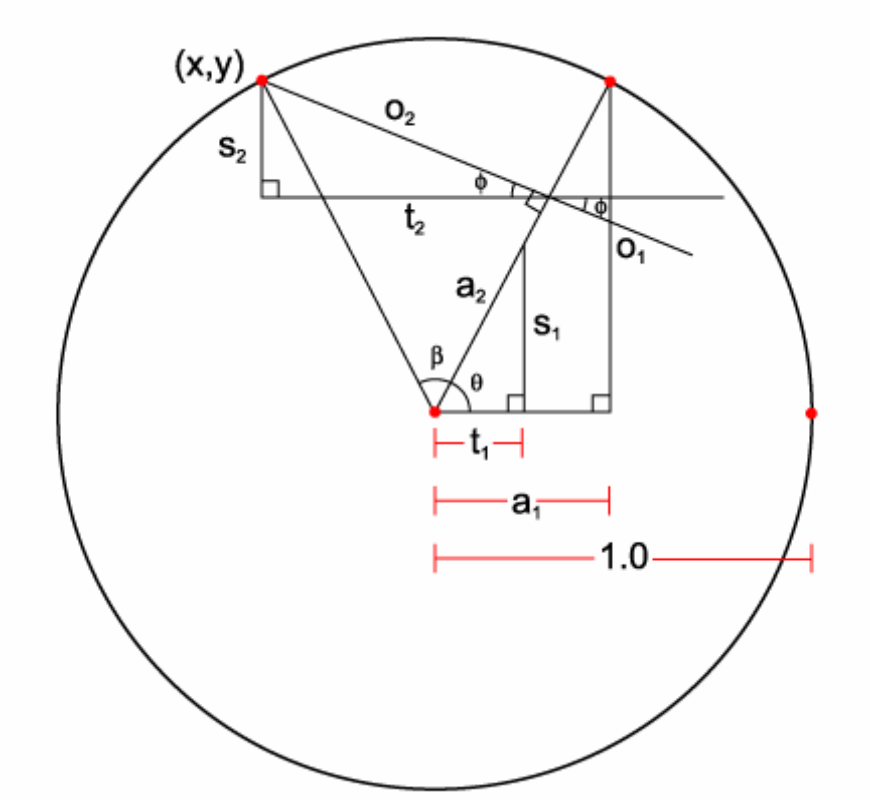


Figure E4.2: Chaotic weather behavior.

Chapter Five

Basic Trigonometry II



Introduction

In our last lesson we discussed the fundamental ideas in trigonometry: the concept of angles, the polar coordinate system, the types and properties of triangles, and the three core trigonometric functions -- sine, cosine, and tangent. While this limited look at trigonometry yielded a wealth of applications, there is a lot more to trigonometry than just these basics.

In this chapter, we will look at derivative trigonometric functions (which are just useful variants of the functions we have already seen), inverse trigonometric functions, and a variety of identities that will help you when you are trying to solve trigonometric problems. We will also learn how to use these functions in game development by deriving the formulas for rotation -- formulas that have been used in every 3D game ever made.

5.1 Derivative Trigonometric Functions

Back in the dark ages of mathematics, a wise soul asked, "Why have just three trigonometric functions when we can have six?" And so it was: the number of trigonometric functions was doubled.

What did these new functions look like? They were reciprocals of the three existing functions. Since all the ratios had been defined (opposite over hypotenuse, adjacent over hypotenuse, and opposite over adjacent), all that was left were the multiplicative inverses of those ratios.

Those derivative trigonometric functions were named *cosecant*, *secant*, and *cotangent*, and are designated by the function names *csc*, *sec*, and *cot*. Cosecant is the reciprocal of sine, secant is the reciprocal of cosine, and cotangent is the reciprocal of tangent.

Table 5.1 summarizes all the trig functions learned thus far.

Function	Mathematical Name	Triangular Definition	Circular Definition
Sine	Sin	o/h	y/r
Cosecant	*csc*	*h/o*	*r/y*
Cosine	Cos	a/h	x/r
Secant	*sec*	*h/a*	*r/x*
Tangent	Tan	o/a	y/x
Cotangent	*cot*	*a/o*	*x/y*

Table 5.1: Summary of the trigonometric functions.

These functions do not really do anything new, but they do enable you to simplify some otherwise complicated expressions. For example, instead of writing $\frac{1}{\cos(x)(\tan(y))^2}$, you can just write $\sec(x)(\cot(y))^2$.

5.2 Inverse Trig Functions

Often in a trigonometric problem, you will know the lengths of two sides of a right triangle, and want to determine from this information what the angles of the triangle are.

For example, you know that the legs of a right triangle are 5 and 3 units long, respectively. Then, by using the tangent function, you know that $\tan(\theta) = 5/3$, where θ is the angle of the triangle opposite the side with the length of 5 (see Figure 5.1). Unfortunately, this equation does not tell you what θ is -- it only tells you what the *tangent* of θ is (namely, 5/3).

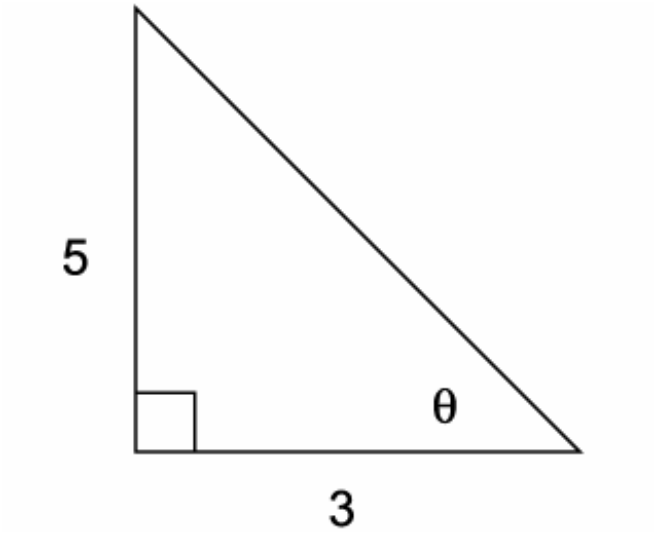


Figure 5.1: A right triangle that has two unknown angles.

In theory, you could randomly choose a variety of angles, take their tangents, and see which one was closest to 5/3. Then you could modify that angle, make it larger or smaller, and see if the tangent of that angle moved closer or father away from 5/3. With much trial and error, you would eventually stumble upon an angle whose tangent was approximately equal to 5/3.

If the process were really this cumbersome though, you would not be able to use it in computer games; it would be only minimally useful for trigonometric calculation. Fortunately, there is a better way.

Recall that if invertible function *f* maps from *A* to *B*, sending an element *a* in *A* to *b* in *B*, then the inverse of the function, denoted f^{-1}, maps from *B* to *A*, sending *b* in *B* to *a* in *A*.

If we had an inverse for the tangent function, then in order to solve for θ in the equation $\tan(\theta) = 5/2$, all we would have to do is take the inverse tangent of both sides, giving us the equation, $\tan^{-1}(\tan(\theta)) = \tan^{-1}(5/2)$. By definition of an inverse function, $\tan^{-1}(\tan(\theta)) = \theta$, the expression simplifies to $\theta = \tan^{-1}(5/2)$. At this stage, all we would have to do to calculate θ is evaluate the inverse tangent function for the value 5/2.

However, the problem is not quite that simple, since not *all* functions are invertible. What defines an invertible function? Recall that an invertible function is a function $f : A \to B$ such that every element in *B* is associated with *exactly* one element in *A*.

Now consider what this condition tells us about the graph of an invertible function, so we can see if the trigonometric functions are invertible. The graph of a function $f : A \to B$ is a set $G = \{(x, y) \mid x \in A, y \in B\}$. If the function is an inverse, we know that if (x_1, y) and (x_2, y) are both elements in *G*, then $x_1 = x_2$; otherwise, the element *y* in *B* would be associated with two elements in *A* (namely, x_1 and x_2), which violates the definition of an invertible function.

So for a given *y* value, there can only be a *single* corresponding *x* value in the graph of an invertible function. This gives us a visual test: draw any horizontal line across the graph of a function, and if the function is invertible, then the line will not intersect the graph of that function at more than one point. Figure 5.2 shows an example of an invertible function.

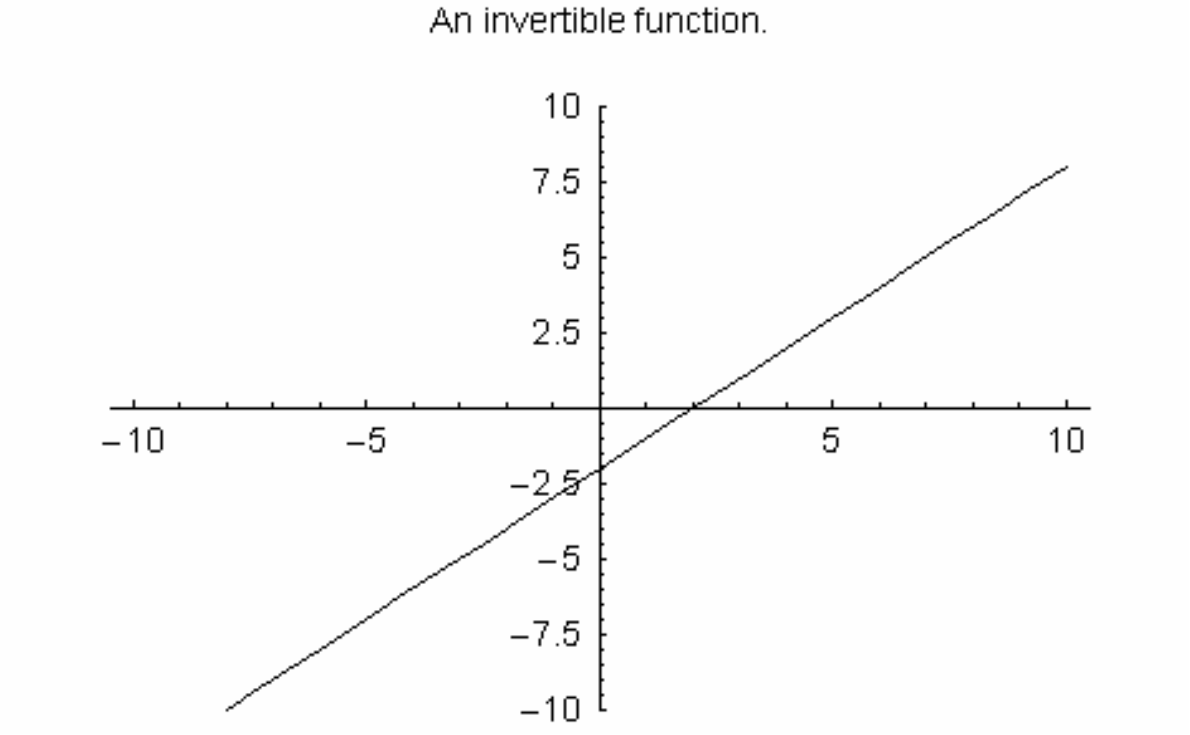


Figure 5.2: Graphical representation of an invertible function.

To refresh your memory of the three primary trigonometric functions, the sine function graph is shown in Figure 5.3. (You can refer back to Chapter Four if you wish to see the graphs for cosine and tangent.)

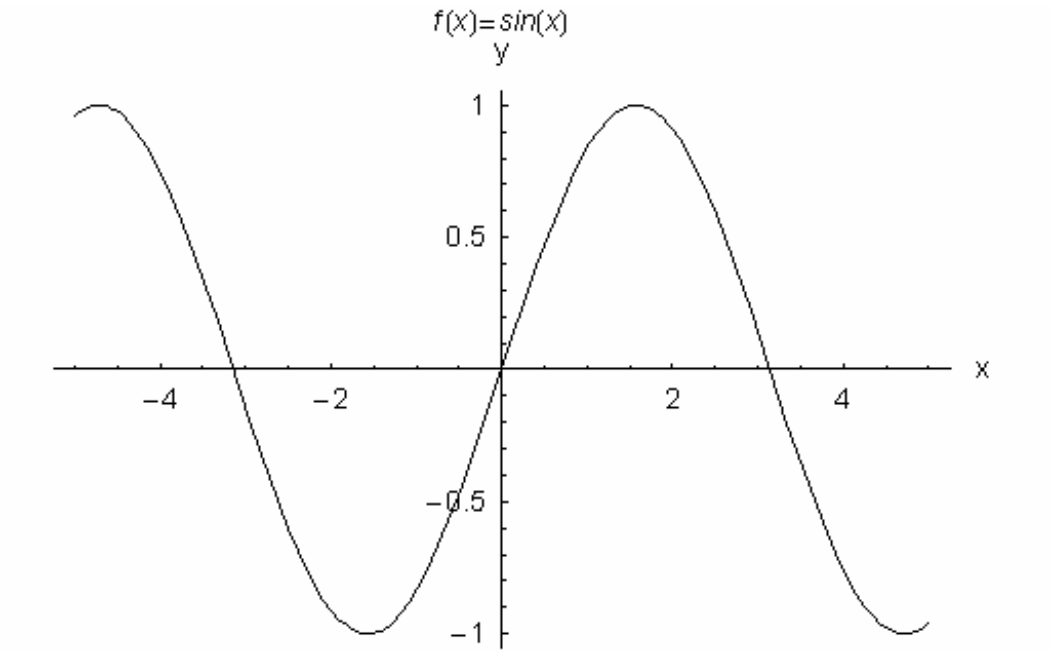


Figure 5.3: The graphs of the sine, cosine, and tangent functions.

As you can see, a horizontal line drawn at 0.5 (for example) will intersect the graphs of the trig functions many times -- in fact, an infinite number of times. This means the trig functions associate an infinite number of *x* values with *y* = 0.5. Thus, the trigonometric functions are not invertible.

This is not too surprising when you think about it. Consider the sine function. Suppose we know that $\sin(\theta) = 1$, and we want to find out what θ is. Now $\pi/4$ would satisfy the equation, since $\sin(\pi/4) = 1$. So we might think to define the inverse sine function so that $\sin^{-1}(1) = \pi/4$, since that would enable us to write $\sin^{-1}(\sin(\pi/4)) = \sin^{-1}(1) = \pi/4$. But the problem is, $\pi/4$ is not the *only* angle that satisfies the equation $\sin(\theta) = 1$. The angle $5\pi/4$ works just as well, since $\sin(5\pi/4) = 1$. In fact, any angle of the form $\pi(1+4n)/4$ will work, where *n* is *any integer at all*. So if we went ahead and said $\sin^{-1}(1) = \pi/4$, then since $\sin(5\pi/4) = 1$, we could write, $\sin^{-1}(1) = \sin^{-1}(\sin(5\pi/4)) = \pi/4$. But this equation clearly violates the definition of an inverse function, since $5\pi/4$ is not equal to $\pi/4$.

If this seems abstract, here is another way to view it. The trig functions send many different angles to a single real number. For example, the sine function sends $\pi/2$, $5\pi/2$, $9\pi/2$ and so on, to the real number 1. If our inverse function sent 1 back to only one of these angles, then it would not be a true inverse function. On the other hand, if it sent 1 back to all of these numbers, it would not be a function at all, since a function cannot send an item in the domain to more than one item in the range.

However, all hope is not lost. Even functions with limited inverse trigonometric capabilities are extremely useful, which is why the mathematicians came up with *partial* inverses to the trig functions. These functions send real numbers to angles. And while these angles might not be the angles you are looking for, they are related to them.

The inverse functions are known as *arcsine*, *arccosine*, and *arctangent*, and are denoted with the function names *arcsin*, *arcos*, and *arctan*. (Sometimes you will also see the standard inverse notation $\sin^{-1}$, $\cos^{-1}$, and $\tan^{-1}$, but keep in mind that these functions are not really true inverses of the trigonometric functions.)

The function $\arcsin(x)$ is defined to be a real number θ such that $\sin(\theta) = x$, where θ falls in between $-\pi/2$ and $\pi/2$. The first condition gives the function its inverse properties. The second condition cuts the number of solutions down from infinity to exactly 1. The domain of the function is the same as the range of the sine function, the set of all real numbers between -1 and 1.

The function $\arccos(x)$ is defined to be a real number θ such that $\cos(\theta) = x$, where θ falls in between 0 and π. The domain of the function is the same as the range of the cosine function, the set of all real numbers between -1 and 1.

The function $\arctan(x)$ is defined to be a real number θ such that $\tan(\theta) = x$, where θ falls in between $-\pi/2$ and $\pi/2$. The domain of the function is the same as the range of the tangent function, the set of all real numbers.

For illustration, the graph of the inverse cosine function is shown in Figure 5.4.

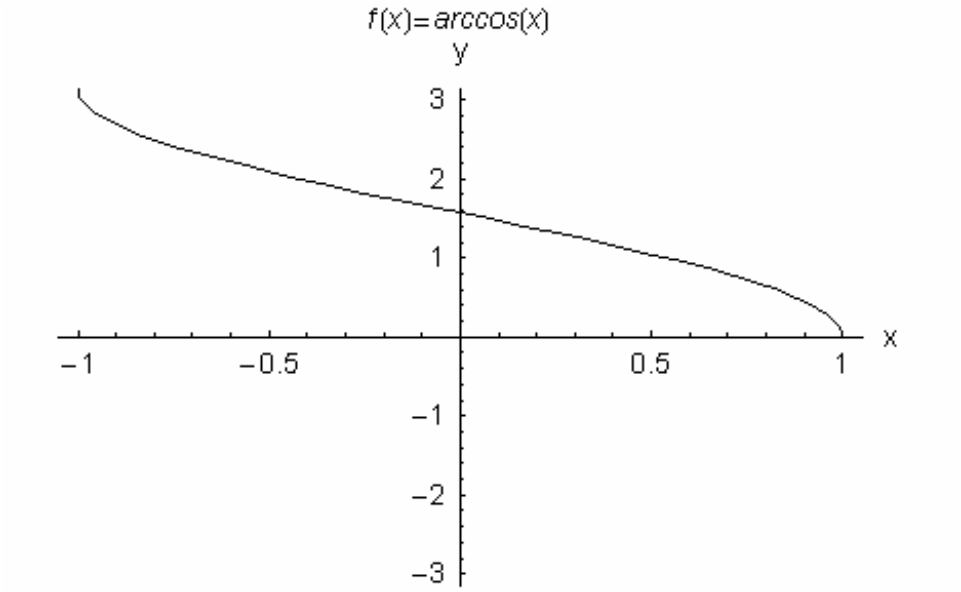


Figure 5.4: The graph of the inverse cosine function.

Without relying on the graphs of the arc functions, let us look at an example of how we would evaluate them for a given real number between -1 and 1. Say we want to know what $\arcsin(1)$ is. By definition, we are looking for a real number θ such that $\sin(\theta) = 1$, and we want θ to fall between $-\pi/2$ and $\pi/2$. All we have to do is look at a graph of the sine function from $-\pi/2$ to $\pi/2$ and find the point in this interval where the *y* value is equal to 1. The *x* value of this point is θ. Or, if you are familiar with

the sine function, you can guess what θ is immediately. In either case, $\theta = \pi/2$, since $\sin(\pi/2) = 1$ and $\pi/2$ falls between $-\pi/2$ and $\pi/2$. That is how we know that $\arcsin(1) = \pi/2$.

In real life, you would never go through this complicated process to calculate the value of an inverse trigonometric function. You would instead use a calculator, a computer algebraic system like Mathematica™, or your compiler's built-in inverse trigonometric functions (in C and C++, they are called **asin()**, **acos()**, and **atan()**, and are defined in the header math.h). But it is important for you to understand exactly how the definitions of the functions work. You may want to repeat the above process for a few other easy angles, relying on Figure 5.4 (or your calculator) if you get stuck.

Now that we have defined the inverse trigonometric functions and have seen what the definitions entail, we will cover the *sense* in which these inverse trigonometric functions are *inverses*. After all, if they can not give us angles from ratios (which is what we are interested in), then they can not help us.

To see the properties of the inverse trig functions, let us further examine the properties of the arcsine function. Suppose we have an angle θ that falls in between $-\pi/2$ and $\pi/2$. Send the *sine* of this angle to the arcsine function. (This is perfectly permissible because the *range* of the sine function is the same set as the *domain* of the arcsine function.) Then we will have the quantity $\arcsin(\sin(\theta))$. What is this number? By definition of the arcsine function, $\arcsin(\sin(\theta))$ is a real number ϕ ("phi," since we are already using θ) such that both $\sin(\phi) = \sin(\theta)$ and ϕ falls in between $-\pi/2$ and $\pi/2$. At this point, there is only one possible value that ϕ can be, and that is θ. This is because the sines of the numbers from $-\pi/2$ to $\pi/2$ are unique, which means that the only way the equation $\sin(\phi) = \sin(\theta)$ can be true is if $\phi = \theta$. You can see this for yourself in Figure 5.5.

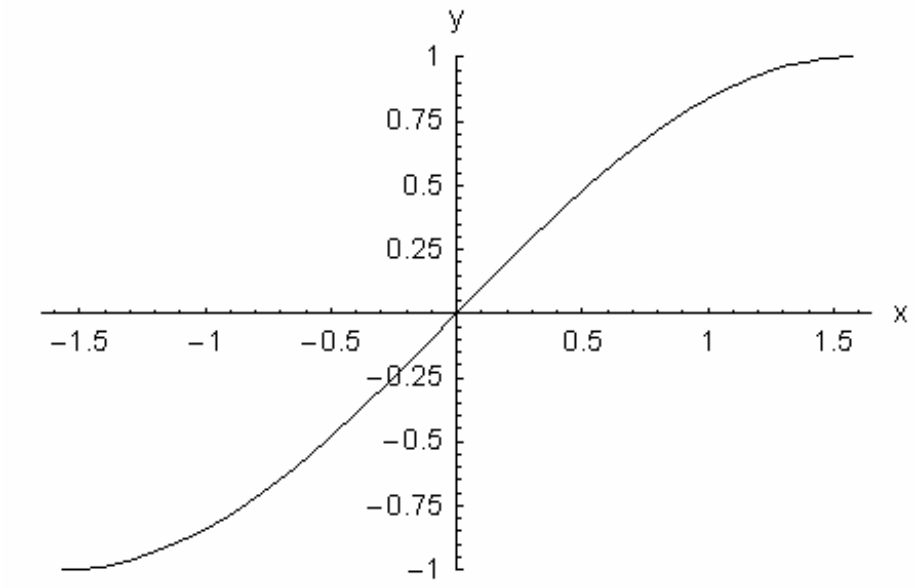


Figure 5.5: The graph of the sine function over the interval from $-\pi/2$ to $\pi/2$.

Since $\phi = \theta$, $\arcsin(\sin(\theta)) = \theta$. Does this look familiar? It should, because arcsine is acting as the inverse of the sine function, which is exactly what we wanted it to do. Similar results follow for the cosine and tangent functions, when you look at angles in the range of their respective arc functions.

What happens if, in our current example, θ does *not* fall in between $-\pi/2$ and $\pi/2$? We can still use the arcsine function on the sine function, but we will not get back θ. Instead, we will get an angle that *does* fall in between $-\pi/2$ and $\pi/2$ and is only *related* to θ. Exactly how it is related to θ depends on the quadrant that θ falls in. If it falls in the first quadrant, then the new angle is equal to θ plus some multiple of 2π. If it falls in the second or third quadrant, then the new angle is equal to $\pi - \theta$ plus some multiple of 2π. And lastly, if it falls in the fourth quadrant, then the new angle is equal to $\theta - 2\pi$ plus some multiple of 2π. (If you know that θ falls in between *0* and 2π, you do not need to add a multiple of 2π -- all that addition does is take care of the angles outside that range.)

Notice that you have to know which quadrant θ is in. Also, you have to know if and how much the angle wraps around the unit circle -- that is, how many multiples of 2π you have to add to the appropriate equation to get θ. You can usually drop the latter requirement, since angles outside the *0* to 2π range do not describe any *new* directions; they just describe the old ones in a slightly different way. The former requirement, however, is a fundamental limitation of all inverse trigonometric functions. Fortunately, more often than not, we *will* know which quadrant θ is in, making this limitation a manageable one.

You might be thinking that this is terribly tedious, and this is certainly true. But without inverse trigonometric functions, life would be far worse; you would end up having to guess what the angles were whenever you wanted to solve for them in a trigonometric equation. Also, for many of the applications that we run across in real life, we can use the arc functions as if they were real inverses of the trig functions with no ill effects, since the angle we are trying to solve for will fall in to the range of the appropriate inverse function.

Table 5.2 summarizes our discussion of the arcsine function and also shows us how the arccosine and arctangent functions work with angles in all of the different quadrants. Study this table thoroughly! The inverse trig functions are a notorious sticking point in most people's study of trigonometry.

Table 2: Summary of the inverse trig functions.

θ	**Sine**	**Cosine**	**Tangent**
Quadrant I	$\theta = \arcsin(\sin(\theta)) + 2n\pi$	$\theta = \arccos(\cos(\theta)) + 2n\pi$	$\theta = \arctan(\tan(\theta)) + 2n\pi$
Quadrant II	$\theta = \pi - \arcsin(\sin(\theta)) + 2n\pi$	$\theta = \arccos(\cos(\theta)) + 2n\pi$	$\theta = \pi + \arctan(\tan(\theta)) + 2n\pi$
Quadrant III	$\theta = \pi - \arcsin(\sin(\theta)) + 2n\pi$	$\theta = 2\pi - \arccos(\cos(\theta)) + 2n\pi$	$\theta = \pi + \arctan(\tan(\theta)) + 2n\pi$
Quadrant IV	$\theta = 2\pi + \arcsin(\sin(\theta)) + 2n\pi$	$\theta = 2\pi - \arccos(\cos(\theta)) + 2n\pi$	$\theta = 2\pi + \arctan(\tan(\theta)) + 2n\pi$

NOTES: If $0 \le \theta < 2\pi$, then (x_2, y). Otherwise, *n* is an integer (possibly positive or negative) that specifies how many times θ wraps around the unit circle. Usually this information is not required, and the term can be safely dropped, in which case the angle calculated by the arc equation above will not be θ; rather, it will be an angle that describes the same *direction* as θ, but falls between 0 and 2π.

5.2.1 Using the Inverse Trigonometric Functions

The simplest way to use inverse trig functions is to solve problems involving right triangles. This is the easiest of all applications because the angles of a right triangle are all less than or equal to $\pi/2$ radians -- which means the arc functions act as true inverses for the trig functions.

In Figure 5.6, we see a right triangle where the legs of this triangle are 3 and 5 units in length, respectively. One thing we know about the triangle is that $\tan(\theta) = 3/5$, where θ is the angle opposite the side with the length of 3, and adjacent to the side with the length of 5. Applying the arctangent function to both sides, we find that $\arctan(\tan(\theta)) = \theta = \arctan(3/5) \approx 0.54$ radians.

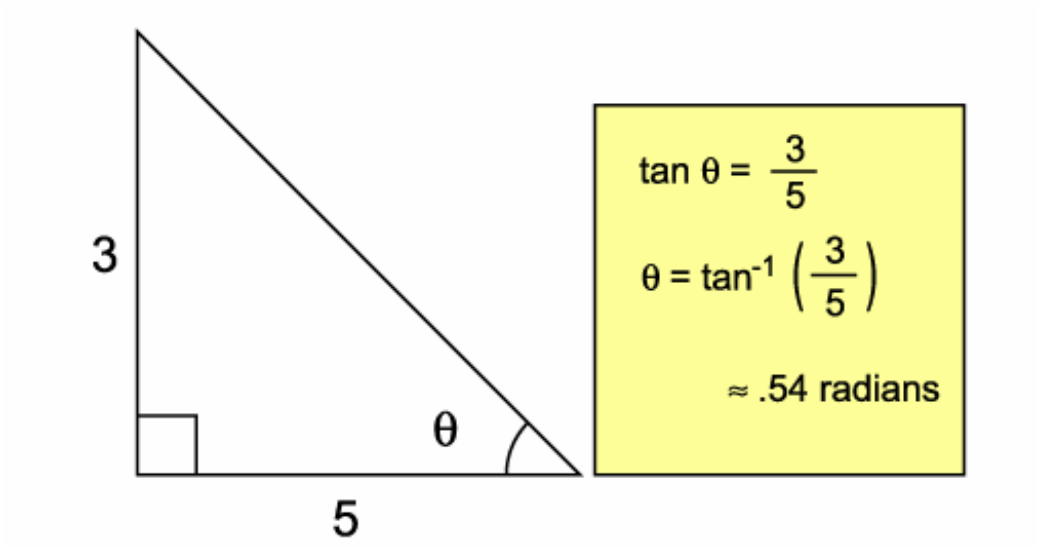


Figure 5.6: Using the arctangent function to determine the angles of a right triangle.

You can also use the other arc functions to solve for the angles of a triangle. Suppose you know the hypotenuse of a right triangle is 13 units long, and the length of one of the legs is 2 units long. Then you know that $\sin(\theta) = 2/13$, where θ is the angle opposite the leg with the length of 2. Solving for θ, we get $\theta = \arcsin(2/13) \approx 0.15$ radians.

A harder application of inverse trigonometry involves converting Cartesian coordinates to polar coordinates. Say we have a point (*x*, *y*) measured in Cartesian coordinates and want to convert it to a point (r, θ) measured in polar coordinates. If we draw a right triangle whose hypotenuse extends from the origin to the point, then we can calculate *r* by using the Pythagorean theorem. This theorem tells us that $r^2 = x^2 + y^2$, so we know that $r = \sqrt{x^2 + y^2}$. (We use the positive square root because *r* is defined to be positive, so the negative square root would not make much sense). You can see this relationship in Figure 5.7.

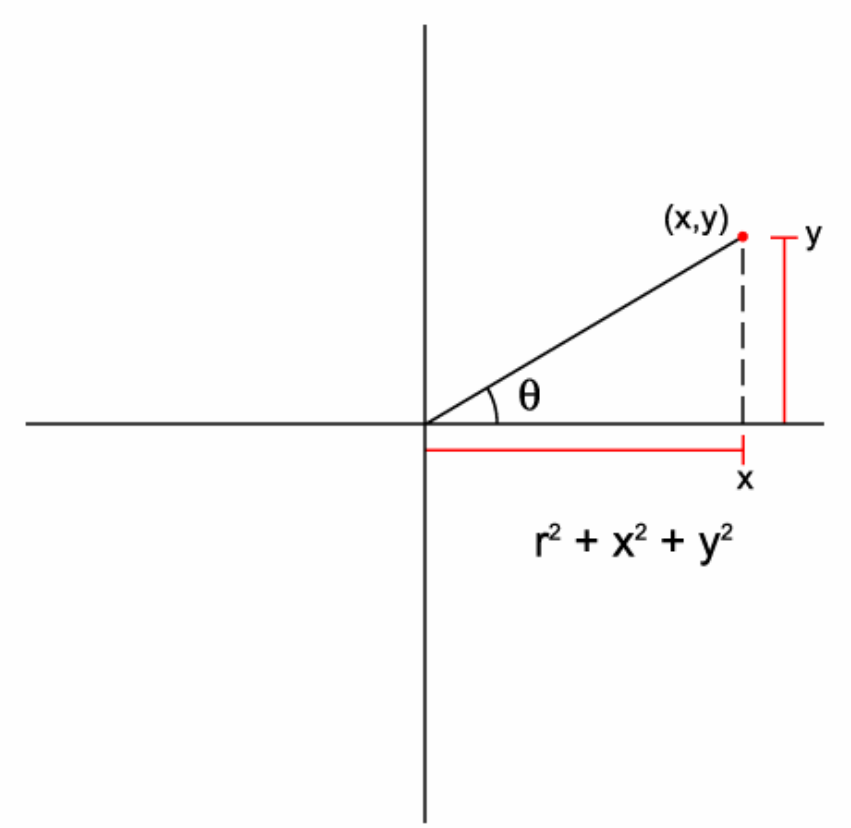


Figure 5.7: Converting Cartesian coordinates to polar coordinates.

Now comes the hard part -- determining what θ is. To make things simpler, we first assume that $0 \le \theta < 2\pi$, so we do not have to add multiples of 2π to the angle that we calculate. Next, we determine what *quadrant* θ is in. This is not as difficult as it sounds, since the signs (whether positive or negative) of *x* and *y* already provide us with this information. Specifically, if both *x* and *y* are positive, θ lies in the first quadrant. If *x* is negative and *y* is positive, θ lies in the second quadrant. If both *x* and *y* are negative, θ lies in the third quadrant. And if *x* is positive and *y* is negative, θ lies in the fourth quadrant.

Once we have determined which quadrant θ lies in, we just use the appropriate arctangent equation given in Table 5.2. For example, if θ fell in the fourth quadrant, we would know that $\theta = 2\pi + \arctan(y / x)$.

The conversion from Cartesian coordinates to polar coordinates is done so often that some compilers include a second version of the arctangent function; one that accepts a point and returns the angle between the positive *x*-axis and the point (in C++, this function is called **atan2 ()**). However, depending on how the function is implemented, it may not return an angle between *0* and 2π (**atan2()**, for example, returns an angle between $-\pi$ and π).

While that is not all there is to say about inverse functions -- it is enough to get you started. The rest of this lesson's material will focus on easier issues, such as useful trigonometric equations and how you can use the new functions you have learned in game development.

5.3 The Identities of Trig Functions

The term *identity* in trigonometry refers to an equation that relates one trigonometric expression to another.

Trigonometry is full of identities, ranging from the very simple, involving one or two terms, to the outrageously complex. You probably will never need to program these identities into the games you write; however, when you are brainstorming an equation or deriving a formula, you may need to call upon a trigonometric identity to transform your work into something that is a bit easier to use. Later on, we will discuss how to use trigonometry to rotate points around the origin of a Cartesian coordinate system. In order to derive an expression for this rotation, we will have to use a certain trigonometric identity.

Many mathematical courses require proving one theorem or another. There are many reasons why even game developers should know how to construct mathematical proofs for the formulas and equations they derive. Not only will this make results a lot more persuasive if shared with other game programmers, but they can have confidence knowing that a particular equation they are using will work flawlessly in their games.

For these reasons, we will now learn how to prove some trig identities. While this is not a comprehensive list proving all trig identities, the proofs we discuss will give you all the skills you need to enable you to prove your own results (as well as the selected theorems for you to prove in the exercises).

5.3.1 Pythagorean Identities

The first Pythagorean identity (there are three) relates the sine and the cosine function as follows:

Identity 5.1: $\sin^2\theta + \cos^2\theta = 1$

> **Note:** You can see that the parentheses have been dropped from the trig functions. Also, the squared symbols attached to the sine and cosine function are shorthand ways of writing $(\sin\theta)^2$ and $(\cos\theta)^2$, respectively. Both of these notations make the equations much more readable, and for this reason, you will see them often throughout this section.

Proof of Identity 5.1:

The Pythagorean identities are all based on the Pythagorean theorem. Figure 5.8 shows a right triangle whose legs are labeled *x* and *y* and whose hypotenuse is labeled *r*. The angle opposite the side *y* is designated θ.

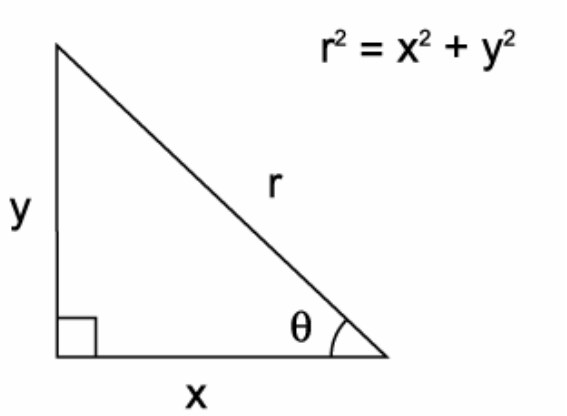


Figure 5.8: A right triangle.

The Pythagorean Theorem tells us that the length of the hypotenuse squared is equal to the sum of the lengths of the legs squared:

$$x^2 + y^2 = r^2$$

If r is non-zero (and it must be, or we have a point instead of a triangle), then we can divide both sides of the equation by r^2, which leaves us with the following equation:

$$\frac{x^2 + y^2}{r^2} = \frac{x^2}{r^2} + \frac{y^2}{r^2} = \left(\frac{x}{r}\right)^2 + \left(\frac{y}{r}\right)^2 = 1$$

We can replace x/r and y/r with $\cos(\theta)$ and $\sin(\theta)$, respectively, which gives us Identity 5.1. That is all there is to the proof.

Identity 5.2: $1 + \tan^2\theta = \sec^2\theta$

Proof of Identity 5.2:

Since we have already proven Identity 5.1, we will use that as a starting point. Dividing this identity by $\cos(\theta)^2$, we get the following result:

$$\frac{\sin^2\theta + \cos^2\theta}{\cos^2\theta} = \frac{\sin^2\theta}{\cos^2\theta} + \frac{\cos^2\theta}{\cos^2\theta} = \tan^2\theta + 1 = \frac{1}{\cos^2\theta} = \sec^2\theta$$

This is the exact result we want, so our proof is complete.

The last Pythagorean identity is provided without proof. See if you can derive your own proof for it based on what we have learned thus far:

Identity 5.3: $1 + \cot^2\theta = \csc^2\theta$

5.3.2 Reduction Identities

From looking at the graphs of the trigonometric functions, we learn that the sine function looks a lot like the cosine function, which looks a lot like the cosecant and secant functions. Similarly, the tangent function looks much like the cotangent function.

The only difference between these functions is that the graph of one is shifted to the left or right from the graph of the other by a certain number of radians. Consequently, it is possible to write down relationships between the functions.

One of these relationships is listed below:

Identity 5.4: $\sin(\pi/2 \pm \theta) = \cos(\theta)$

The symbol '$\pm$' should be read as "plus or minus." This combination symbol indicates that the expression can take two forms: one with the plus sign, and one with the minus sign. (This is more compact than writing out two identities: one with the plus sign, and the other with the minus sign.)

Identity 5.5: $\cos(\pi/2 \pm \theta) = \mp\sin(\theta)$

The new combination symbol '$\mp$' is read as "minus or plus."

Whenever there is more than one of these combination symbols, the first symbol indicates how you should interpret the following symbols. If you choose the sign on the top for the first combination symbol, then you must choose the sign on the top for all subsequent symbols, regardless of what they may be. Similarly, if you choose the sign on the bottom for the first combination symbol, then you must choose the sign on the bottom for all subsequent symbols, no matter what they are. Thus, Identity 5.5 expands in to the following two identities:

$$\cos(\pi/2 + \theta) = -\sin(\theta)$$
$$\cos(\pi/2 - \theta) = \sin(\theta)$$

The rest of the *reduction identities*, as they are often called, are shown in Table 5.3.

Table 5.3: The reduction identities.

angle	sin	cos	tan	cot	sec	csc
$-\theta$	$-\sin\theta$	$+\cos\theta$	$-\tan\theta$	$-\cot\theta$	$+\sec\theta$	$-\csc\theta$
$\pi/2+\theta$	$+\cos\theta$	$-\sin\theta$	$-\cot\theta$	$-\tan\theta$	$-\csc\theta$	$+\sec\theta$
$\pi/2-\theta$	$+\cos\theta$	$+\sin\theta$	$+\cot\theta$	$+\tan\theta$	$+\csc\theta$	$+\sec\theta$
$\pi+\theta$	$-\sin\theta$	$-\cos\theta$	$+\tan\theta$	$+\cot\theta$	$-\sec\theta$	$-\csc\theta$
$\pi-\theta$	$+\sin\theta$	$-\cos\theta$	$-\tan\theta$	$-\cot\theta$	$-\sec\theta$	$+\csc\theta$
$3\pi/2+\theta$	$-\cos\theta$	$+\sin\theta$	$-\cot\theta$	$-\tan\theta$	$+\csc\theta$	$-\sec\theta$
$3\pi/2-\theta$	$-\cos\theta$	$-\sin\theta$	$+\cot\theta$	$+\tan\theta$	$-\csc\theta$	$-\sec\theta$
$n2\pi+\theta$	$+\sin\theta$	$+\cos\theta$	$+\tan\theta$	$+\cot\theta$	$+\sec\theta$	$+\csc\theta$
$n2\pi-\theta$	$-\sin\theta$	$+\cos\theta$	$-\tan\theta$	$-\cot\theta$	$+\sec\theta$	$-\csc\theta$

5.3.3 Angle Sum/Difference Identities

The angle sum/difference identities show us an alternate way of evaluating the sine, cosine, tangent, or cotangent functions for the sum of two angles. Why is this important? This is the precise result needed to derive the formulas for the rotation of a point about the origin of a coordinate system.

Identity 5.6: $\sin(\theta \pm \beta) = \sin\theta\cos\beta \pm \cos\theta\sin\beta$

Identity 5.7: $\cos(\theta \pm \beta) = \cos\theta\cos\beta \mp \sin\theta\sin\beta$

Proof of Identity 5.6 and 5.7:

These are two of the most difficult identities to prove. At this point, we will only discuss the top of the combination symbols here -- the other cases can be derived from those.

Figure 5.9 shows the setup for the problem. Pay attention to the layout of the four triangles, and the names given to each side -- these will be critical in our proof.

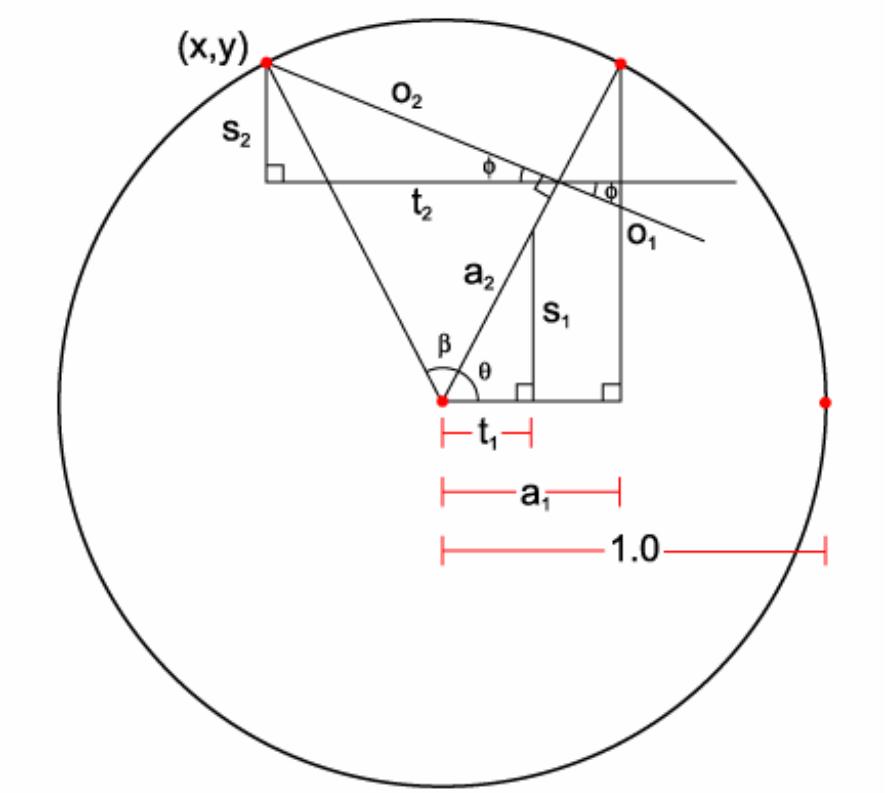


Figure 5.9: The setup for the proof of the sum and difference formulas for sine and cosine.

Notice that the radius of the circle in Figure 5.9 is 1. This simplifies the definition of the trigonometric functions. While a more general radius of *r* could have been used, doing so would only complicate the problem. Besides, by the properties of similar triangles, proving the relation for one radius proves it for all.

What we are looking for is an expansion of $\sin(\theta+\beta)$ and $\cos(\theta+\beta)$. Using the circular definition of the trig functions (and the fact that *r = 1*), you can see that these expressions are just $x/r = x/1 = x$ and $y/r = y/1 = y$, respectively. As the figure shows, $x = t_1 - t_2$ and $y = s_1 + s_2$. So what we need to do now is figure out what t_1, t_2, s_1, and s_2 are.

By the triangular definition of the trig functions, we know that $s_1 = a_2 \sin\theta = (\cos\beta)\sin\theta$ and $s_2 = o_2 \sin\phi = (\sin\beta)\sin\phi$. But since $\phi = \pi/2 - \theta$, and $\sin(\pi/2-\theta) = \cos\theta$, we can simplify this as $s_2 = \sin\beta\cos\theta$. Thus, $y = s_1 + s_2 = \cos\beta\sin\theta + \sin\beta\cos\theta$. And since $\sin(\theta+\beta) = y$, $\sin(\theta+\beta) = \cos\beta\sin\theta + \sin\beta\cos\theta$.

The proof for the cosine function runs along similar lines. We know $t_1 = a_2\cos\theta = (\cos\beta)\cos\theta$ and $t_2 = o_2\cos\phi = (\sin\beta)\cos(\pi/2-\theta) = \sin\beta\cos\theta$. Thus $x = t_1 - t_2 = \cos(\theta+\beta) = \cos\beta\cos\theta - \sin\beta\sin\theta$. Proof complete.

The following sum/difference identities are listed without proof:

Identity 5.8: $\tan(\theta \pm \beta) = \dfrac{\tan\theta \pm \tan\beta}{1 \mp \tan\theta \tan\beta}$

Identity 5.9: $\cot(\theta \pm \beta) = \dfrac{\cot\beta \cot\theta \mp 1}{\cot\beta \pm \cot\theta}$

5.3.4 Double-Angle Identities

The double-angle identities allow you to convert from a trig function evaluated at *twice* some angle to an alternate expression involving the angle *itself* (rather than twice that angle).

Identity 5.10: $\sin 2\theta = 2\sin\theta\cos\theta$

Identity 5.11: $\cos 2\theta = \cos^2\theta - \sin^2\theta = 2\cos^2\theta - 1 = 1 - 2\sin^2\theta$

Identity 5.12: $\tan 2\theta = \dfrac{2\tan\theta}{1 - \tan^2\theta}$

5.3.5 Sum-To-Product Identities

As the name suggests, sum-to-product identities convert from a sum of trigonometric functions to a product of trigonometric functions.

Identity 5.13: $\sin\theta \pm \sin\beta = 2\sin\left(\dfrac{\theta \pm \beta}{2}\right)\cos\left(\dfrac{\theta \mp \beta}{2}\right)$

Identity 5.14: $\cos\theta + \cos\beta = 2\cos\left(\dfrac{\theta + \beta}{2}\right)\cos\left(\dfrac{\theta - \beta}{2}\right)$

Identity 5.15: $\cos\theta - \cos\beta = -2\sin\left(\dfrac{\theta + \beta}{2}\right)\sin\left(\dfrac{\theta - \beta}{2}\right)$

5.3.6 Product-to-Sum Identities

Product-to-sum identities are based on sum-to-product identities. They convert from products of trigonometric functions to sums of trigonometric functions.

Identity 5.16: $\sin\theta\sin\beta = \dfrac{\cos(\theta - \beta) - \cos(\theta + \beta)}{2}$

Identity 5.17: $\cos\theta\cos\beta = \dfrac{\cos(\theta - \beta) + \cos(\theta + \beta)}{2}$

Identity 5.18: $\sin\theta\cos\beta = \dfrac{\sin(\theta+\beta)+\sin(\theta-\beta)}{2}$

Identity 5.19: $\cos\theta\sin\beta = \dfrac{\sin(\theta+\beta)-\sin(\theta-\beta)}{2}$

5.3.7 Laws of Triangles

These identities relate the angles and sides of arbitrary triangles, even if they *are not* right triangles. The identities are derived by splitting an arbitrary triangle in to right triangles.

All of the following identities refer to Figure 5.10.

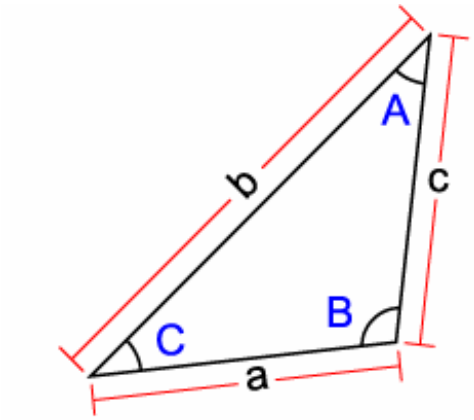


Figure 5.10: An oblique triangle.

Identity 5.20 (Law of Cosines):

$$a^2 = b^2 + c^2 - 2bc\cos A$$
$$b^2 = a^2 + c^2 - 2ac\cos B$$
$$c^2 = a^2 + b^2 - 2ab\cos C$$

Identity 5.21 (Law of Sines): $\dfrac{a}{\sin A} = \dfrac{b}{\sin B} = \dfrac{c}{\sin C}$

5.4 Applications

We will conclude this lesson by covering two applications: point rotation and field-of-view calculations.

5.4.1 Rotating Points

One of the most important applications in this lesson is *point rotation*. Point rotation, in its most general form, involves taking a point and rotating it around another point, called the *center of rotation*. However, it is simpler mathematically to rotate a point around the origin of a Cartesian coordinate system rather than some arbitrary point.

So what programmers do is *translate* (*move* without changing the orientation of) the center of rotation so it is at the origin, and then translate the point to be rotated by the exact same amount. Then they rotate the point as needed, and undo their original translation by adding back whatever they subtracted (or subtracting whatever they added). The end result is exactly the same. (Technically, this is not really the case -- programmers do not literally translate the center of rotation, but conceptually, this is a nice way to think about it -- and is the reason the whole process works).

Now, take a look at the example shown in Figure 5.11. Say we want to rotate the point (4, 1) around the point (5, 5) by π radians. The first thing we have to do is figure out how to translate the center of rotation to the origin. Simply add -5 to the *x* value and -5 to the *y* value. This moves the center of rotation to the origin. Now we add the same values to the point (4, 1), which gives us the new point (-1, -4). Rotating this point by π radians, we get the point (1, 4). (A simple geometric argument should convince you why this is so). Lastly, we undo our translations by adding 5 to the *x* value and 5 to the *y* value of both the center of rotation and the rotated point. We find the rotated point is (6, 9), which looks about right according to Figure 5.11.

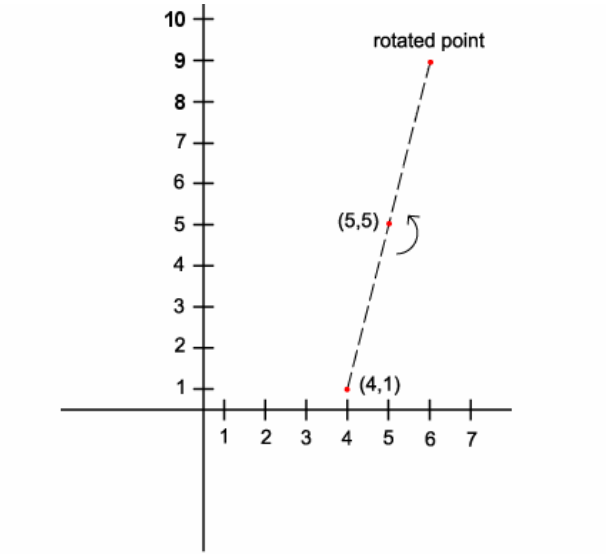


Figure 5.11: Rotating a point around another point.

In two dimensions, rotation is pretty straightforward. In three dimensions, however, there are actually three different ways you can rotate a point. Specifically, you can rotate the point around any one of the three axes. The axis you rotate the point around is called the *axis of rotation*.

What does it mean to rotate something around an axis? Imagine thrusting an axis through some object, and twirling the object around the axis (Fig 5.12.)

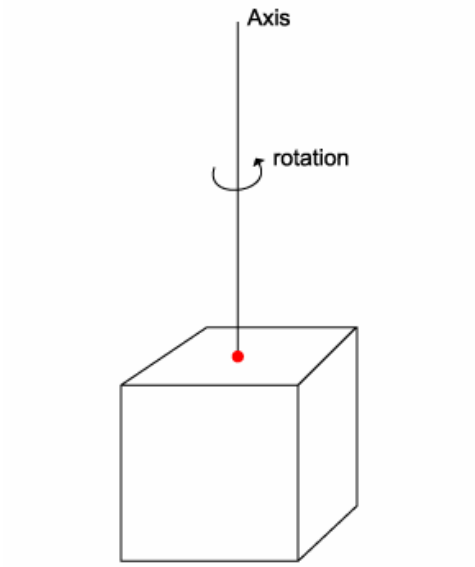


Figure 5.12: Rotating an object around an axis in three dimensions.

Now let us discuss the math behind rotating a point. We will derive equations for the case of rotation around the *z*-axis, which effectively makes this a two-dimensional problem. (If the point has a *z* component, it does not change upon rotation around the *z*-axis.) Figure 5.13 depicts a 2D point located at (x, y) in Cartesian coordinates and (r, θ) in polar coordinates. Rotating this point by ϕ radians, we will end up with a new point located at (x', y') ("*x* prime, *y* prime") in Cartesian coordinates and $(r, \theta + \phi)$ in polar coordinates. What we need to determine is what x' and y' are -- everything else we know.

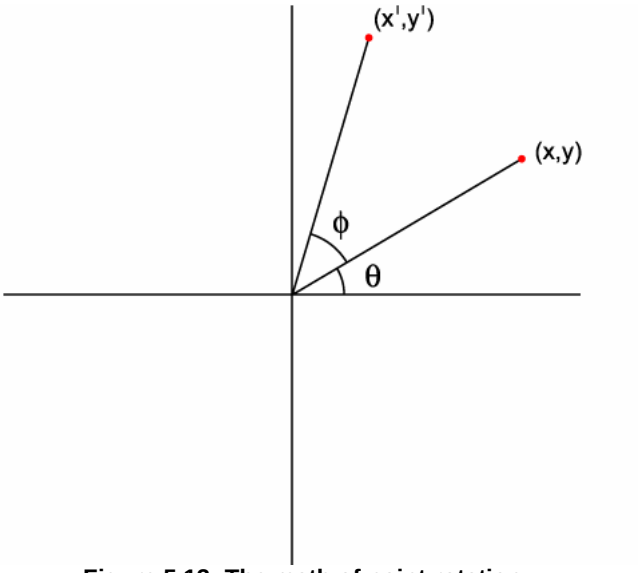


Figure 5.13: The math of point rotation.

One way to do this would be to simply convert the point $(r, \theta + \phi)$ into Cartesian coordinates. But this is not a good idea, since games typically represent points with Cartesian coordinates and *not* polar coordinates. (This will be discussed in greater depth in Chapter Seven.) So in order for us to rotate a point in this way, we would have to convert from Cartesian coordinates to polar coordinates (to get (r, θ)), perform an addition (to get $(r, \theta + \phi)$), and then convert from polar coordinates back to Cartesian coordinates. This is extremely complex -- involving multiplications, additions, trig functions, comparisons, and inverse trig functions -- so let us see if we can come up with a better way.

By definition of sine and cosine, we know that:

Equation 1: $\cos\theta = \frac{x}{r}$

Equation 2: $\sin\theta = \frac{y}{r}$

Equation 3: $\cos(\theta + \phi) = \frac{x'}{r}$

Equation 4: $\sin(\theta + \phi) = \frac{y'}{r}$

Solving equations (3) and (4) for *x'* and *y'*, respectively, we find that,

Equation 5: $x' = r\cos(\theta + \phi)$

Equation 6: $y' = r\sin(\theta + \phi)$

This does tell us what *x'* and *y'* are, but not in a way that saves us from the complex Cartesian-to-polar and polar-to-Cartesian conversions.

Fortunately, the sum identities allow us to write equations (5) and (6) as follows:

Equation 7: $x' = r(\cos\theta\cos\phi - \sin\theta\sin\phi)$

Equation 8: $y' = r(\sin\theta\cos\phi + \cos\theta\sin\phi)$

This is good news, since we can plug equations (1) and (2) in to equations (7) and (8), giving us the following two vastly simplified formulas:

Equation 9: $x' = r(\frac{x}{r}\cos\phi - \frac{y}{r}\sin\phi) = r\left\{\frac{1}{r}(x\cos\phi - y\sin\phi)\right\} = x\cos\phi - y\sin\phi$

Equation 10: $y' = r(\frac{y}{r}\cos\phi + \frac{x}{r}\sin\phi) = r\left\{\frac{1}{r}(y\cos\phi + x\sin\phi)\right\} = y\cos\phi + x\sin\phi$

Notice that the final equations do not involve converting between coordinate systems. They require just four multiplications, two additions, and two trigonometric functions evaluated at the angle ϕ.

You will find equations (9) and (10) in just about every graphics and 3D game development book out there (as well as in a number of mathematical books). Now you can derive these legendary equations for yourself.

Rotating around the other two axes is easy: you can re-derive the equations or just substitute *z* for either *x* or *y* in the above equations, depending on whether you want to rotate around the *x* or *y* axes, respectively.

5.4.2 Field-of-View Calculations

In our last lesson we discussed the equations for projecting 3D geometry onto a two-dimensional screen. These equations relied on a factor *d*, the distance from the viewer to the screen. Recall that this factor determines how much of the world the viewer sees: large *d*'s correspond with narrow views of the world, and small *d*'s correspond with wide views of the world.

We can be a lot more precise: we can exactly relate any *d* value to the viewer's *field-of-view* (an angular measurement that describes how much of the world the viewer sees). The concept of a field-of-view is shown in Figure 5.14.

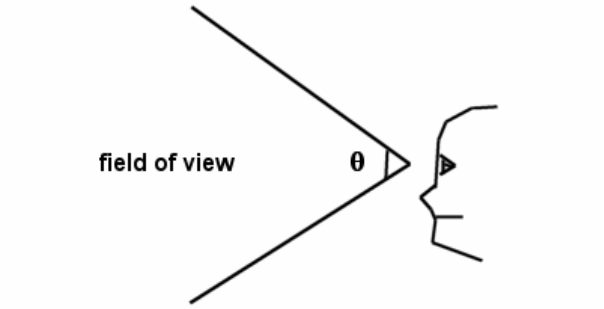


Figure 5.14: The viewer's field-of-view.

The math behind this concept is not outrageously complex, although it does involve something previously unavailable to us: an inverse trig function. Figure 5.15 shows the setup of the problem.

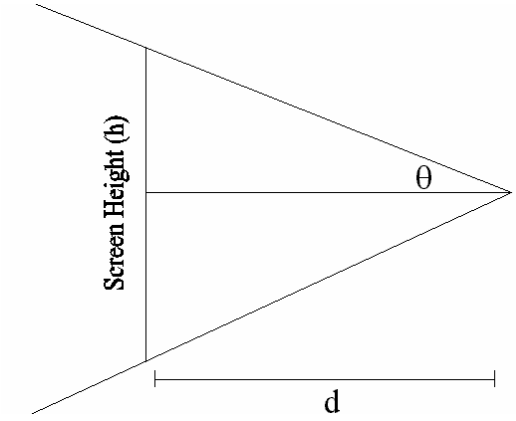


Figure 5.15: Relating *d* to the field-of-view.

The height of the screen (in pixels) is *h*, so we can write $\tan(\theta) = \frac{h/2}{d}$, where θ is *half* the field of view. Solving for θ and multiplying by two, we find the field of view is $2\tan^{-1}\left(\frac{h/2}{d}\right)$.

Conclusion

In our next lesson, we leave the realm of trigonometry (we will not be studying trig exclusively anymore, although it will come up quite frequently in future calculations) and delve into analytic geometry. Analytic geometry is a field of mathematics that gives geometric shapes algebraic representations and uses these representations to answer questions about the underlying geometry. Analytic geometry lies at the heart of collision detection, shadows, reflections, and many other features in today's games.

Exercises

1. Fill in the missing information about this triangle:

Hypotenuse: 12 Adjacent: 3 Opposite: Angle:

2. Fill in the missing information about this triangle:

Hypotenuse: Adjacent: 3 Opposite: 5 Angle:

3. Fill in the missing information about this triangle:

Hypotenuse: 4 Adjacent: Opposite: 2 Angle:

4. Prove Identity 5.3.

*5. Prove Identities 5.8 and 5.9.

6. Derive formulae to rotate any point around the point (*x*, *y*), using the translation trick mentioned in this lesson's material.

! 7. One problem in designing intelligent computer opponents in action games is determining if an enemy can see the player, which in turn determines if the enemy attacks. Suppose the position of the player is (*x*, *y*) and the position of an enemy is (*s*, *t*). Further, suppose the enemy's field of view is 2β, and that the angle the enemy's forward line of sight makes with the positive *x*-axis is θ. Derive equations to determine if the enemy can see the player. (You can also use this technique to optimize the performance of 3D games by only displaying the objects the player can see.)

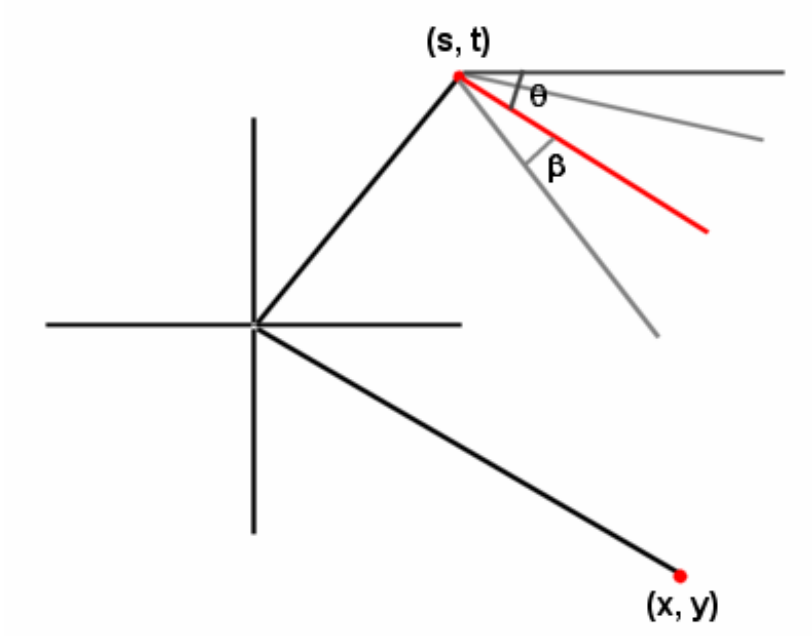


Figure E5.1: The illustration for exercise 7.

*8. Suppose the player has a horizontal field of view of *h* radians, and a vertical field of view of *v* radians (see Figure E5.2). Determine the angle of a cone that completely encompasses what the player sees. (This information can be used to quickly and easily determine which objects in a 3D world are visible to the player.)

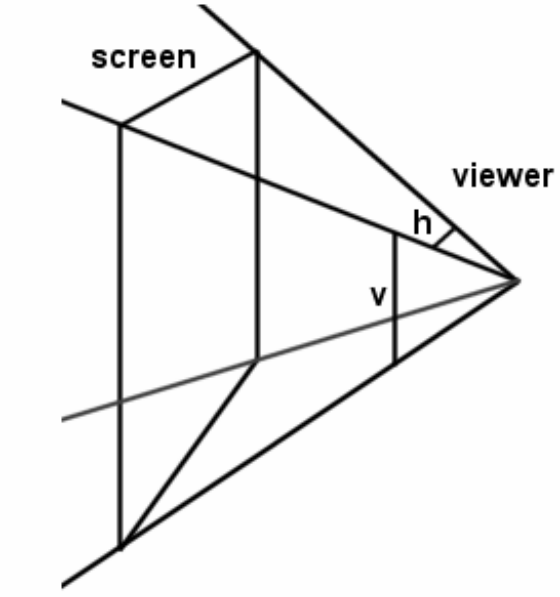
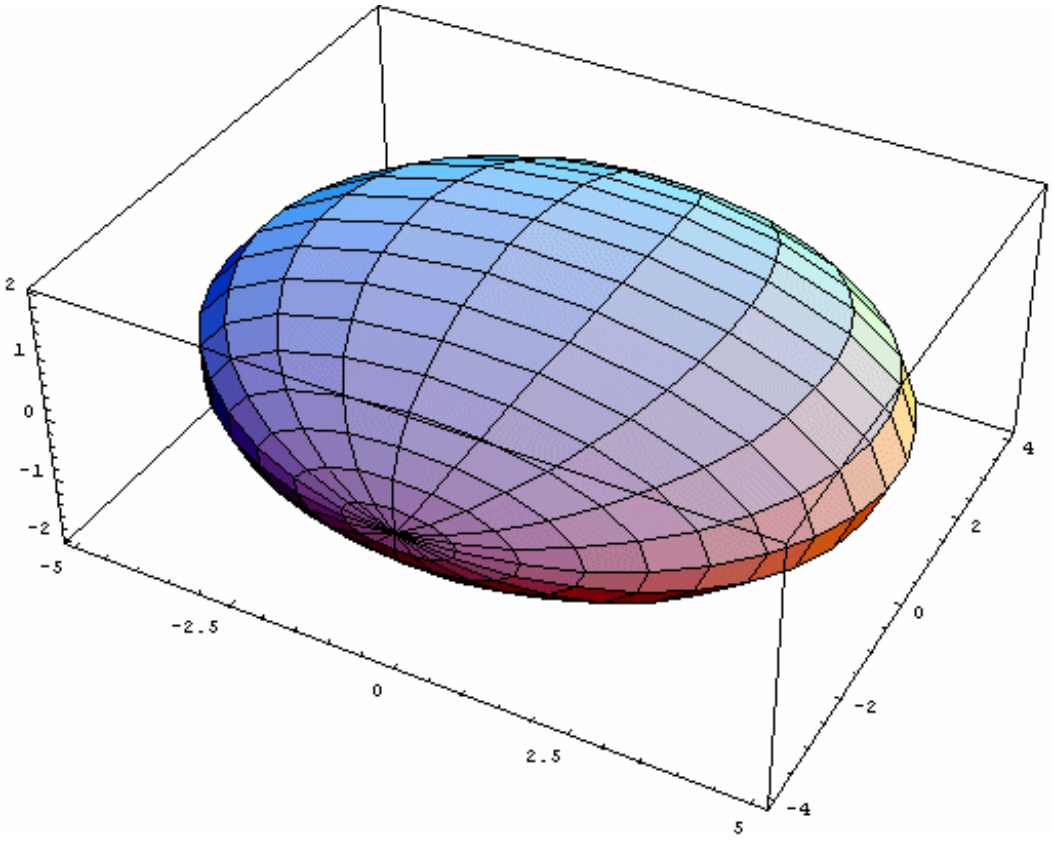


Figure E5.2: The illustration for exercise 8.

9. Derive formulas for rotation around the *x*- and *y*-axes.

Chapter Six

Analytic Geometry I



Introduction

When you think of geometry, you probably remember the hundreds of tedious theorems that your high school teacher carefully derived (no matter how intuitive or obvious they seemed). This kind of geometry is useful, especially when coming up with trigonometric expressions. But for the most part, game programmers tend to rely on a different kind of geometry -- one termed *analytic geometry*.

Analytic geometry takes a whole new approach to geometric problems. Instead of trying to deduce geometric solutions from theorems (which computers are not very good at doing), analytic geometry tries to *calculate* them. How? By giving geometric shapes equations, and then manipulating those equations according to the standard rules of algebra to produce the desired results.

In this chapter, you will be introduced to some very basic topics in analytical geometry. We will discuss how you can represent points, lines, planes, ellipses, and ellipsoids, and then discuss some interesting things that you can do with their mathematical representations.

6.1 Points

A point in an *n* dimensional coordinate system is just an ordered list of *n* numbers -- in other words, an *n*-tuple. For the Cartesian coordinate system, these points tell you how far the point is along each axis.

One of the only things you can do with points is to calculate the distance between them. For one-dimensional points, the distance between a point x_0 and a point x_1 is given by the following equation:

$$d = \sqrt{(x_1 - x_0)^2} = | x_1 - x_0 |$$

If the points are two-dimensional, the problem is a bit more interesting. In this case, all we have to do is construct a right triangle whose hypotenuse joins the points, as shown in Figure 6.1.

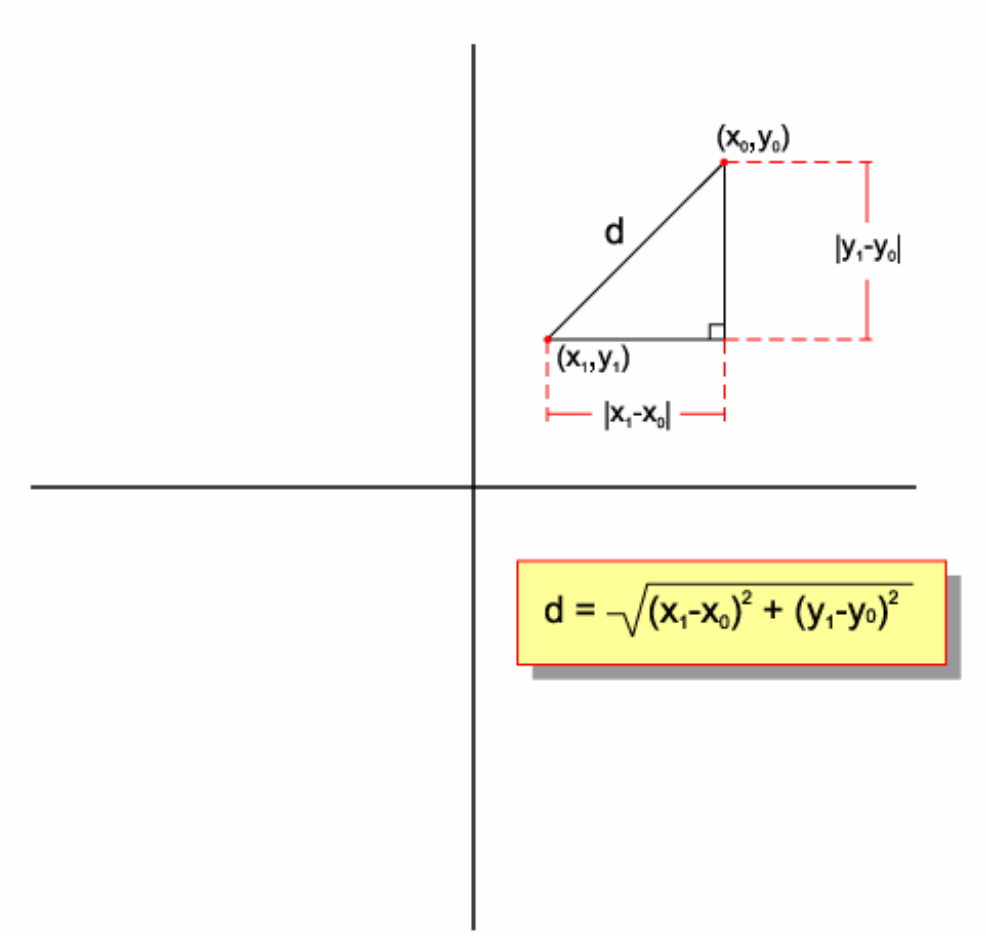


Figure 6.1: A right triangle whose hypotenuse connects two points of interest.

To calculate the length of the hypotenuse, we could use trigonometry, but that is not a good idea since we would have to know the angle of the triangle. A better approach is to simply use the Pythagorean Theorem, which tells us that the length of the hypotenuse is equal to the square root of the sums of the squares of the lengths of the sides. In our case, the lengths of the sides are $|x_1 - x_0|$ and $|y_1 - y_0|$. We take the absolute value of these quantities, because negative numbers cannot be lengths. Consequently, the distance between the points is given by the following equation:

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Notice that the absolute value signs were dropped because the square of a number is always positive.

The three-dimensional case might seem much harder, but actually, is extremely similar to -- and borrows from -- the two-dimensional case.

Much like Figure 6.1, Figure 6.2 shows a right triangle set in three-dimensional space whose hypotenuse connects one 3D point to another.

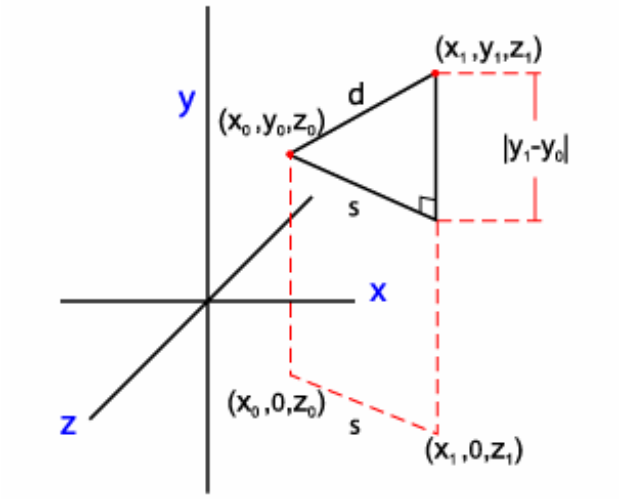


Figure 6.2: A right triangle whose hypotenuse connects two points of interest.

As the figure suggests, we can use the Pythagorean Theorem to yield the following result:

$$d = \sqrt{s^2 + (y_1 - y_0)^2}$$

s is the length of the base of the triangle. How do we calculate *s*? One way to do it is to calculate the distance from the point (x_0, y_0, z_0) to the point (x_1, y_0, z_1). Notice that since the *y* coordinate is the same in these points, this is the same as calculating the distance separating the *two-dimensional* points (x_0, z_0) and (x_1, z_1). This gives us the following value for *s*:

$$s = \sqrt{(x_1 - x_0) + (z_1 - z_0)}$$

Substituting this into our equation for *d*, we get the following equation:

$$d = \sqrt{\left(\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}\right)^2 + (z_1 - z_0)^2} = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

As we can see, the equation looks remarkably similar to the two-dimensional case. You can probably guess how the equation extends to handle points in any number of dimensions.

6.2 Lines

From Chapter Three, you know that a binomial produces the graph of a two-dimensional line. Indeed, all mathematical representations of a line are binomials. In the sections that follow, we will discuss the many ways that you can use a binomial to represent two-dimensional lines, as well as the techniques for representing three-dimensional lines. We will also learn how these representations enable us to solve an array of geometric problems.

6.2.1 Two-Dimensional Lines

The simplest kind of line to represent is a straight horizontal line. For such a line, the *x* value is free to be any real number, but the *y* value is fixed to a certain number, which determines where on the *y*-axis the line appears. The equation for this representation is:

$$y = b$$

A straight vertical line is very similar, except this time, the *y* value is free to be any real number and the *x* value is fixed. The equation for such a line is:

$$x = a$$

This is a perfectly good way to mathematically express a vertical line, although you should keep in mind that the above equation is *not* a function from the real numbers on the *x*-axis to the real numbers on the *y*-axis. If it were, the "function" would associate an infinite number of elements in the range with a single element in the range, which a function cannot do. However, we can certainly think of the equation as a function from the real numbers on the *y*-axis to the real numbers on the *x*-axis.

The most basic equation for a line that can have nearly any orientation and position is called the *slope-intercept form*. This equation relates the *y* coordinate of a point on the line to an *x* coordinate on the line. The slope-intercept form of the equation of a line is usually written as follows:

$$y = mx + b$$

We can see that the expression on the right is a binomial, so we know that the graph of the equation will be a line. The variables *m* and *b* are constants that dictate the exact form of the line: *m* is called the *slope* of the line and *b* is called the *y*-intercept (for reasons which will become clear shortly).

So how do we interpret *m* and *b*? Let us consider two points that lie on the line, which we will call (x_0, y_0) and (x_1, y_1). Then, using the slope-intercept equation, we know that the following two equations hold true:

$$y_0 = mx_0 + b$$
$$y_1 = mx_1 + b$$

Solving the top equation for *b*, we find that $b = y_0 - mx_0$. Plugging this result into the bottom equation, we get the following derived equation:

$$y_1 = mx_1 + y_0 - mx_0 = m(x_1 - x_0) + y_0$$

All we have to do now is solve this equation for *m*, the slope of the line:

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

To understand the significance of this result, take a look at Figure 6.3.

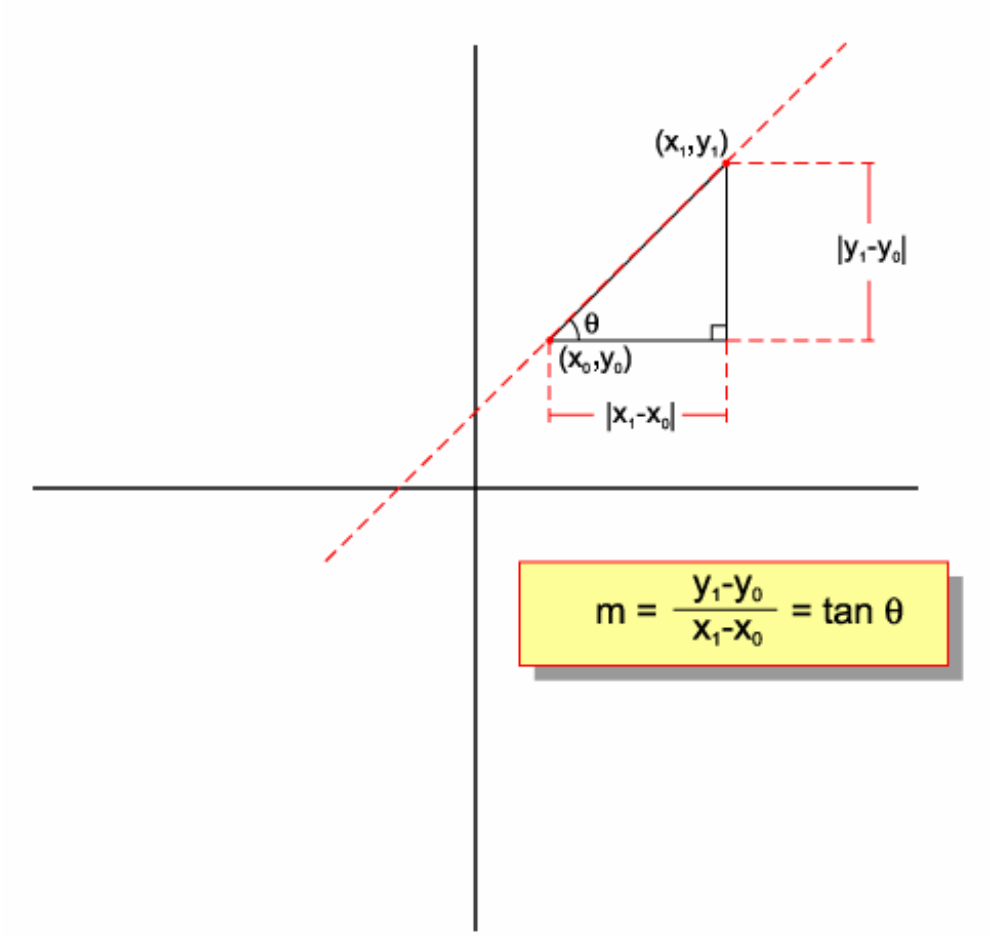


Figure 6.3: A right triangle whose hypotenuse connects two points on a line.

This figure shows a circle centered on the point (x_0, y_0) and extending out to the point (x_1, y_1). From the definition of the tangent function, we can see that *m* is simply the *tangent of the angle that the line makes with the horizontal x*-axis. In other words:

$$m = \frac{y_1 - y_0}{x_1 - x_0} = \tan\theta \qquad 0 \le \theta < \pi$$

This is the result we want, since it tells us that *m* measures the orientation of the line. If we picture the line as a hill, then we can also say that *m* measures the *slope* of the hill, which is, in fact, why *m* is *called* the slope of the line.

Note that θ is defined to be in the interval $[0, \pi)$.This allows us to say that if two lines have equal slopes, they have equal angles. We would not be able to say this otherwise, since, for example, it is possible that the angle of one line could be 0 and the angle of another line could be 2π, and the slopes of these lines and their orientations are equal, even though the angles themselves are different.

Figuring out what *b* does is much simpler. When $x = 0$, the slope-intercept equation tells us that $y = b$. This means that *b* tells us what the *y* value of the line is at the place where the line crosses the *y*-axis; hence, *b* is usually called the *y*-intercept.

Figure 6.4 nicely summarizes our discussion of the slope-intercept equation in a visual way.

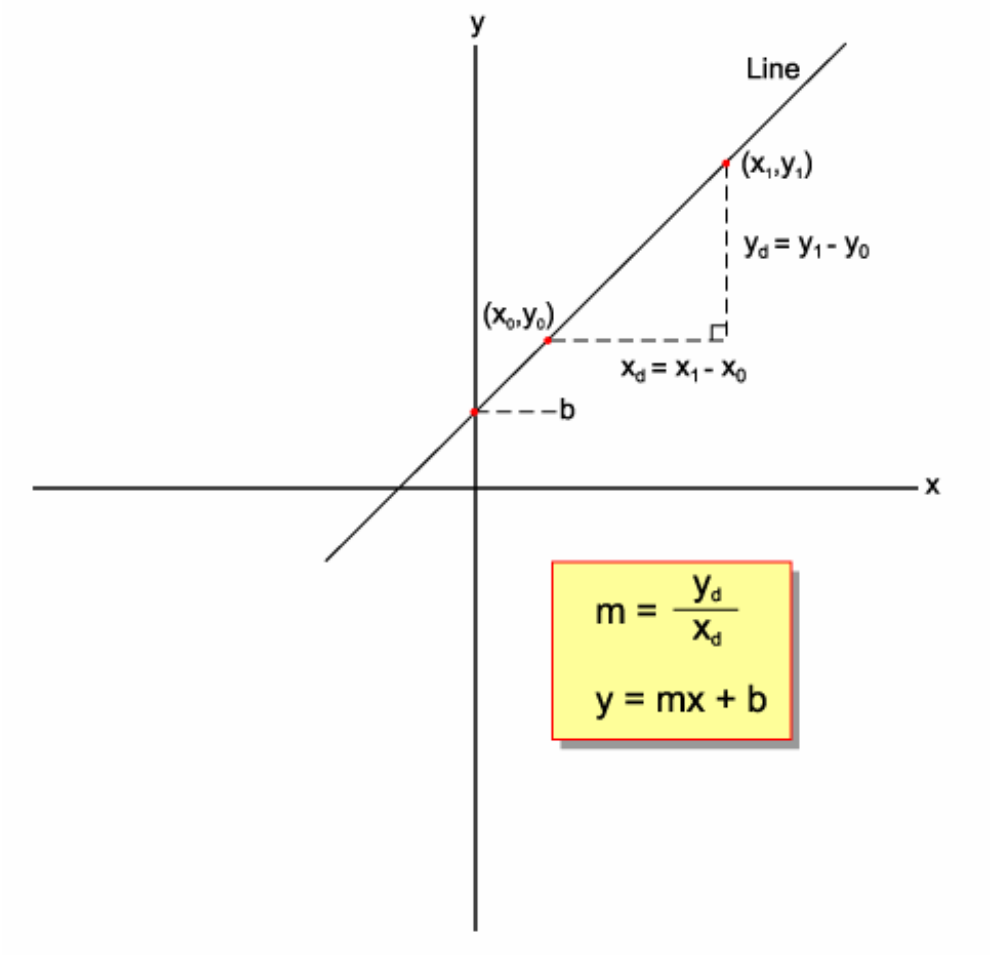


Figure 6.4: The slope-intercept form of the equation of a line.

Another line equation often used is called the *point-slope form of the equation of a line*. This equation relates one point on a line, designated (x, y), to another point, designated (x_0, y_0).

$$y - y_0 = m(x - x_0)$$

This form is actually one of the intermediate equations we derived in trying to figure out what the interpretation of *m* and *b* are in the slope-intercept equation.

A problem with both the slope-intercept and point-slope forms is that they cannot represent perfectly vertical lines, since in a perfectly vertical line, the numerator of the slope would be infinite and the denominator would be zero. One solution is to simply replace all the *x*'s with *y*'s, and vice-versa, in all of the above equations. Then we will have a second set of equations which map from a given *y* value to an *x* value, and thus, allow us to represent perfectly vertical lines. But this second set cannot handle perfectly *horizontal* lines like the first set can, so we really will not have gained much by doing this.

The solution is the *general form* of the equation of a line, shown below:

$$ax + by + c = 0$$

If the line is horizontal, then *a* will be non-zero and *b* will be zero. We can then solve for *x*. If the line is vertical, then *a* will be zero and *b* will be non-zero. We can then solve for *y*. If the line is neither horizontal nor vertical, then neither *a* nor *b* will be zero, and we can solve for whatever we want to. If we solve for *x*, we have a function from the real numbers on the *y*-axis to the real numbers on the *x*-axis. And if we solve for *y*, we have a function from the real numbers on the *x*-axis to the real numbers on the *y*-axis.

We can actually derive all other forms of the equation of a line from the general form -- even $y = b$ and $x = a$. However, in practice, game developers usually use the slope-intercept equation.

6.2.2 Parametric Representation

The last way to represent a line is also the most powerful. It is called the *parametric* form of the equation of a line, since the coordinates of the point rely on a *parameter*. This parameter is free to be any real number; each choice of a parameter results in a different point on the line.

The parametric equations, which are defined in terms of two points on the line called (x_0, y_0) and (x_1, y_1), are shown below:

$$x = t(x_1 - x_0) + x_0$$
$$y = t(y_1 - y_0) + y_0$$

Notice the parametric form does not relate *x* to *y* or *y* to *x* (at least directly), but rather, it relates both of these to the parameter. This form can express any possible line, regardless of orientation or position, and it does not require us to first calculate the slope of the line.

It is fairly easy to derive the slope-intercept equation from the parametric equations. First, we just solve the *x* equation for *t*, giving us the following relationship:

$$t = \frac{x - x_0}{x_1 - x_0}$$

Next, we substitute this into the equation for *y*, and simplify, as shown below:

$$y = \frac{x - x_0}{x_1 - x_0}(y_1 - y_0) + y_0 =$$

$$(x - x_0)\frac{y_1 - y_0}{x_1 - x_0} + y_0 =$$

$$x\left(\frac{y_1 - y_0}{x_1 - x_0}\right) - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right) + y_0 =$$

$$\left(\frac{y_1 - y_0}{x_1 - x_0}\right)x + \left\{y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)\right\} =$$

$$mx + b$$

Notice that *b* was substituted in place of $y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$. How can we do this? As illustrated in Figure 6.5, we can see that *b* is just $y_0 - o$, and *o*, in turn, is $x_0 \tan\theta = x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$, so *b* is $y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$.

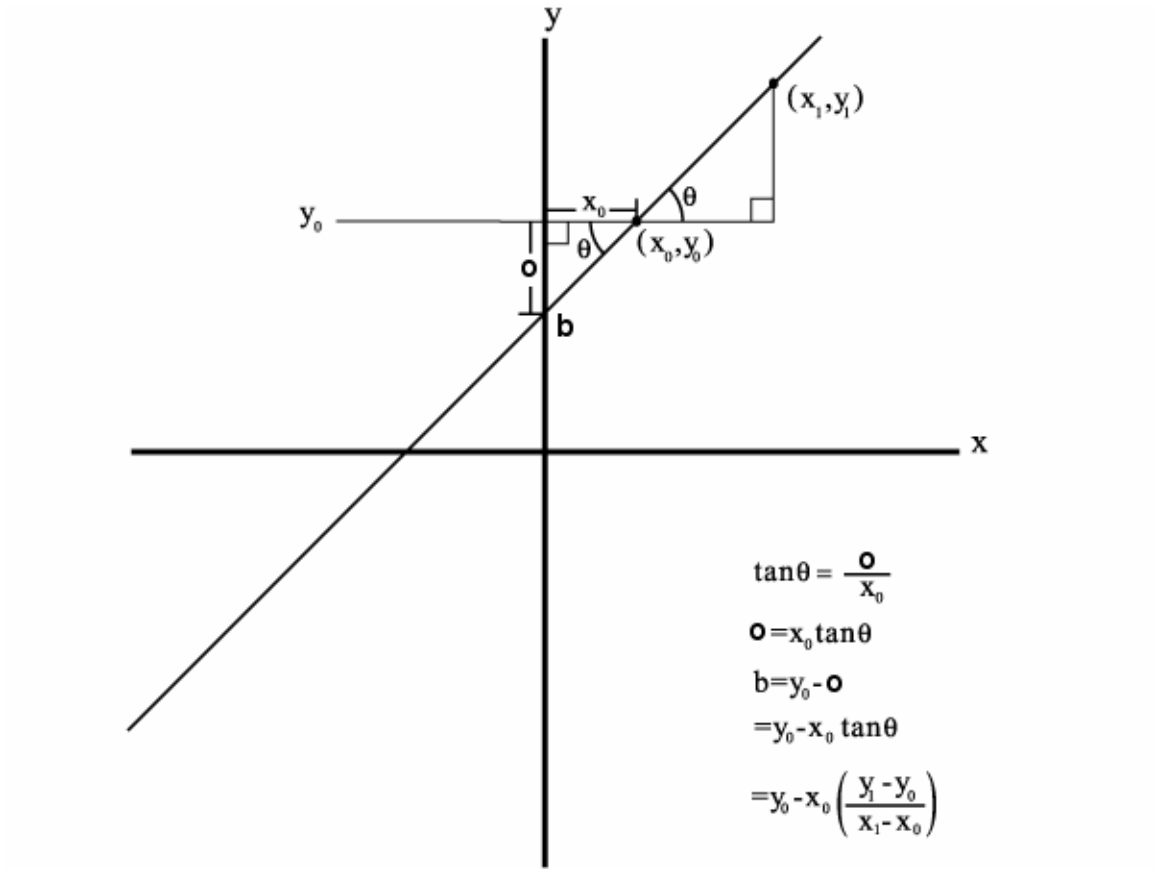


Figure 6.5: A graphical proof that *b* is $y_0 - x_0\left(\frac{y_1 - y_0}{x_1 - x_0}\right)$.

That completes the proof. Now we turn to looking at the analytic operations we can do with lines.

6.2.3 Parallel and Perpendicular Lines

Sometimes we want to know if two lines are parallel or perpendicular to each other. Fortunately, there is an easy way to do this, based on the definition of the slope of a line.

Suppose we have two lines with the slopes $m_0 = \tan\theta_0$ and $m_1 = \tan\theta_1$. If the two lines are parallel, then they have the same angles. Consequently, their tangents are equal. This tells us that *if two lines are parallel to each other, their slopes are equal.*

Perpendicular lines are a bit trickier. Suppose we have two lines with slopes $m_0 = \tan\theta$ and $m_1 = \tan(\theta + \pi/2)$. It is a trigonometric fact (which we discussed in our last lesson) that $\tan(\theta + \pi/2) = -\cot\theta$, so we can rewrite m_1 as $m_1 = -\cot\theta = -\frac{1}{\tan\theta} = -\frac{1}{m_0}$. Thus, we see that if two lines are perpendicular, then the slope of one line is the negative reciprocal of the slope of the other line. This result is more important than the last, since there are many instances where we will need to create a line that is perpendicular to another, and this equation will allow us to do it. We will see one such instance later on in this lesson, when we calculate the distance from a point to a line.

6.2.4 Intersection of Two Lines

An extremely useful tool in analytic geometry is determining the *intersection* of shapes -- in this case, the intersection of two lines. Here, the word "intersection" carries its set theoretic meaning. Two geometric shapes can be thought of as two sets of points. The intersection of these sets, recall, is a set that contains every point in *both* the sets, and no other points.

In the case of lines, as long as two lines are not parallel, they will intersect at exactly one place. To determine where this is, we should first define the two lines using the slope-intercept equation, as shown below:

$$y_0 = m_0 x_0 + b_0$$
$$y_1 = m_1 x_1 + b_1$$

Each of these equations determines a set of points. The first line determines the set $L_0 = \{(x_0, y_0) \mid x_0 \in \Re, y_0 = m_0 x_0 + b_0\}$, and the second line, the set $L_1 = \{(x_1, y_1) \mid x_1 \in \Re, y_1 = m_1 x_1 + b_1\}$.

The intersection of these sets, then, is the set $I = L_0 \cap L_1 = \{(x, y) \mid (x, y) \in L_0, (x, y) \in L_1\}$. We can expand this as $I = \{(x, y) \mid (x \in \Re, y = m_0 x + b), (x \in \Re, y = m_1 x + b)\}$, but since this expansion contains redundant information (we do not need to be told that $x \in \Re$ twice), we can simplify the expression to $I = \{(x, y) \mid x \in \Re, y = m_0 x + b, y = m_1 x + b\}$.

This tells us what all the points in *I* will look like: the first component will be a real number, and the second component will be related to the first in the way specified by the two equations. Actually, the first constraint is no longer helpful: while *x* is a real number, it is not just *any* real number. We have two equations involving *x*, so *x* is forced to be a *specific* real number. Thus, it is sufficient to write the intersection of the sets as $I = \{(x, y) \mid y = m_0 x + b, y = m_1 x + b\}$.

So what does this intersection point look like? By definition of the intersection set, we know that $y = m_0 x + b$ and $y = m_1 x + b$. By deduction, therefore, we also know that $m_0 x + b_0 = m_1 x + b_1$. This is good, since it is an equation with only one unknown -- *x*. We can get an analytic expression for *x*.

The first step is to move all the terms involving *x* to the left side, and all the other terms to the right, leaving us with the equation $m_0 x - m_1 x = b_1 - b_0$. Next we factor out an *x*, giving us $x(m_0 - m_1) = b_1 - b_0$. Finally, we divide both sides by *x*, giving us $x = \frac{b_1 - b_0}{m_0 - m_1}$.

Now that we know what x is, it is simple enough to find out what *y* is; simply plug our known value of *x* into any one of the line equations (either one will work since, at the intersection point, they both have the same *y* values). This step (and resulting simplification) is shown below:

$$y = m_0 x + b_0 = m_0\left(\frac{b_1 - b_0}{m_0 - m_1}\right) + b_0 = \frac{m_0 b_1 - m_0 b_0}{m_0 - m_1} + b_0 \frac{(m_0 - m_1)}{(m_0 - m_1)} =$$

$$\frac{m_0 b_1 - m_0 b_0}{m_0 - m_1} + \frac{b_0 m_0 - b_0 m_1}{m_0 - m_1} = \frac{m_0 b_1 - m_0 b_0 + b_0 m_0 - b_0 m_1}{m_0 - m_1} = \frac{m_0 b_1 - b_0 m_1}{m_0 - m_1}$$

Thus, with very little effort, we have determined the intersection point is $(\frac{b_1 - b_0}{m_0 - m_1}, \frac{m_0 b_1 - b_0 m_1}{m_0 - m_1})$. Notice what happens if the slopes are equal and the lines are therefore parallel -- a division by zero, which is undefined. Mathematics has a way of "encoding" the "no solution" case as a division by zero (or sometimes, as the square root of a negative number).

6.2.5 Distance from a Point to a Line

One extremely useful application of analytic geometry is calculating the distance from a point to a line. By *distance*, it refers to the *shortest* distance from the point to the line. This is shown in Figure 6.6.

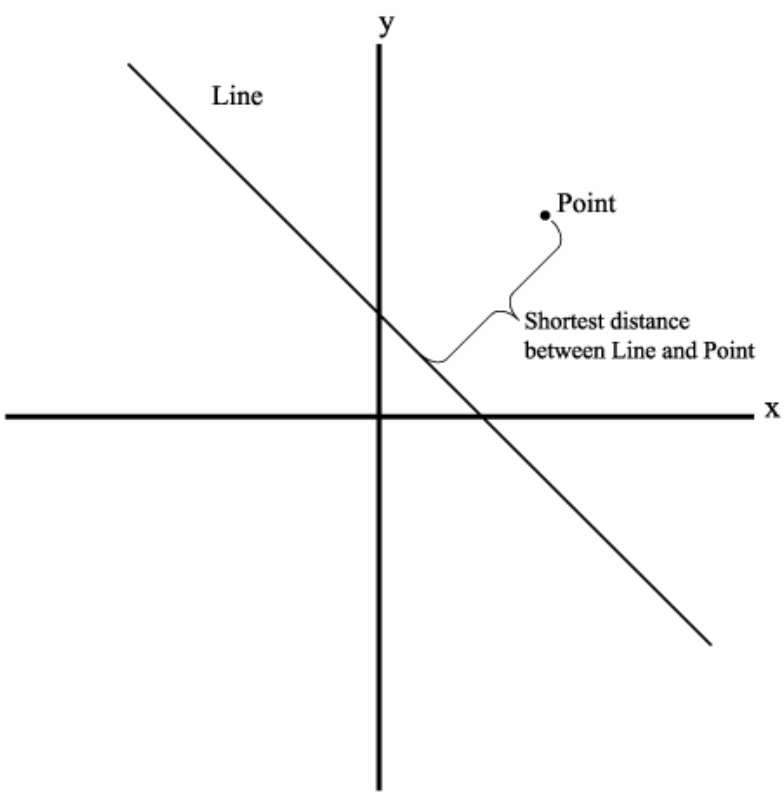


Figure 6.6: The shortest distance between a point and a line.

With a little thought, we will soon come to the conclusion that what we need to solve this problem is *another* line, this one, *perpendicular to the first*. The perpendicular line should pass through the point. The distance from the point to wherever the perpendicular line intersects the other line is the distance we seek. Figure 6.7 illustrates this.

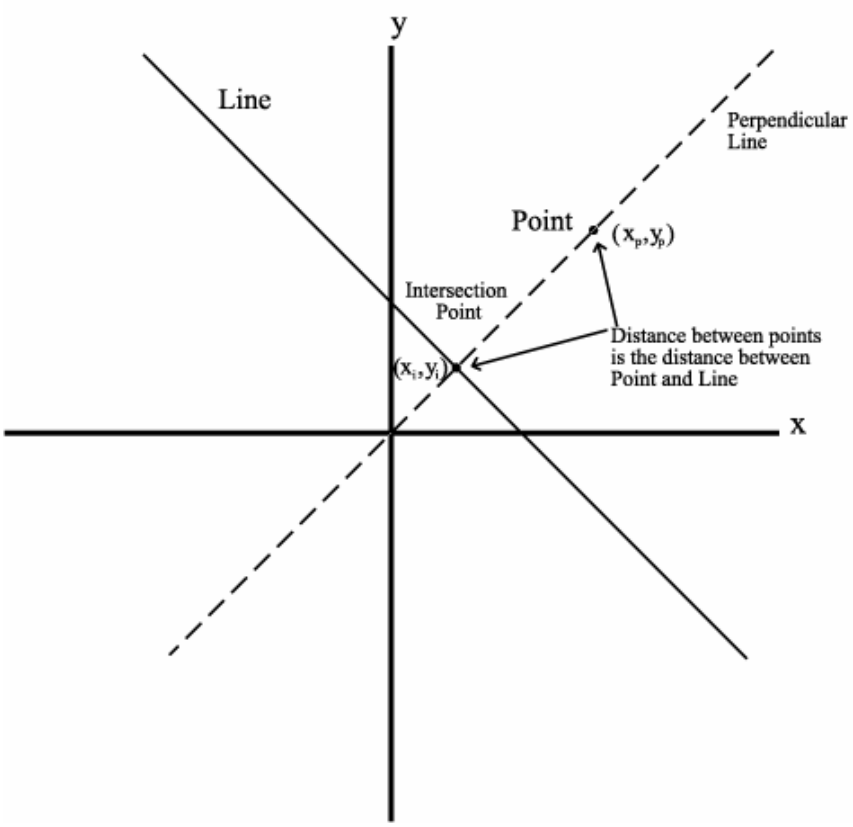


Figure 6.7: A perpendicular line can be used to calculate the shortest distance between a line and a point.

The first step in solving the problem is to describe each line with an equation:

$$y_0 = m_0 x_0 + b_0$$
$$y_1 = m_1 x_1 + b_1$$

We know both the slope and the *y*-intercept of the first line. The perpendicular line, designated y_1, has a slope equal to the negative reciprocal of the first line (which we covered earlier). We can determine what b_1 is by recognizing that the line must pass through the point whose distance we are computing. If we call this point (x_p, y_p), then we know that the perpendicular line must satisfy the following equation:

$$y_p = m_1 x_p + b_1$$

Solving for b_1, we find that,

$$b_1 = y_p - m_1 x_p = y_p - \left(-\frac{1}{m_0} \right) x_p = y_p + \frac{x_p}{m_0}$$

We now have the equations of both lines. What we need to do next is determine where the lines intersect. Fortunately, we just did this in the last section, so we can reuse those results here.

Let us call the point where the lines intersect (x_i, y_i). Then, from the intersection equations, we know that,

$$x_i = \frac{b_1 - b_0}{m_0 - m_1} = \frac{\left(y_p + \frac{x_P}{m_0}\right) - b_0}{m_0 - (-\frac{1}{m_0})} =$$

$$\frac{\frac{x_p + m_0 y_p - m_0 b_0}{m_0}}{\frac{m_0^2 + 1}{m_0}} =$$

$$\frac{x_p + m_0(y_p - b_0)}{m_0} \frac{m_0}{m_0^2 + 1} =$$

$$\frac{x_p + m_0(y_p - b_0)}{m_0^2 + 1}$$

Similarly, we know that y_i is given by the following expression:

$$y_i = \frac{m_0 b_1 - b_0 m_1}{m_0 - m_1} = \frac{m_0\left(y_p + \frac{x_p}{m_0}\right) - b_0(-\frac{1}{m_0})}{m_0 - (-\frac{1}{m_0})} =$$

$$\frac{\left(\frac{m_0 x_p + m_0^2 y_p}{m_0}\right) + \frac{b_0}{m_0}}{\frac{m_0^2 + 1}{m_0}} =$$

$$\frac{\frac{m_0 x_p + m_0^2 y_p + b_0}{m_0}}{\frac{m_0^2 + 1}{m_0}} =$$

$$\frac{m_0 x_p + m_0^2 y_p + b_0}{m_0} \frac{m_0}{m_0^2 + 1} =$$

$$\frac{m_0(x_p + m_0 y_p) + b_0}{m_0^2 + 1}$$

We are nearly done. All we have to do now is calculate the distance between the points (x_i, y_i) and (x_p, y_p), which we can do using our distance formula::

$$d = \sqrt{(x_p - x_i)^2 + (y_p - y_i)^2} =$$

$$\sqrt{\left\{x_p - \left(\frac{x_p + m_0(y_p - b_0)}{m_0^2 + 1}\right)\right\}^2 + \left\{y_p - \left(\frac{m_0(x_p + m_0 y_p) + b_0}{m_0^2 + 1}\right)\right\}^2} =$$

$$\sqrt{\frac{(b_0 + m_0 x_p - y_p)^2}{m_0^2 + 1}}$$

The equation has been vastly simplified in the third step shown above. We could discuss the math -- but it would be another three pages of computation. You should work it out for yourself, however, to verify the result.

Note that we can take the square root into the numerator to simplify further, but we will have to use the absolute value function, since the *positive* square root of the square of a number is not necessarily equal to that number -- rather, it is equal to the *absolute value* of that number.

6.2.6 Angles between Lines

Another useful thing we can do in analytic geometry is calculate the angle between two lines.

Suppose we have two lines defined as follows:

$$y_0 = m_0 x_0 + b_0$$
$$y_1 = m_1 x_1 + b_1$$

Say the angles of the lines are ϕ_0 and ϕ_1, respectively. Then angle θ between the lines is defined to be the absolute value of the difference between these angles -- i.e. $\theta = |\phi_1 - \phi_0|$. This angle, which happens to be the smaller of the two angles formed by the intersection of the two lines, is shown graphically in Figure 6.8.

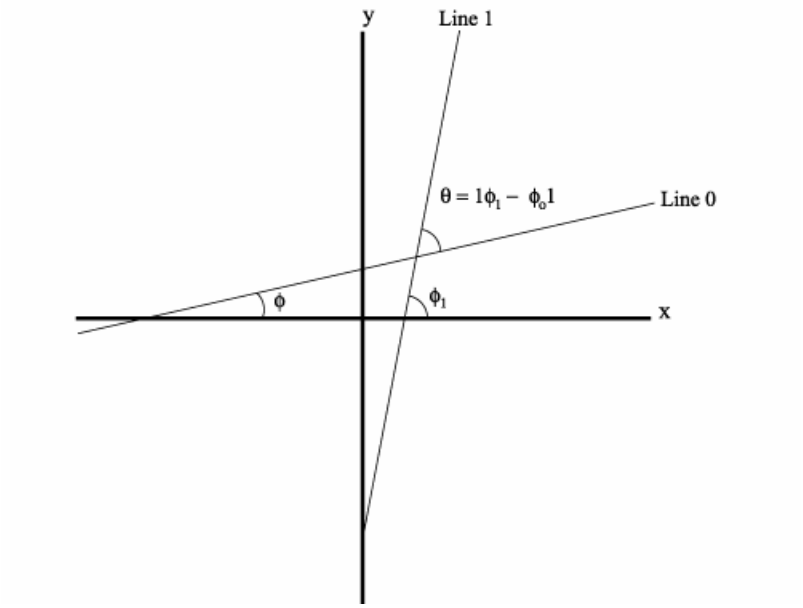


Figure 6.8: The angle between two lines.

We can immediately write down the following expression for the tangent of θ:

$$\tan\theta = \tan(|\phi_1 - \phi_0|)$$

In order to calculate what θ is, we need to use the difference identity for the tangent function (discussed in Chapter Five). But that identity did not tell us how to expand the difference of an absolute value, so what do we do? Recall from that same lesson that $\tan(-\theta) = -\tan\theta$, so we can write $\tan(|\phi_1 - \phi_0|)$ as $|\tan(\phi_1 - \phi_0)|$. This enables us to easily use the difference identity, as shown below:

$$\tan\theta = |\tan(\phi_1 - \phi_0)| = \left|\frac{\tan\phi_1 - \tan\phi_0}{1 + \tan\phi_0 \tan\phi_1}\right|$$

This may not seem like a better form, but actually, it is. Recall that the tangent of the angle of a line is equal to the slope of that line, so we can rewrite the equation as shown below:

$$\tan\theta = \left|\frac{\tan\phi_1 - \tan\phi_0}{1 + \tan\phi_0 \tan\phi_1}\right| = \left|\frac{m_1 - m_0}{1 + m_0 m_1}\right|$$

This gives us a nice, convenient expression for the angle between two lines that relies only on the slope of those lines. Of course, to get the actual angle, we will have to use the inverse tangent function, but since the angle will always be in the interval $[0, \pi/2]$, this is quite trivial.

6.2.7 Three-Dimensional Lines

Lines in the 3rd dimension can be described in a variety of ways, but all can be derived from the *parametric representation* of 3D lines. This form, exactly like the parametric representation of two-dimensional lines, depends on a parameter that can assume any real number. Different real numbers correspond to different points on the 3D line.

The form of this line is shown below:

$$x = t(x_1 - x_0) + x_0$$
$$y = t(y_1 - y_0) + y_0$$
$$z = t(z_1 - z_0) + z_0$$

where (x_0, y_0, z_0) and (x_1, y_1, z_1) are two points on the line and *t* is the parameter.

We can derive from the parametric equation new equations that depend on *x*, *y*, or *z*, according to our preference. If we wanted to know what the *x* and *y* coordinates of the line are for a given *z* value, for example, then we simply solve the *z* equation for *t*, and plug this value of *t* into the other two equations.

Like two-dimensional lines, there are lots of neat tricks we can do with 3D lines as well. But most of these lend themselves to vector mathematics, so we will discuss their coverage later in the course.

6.3 Ellipses

Ellipses generally resemble squashed circles, although they can also look like perfectly round circles. Like lines, they too can be represented mathematically. The usual form of the ellipse equation is shown below:

$$\frac{(x-h)^2}{r_x^{\,2}} + \frac{(y-k)^2}{r_y^{\,2}} = 1$$

Here (h,k) is the point at the center of the ellipse, r_x is half the width of the ellipse, and r_y is half its height. The equation is true for all points (x, y) on the ellipse, and false otherwise. Figure 6.9 shows us an ellipse and how the various parameters shape how the ellipse looks.

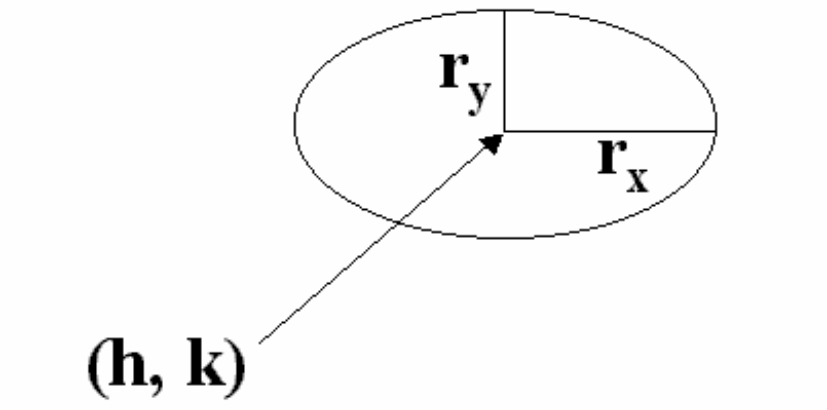


Figure 6.9: An ellipse described by the standard ellipse equation.

We can solve for either *x* or *y* in the ellipse equation, however, we will always get two solutions, one involving a positive square root and the other involving a negative square root. These two equations correspond to the two halves of the ellipse.

If $r_x = r_y$, then the ellipse is actually a circle, and the equation simplifies as follows:

$$(x-h)^2 + (y-k)^2 = r^2$$

As with lines, ellipses can be represented parametrically. This time the parameter is an angle. The angle describes the orientation of a ray that emanates from the center of the ellipse. The ray intersects the point on the ellipse associated with that angle.

The equations for the parametric representation of ellipses are shown below:

$$x = r_x \cos\theta + h$$
$$y = r_y \sin\theta + k$$

If $r_x = r_y$, then as before, the ellipse is actually a circle, and the equation simplifies as follows:

$$x = r\cos\theta + h$$
$$y = r\sin\theta + k$$

6.3.1 Intersecting Lines with Ellipses

Just as we can determine the points in the intersection of two lines, we can also determine the points in the intersection of a line with an ellipse. (Intersecting ellipses do not have that many applications, so we will not discuss them here.)

When a line intersects an ellipse, it will usually intersect it twice -- once as the line enters the ellipse, and once as the line exits the ellipse. For a line and ellipse in standard form, the first intersection point is given by the following equations:

$$x_i = -\frac{bmr_x^2 + r_x r_y \sqrt{-b^2 + m^2 r_x^2 + r_y^2}}{m^2 r_x^2 + r_y^2}$$

$$y_i = \frac{br_y^2 - mr_x r_y \sqrt{-b^2 + m^2 r_x^2 + r_y^2}}{m^2 r_x^2 + r_y^2}$$

The second intersection point is given by the following equation:

$$x_i = \frac{-bmr_x^2 + r_x r_y \sqrt{-b^2 + m^2 r_x^2 + r_y^2}}{m^2 r_x^2 + r_y^2}$$

$$y_i = \frac{br_y^2 + mr_x r_y \sqrt{-b^2 + m^2 r_x^2 + r_y^2}}{m^2 r_x^2 + r_y^2}$$

If the line just grazes the ellipse, then it will intersect it only once, and both of the above sets of equations will produce the same point. If the line does not intersect the ellipse at all, then the *discriminant* -- that expression inside the square root operator -- will be negative, and as we all know, there is no real number such that, when multiplied by itself, is equal to a negative number.

If the ellipse is a circle, then the equations simplify as follows:

$$x_i = -\frac{bm + \sqrt{-b^2 + r^2(m^2 + 1)}}{m^2 + 1}$$

$$y_i = \frac{b - m\sqrt{-b^2 + r^2(m^2 + 1)}}{m^2 + 1}$$

And for the second intersection point:

$$x_i = \frac{-bm + \sqrt{-b^2 + r^2(m^2 + 1)}}{m^2 + 1}$$

$$y_i = \frac{b + m\sqrt{-b^2 + r^2(m^2 + 1)}}{m^2 + 1}$$

6.4 Ellipsoids

Ellipsoids are three-dimensional versions of ellipses. Essentially, they are spheres that have been compressed or expanded on each of the three axes. An example of an ellipsoid is shown in Figure 6.10.

Note: a sphere is a special case of an ellipsoid, like a circle is a special case of an ellipse.

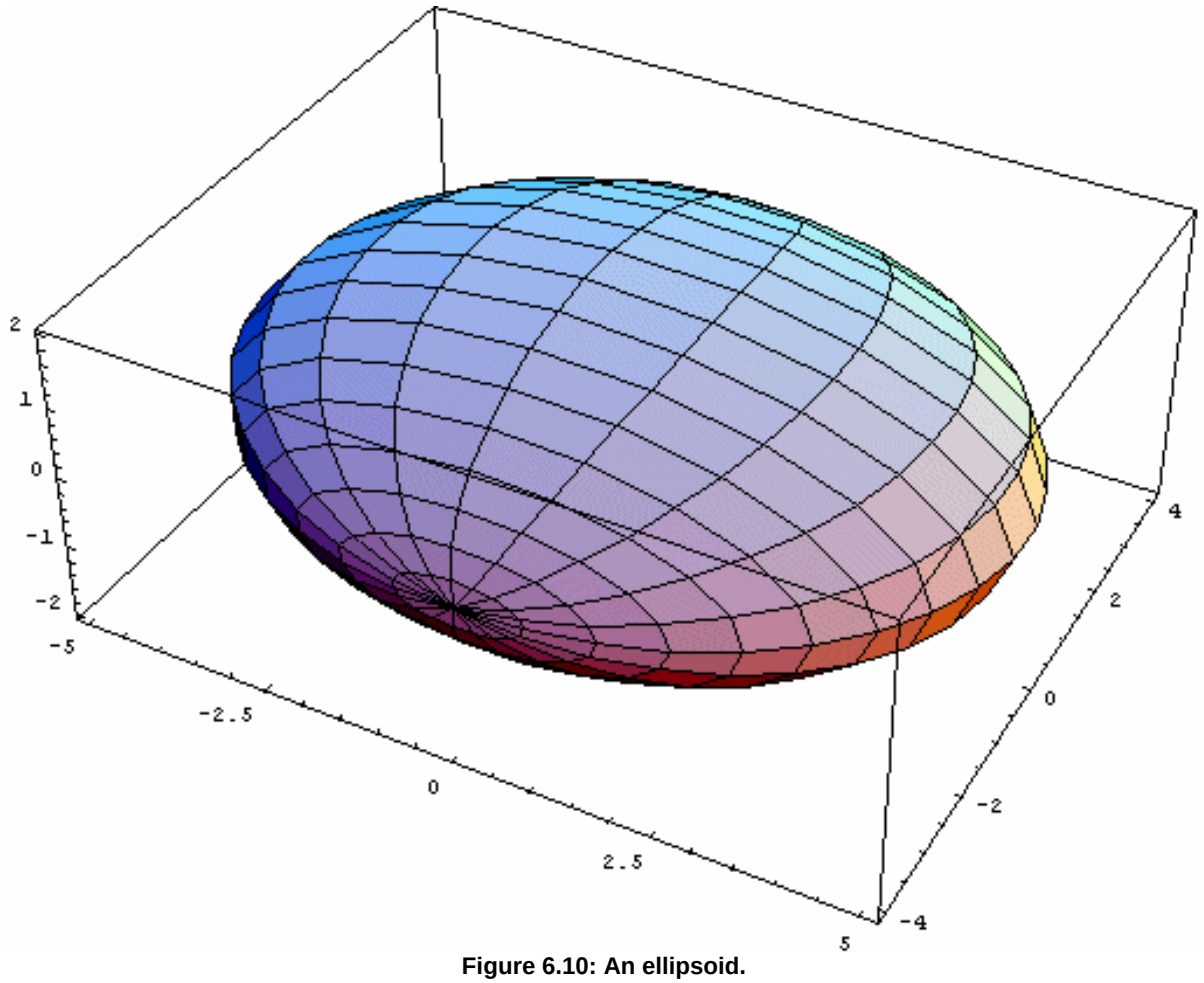


Figure 6.10: An ellipsoid.

The equations for ellipsoids are almost identical to those for ellipses. The standard form for an ellipse centered at (h,k,j) is shown below:

$$\frac{(x-h)^2}{r_x^{\,2}}+\frac{(y-k)^2}{r_y^{\,2}}+\frac{(z-j)^2}{r_z^{\,2}}=1$$

In this equation, we can solve for one variable -- *x*, *y*, or *z*. We will get two solutions, as before, one representing one half of the ellipse, and the other representing the other half.

The parametric form uses a form of spherical coordinates to describe a ray that emanates from the origin of the ellipsoid. The spherical coordinates are mapped to the point where the ray intersects the ellipsoid.

The parametric equations are shown below:

$$x = r_x \cos\phi \cos\theta + h$$
$$y = r_y \sin\phi + k$$
$$z = r_z \cos\phi \sin\theta + j$$

Here θ describes the rotation of the ray around the *y*-axis (0 radians points in the direction of the positive *x*-axis), and ϕ describes the tilt of the ray -- the angle it makes with the plane formed by the *x*- and *z*-axes.

For the special case of a sphere, the equations simplify as follows:

$$(x-h)^2 + (y-k)^2 + (z-j)^2 = r^2$$

$$x = r \cos\phi \cos\theta + h$$
$$y = r \sin\phi + k$$
$$z = r \cos\phi \sin\theta + j$$

6.4.1 Intersecting Lines with Spheres

It is possible to determine exactly where a three-dimensional line and an ellipsoid intersect, but the resulting equations are extremely complex, and generally too slow to compute to be of much use to game developers. However, for the special case of the sphere, the resulting equations are actually simple enough to be used in computer games.

Suppose we have a line described parametrically by the following equations:

$$x = t(x_1 - x_0) + x_0$$
$$y = t(y_1 - y_0) + y_0$$
$$z = t(z_1 - z_0) + z_0$$

Further, suppose we have a sphere described by the following equation:

$$(x_s - h)^2 + (y_s - k)^2 + (z_s - j)^2 = r^2$$

To find out where the line and sphere intersect, the first thing to notice is that the intersection points must (1) satisfy the line equations, and (2) satisfy the sphere equation. So one thing we can do is

substitute the expressions for *x*, *y*, and *z* into the sphere equation. (If it is not obvious why this works, go through the whole set theory procedure we discussed when intersecting two lines). Doing this and collecting like terms, we get the rather complicated expression shown below:

$$t^2\left\{(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2\right\}+$$
$$t\left\{\begin{array}{c}-2h(x_1-x_0)+2x_0(x_1-x_0)-\\ 2k(y_1-y_0)+2y_0(y_1-y_0)-\\ 2j(z_1-z_0)+2z_0(z_1-z_0)\end{array}\right\}+$$
$$(h^2+k^2+j^2-2hx_0+x_0^2-2ky_0+y_0^2-2jz_0+z_0^2)=r^2$$

To simplify, we should let:

$$a=(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2$$
$$y-y_0=m(x-x_0)$$
$$c=h^2+k^2+j^2-2hx_0+x_0^2-2ky_0+y_0^2-2jz_0+z_0^2-r^2.$$

Now we can plug these values into the quadratic formula to solve this equation for *t*. We get the following two solutions:

$$t_0=\frac{-b+\sqrt{b^2-4ac}}{2a}$$
$$t_1=\frac{b+\sqrt{b^2-4ac}}{2a}$$

To find the two points where the line intersects the sphere, simply plug these values of *t* into the parametric equation for the line. Note that if the discriminant of the above equations is negative, then the line does not intersect the sphere at all. This allows us to use the equations just to see if a line intersects a sphere (even if we are not interested in the location of the intersection).

6.5 Planes

Planes are two-dimensional surfaces that extend infinitely in both dimensions and reside in three-dimensional space. Sometimes, people talk about the *x*-*y* plane, for example. This simply refers to the plane that contains the *x*- and *y*-axes. Similarly, the *x*-*z* plane contains the *x* and *z* axes, and the *y*-*z* plane contains the *y* and *z* axes. Figure 6.11 illustrates this:

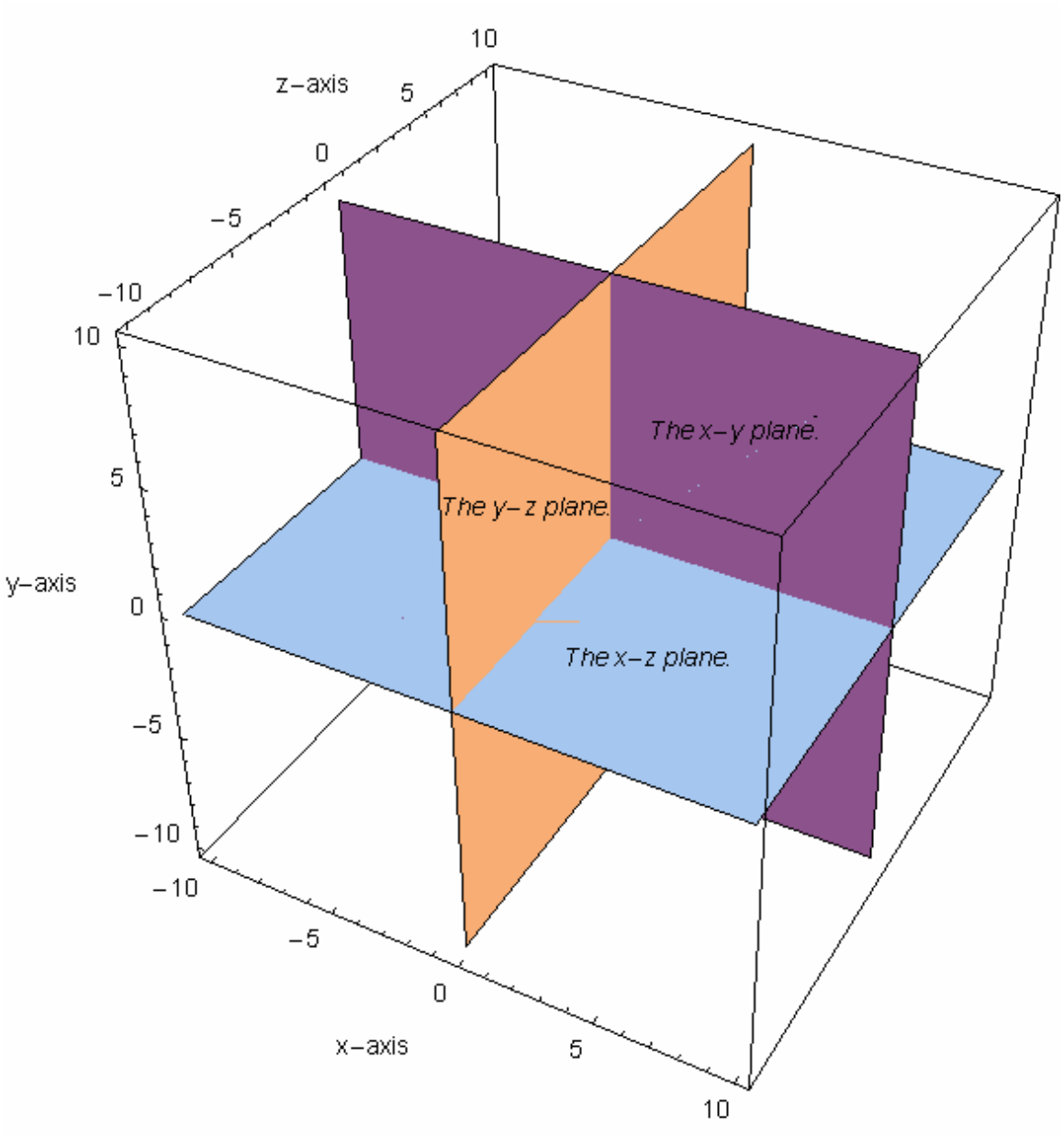


Figure 6.11: The names for planes containing two standard axes.

If a plane is parallel to the *y*-*z* plane, then the usual way to represent it mathematically is $x = a$. This is a shorthand way of saying that the plane contains all points in the set $\{(x, y, z) \mid x = a, y \in \Re, z \in \Re\}$. Similar notations are observed for planes parallel to the *x*-*y* and *x*-*z* planes.

We should have no problem believing that if we look at a vertical cross section of a non-vertical plane, we will see a line. (Looking at a cross section corresponds to intersecting the plane with another plane). So in particular, if we look at a cross section parallel to the *y*-*z* plane, we will see a line, and if we look at a cross section parallel to the *x*-*y* plane, we will see a line. Figure 6.12 shows a plane and the two lines that result from intersecting the plane with the *y*-*z* and *x*-*y* planes.

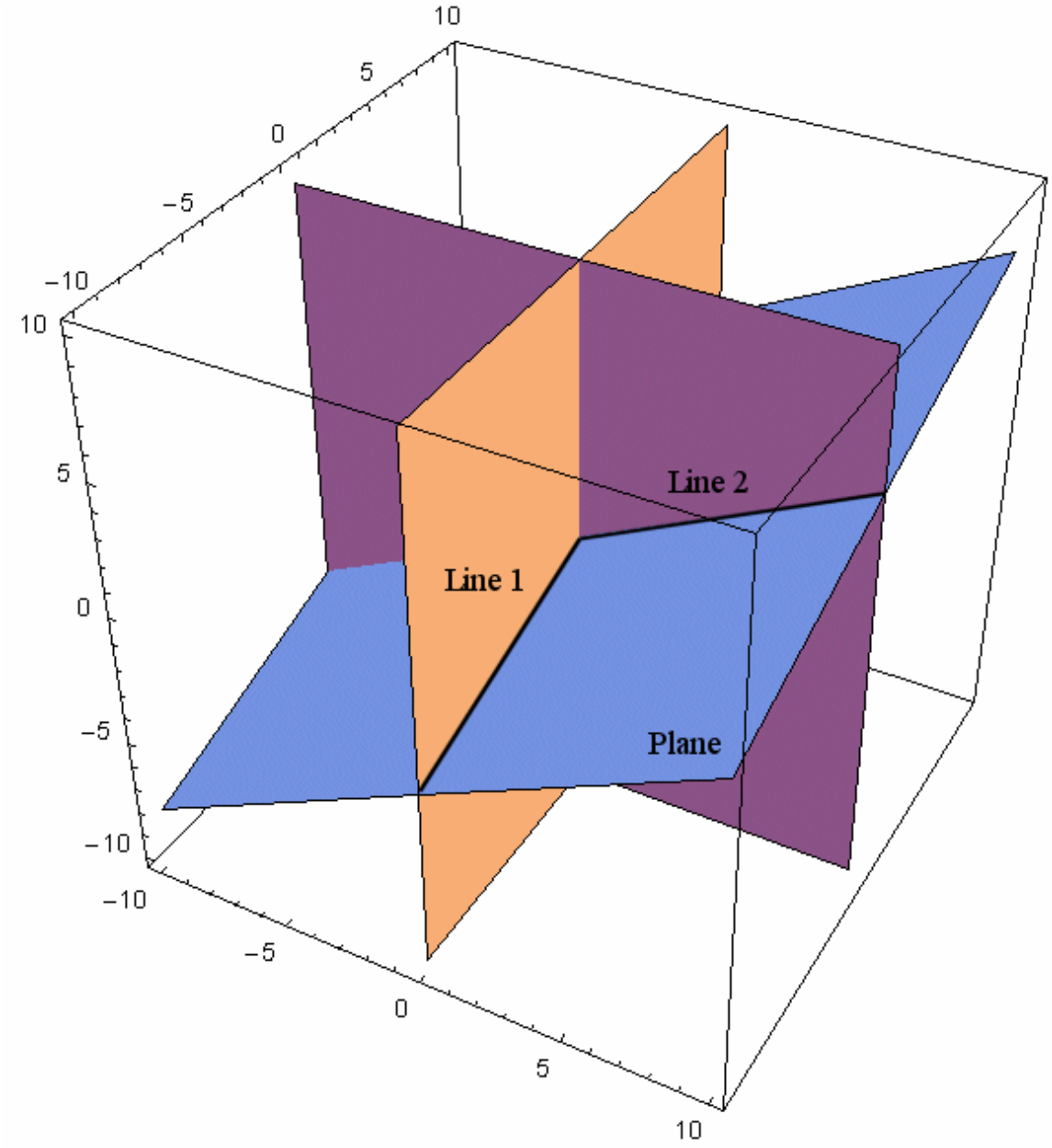


Figure 6.12: Cross sections of a plane.

This should lead us to conclude that perhaps we can represent a non-vertical plane with the sum of two line equations: one line equation that depends on *x*, and one line equation that depends on *y*. This turns out to be correct, and the resulting equation that describes the height of the plane for any given (x, z) is shown below:

$$y = mx + nz + c$$

Here, *m* is the slope of the line that runs along the *y*-*z* plane, *n* is the slope of the line that runs along the *x*-*y* plane, and *c* is the *y* value of the plane at the point (0, 0). For non-vertical planes, we can express *x* in terms of *y* and *z*, or *z* in terms of *x* and *y*, depending on the plane. In our next lesson, we will discuss a more elegant equation called the *general* (or *standard*) *plane equation*, that represents all planes, regardless of orientation. But for now, this one will suffice.

6.5.1 Intersecting Lines with Planes

It is quite trivial to calculate the intersection point between a line and a plane. Assuming the line is in standard parametric form, and the plane is determined by the equation $y = mx + nz + c$, then the value of *t* that produces the intersection point is given by the following equation (you will derive this equation in an exercise):

$$t = \frac{c + mx_0 + nz_0 - y_0}{mx_0 - mx_1 + nz_0 - nz_1 - y_0 + y_1}$$

To find the point, just plug this value of *t* into the parametric equation for the line, and we are done.

If the line is parallel to the plane, then the denominator of the above equation will evaluate to zero, indicating there is no solution.

Conclusion

In the next chapter, we bring out the heavy-duty math: vector algebra. Vectors allow us to do even more advanced analytic geometry than what we have looked at here in this lesson. Plus, we can use them for geometry transformation, animation, performance optimization, and much more. They also lead naturally to a discussion of matrices, the workhorse of every 3D program on the market.

Exercises

*1. Determine the intersection between a line and a quadratic.

*2. Determine the intersection between a line (in parametric form) and a plane.

3. A direct hit to the head with a gun is far more serious than a direct hit to the foot (in fact, the former is usually lethal!), and games like *Hitman*™ and *Soldier of Fortune*™ take this into consideration to increase realism. This approach requires you to know which body part is hit by the weapons. The exact math required to do this is covered in Chapter Eleven, but for now, describe how you could use spheres to achieve a rough solution to this problem.

4. Ray-tracing is a rendering technique whereby a two-dimensional view of a 3D world is produced by sending out imaginary rays from the viewer to each pixel on the screen, and then following the rays out into the 3D world to see which point they intersect first (see Figure E6.1). Suppose the origin of the coordinate system is placed at the viewer's location, which is assumed to be a distance *d* from the screen, centered vertically and horizontally. Derive an equation for the 3D line that passes through the pixel located at the 3D point (*x*, *y*, *d*).

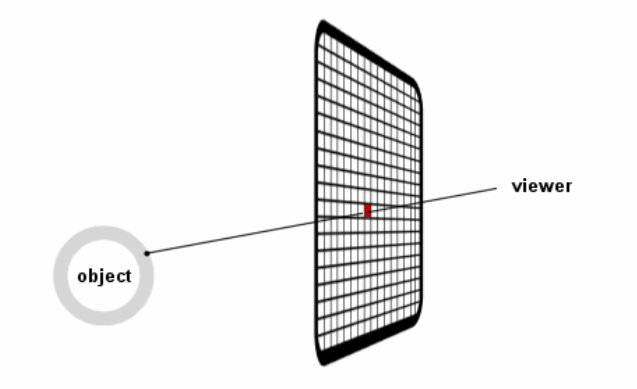


Figure E6.1: Ray-tracing.

! 5. Find an inequality (i.e. an equation stating that one thing is less than, greater than, less than or equal to, or greater than or equal to something else) that holds for all points above a line in slope-intercept form, and another inequality that holds for all points below the line.

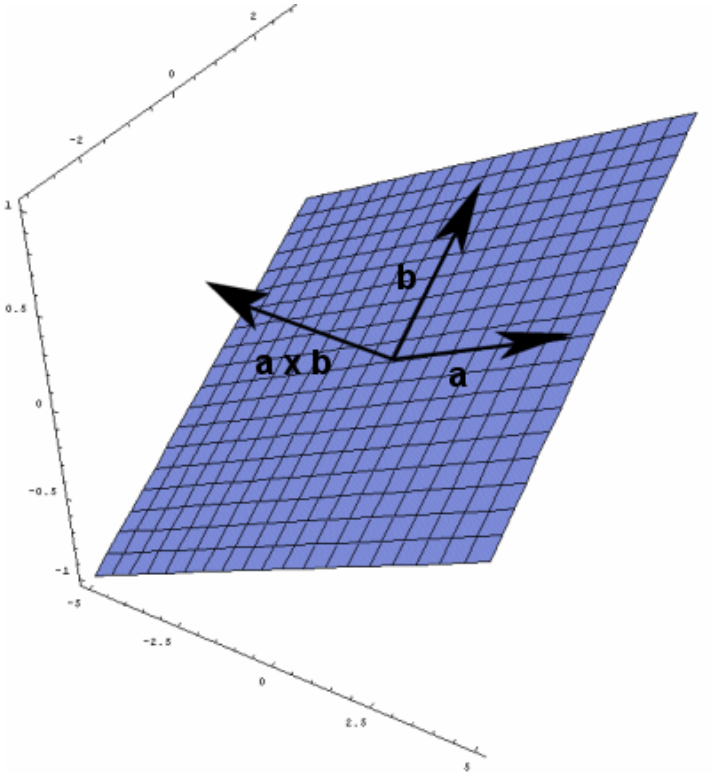
*6. Describe how you could use the inequalities of problem 5 to determine if a point falls within a two-dimensional polygon.

! 7. Suppose the viewer (situated as described in problem 4) has a horizontal field of view of *h*, and a vertical field of view of *v*, and is looking through a computer screen at a 3D world. Find equations that describe six planes that enclose the portion of the 3D world that the viewer sees.

*8. Describe how you can use the results of problems 6 and 7 to determine if the viewer can see a point.

Chapter Seven

Vector Mathematics



Introduction

If you asked me for my top 10 favorite math subjects, vectors would definitely be right up there. Ever since I was first introduced to them, vectors have captivated me with their elegance and wide applicability.

It is not often in mathematics that a clean, simple concept generates an enormous variety of applications, but with vectors, that is surely the case. From two-dimensional and three-dimensional graphics, to physics, to sophisticated bones-based animation -- anywhere you look in a modern game, you are bound to run across vectors and the mathematics that deals with them.

In this chapter, we are going to get a thorough introduction to the world of vector math. We will cover what vectors are, what operations are defined for them, and a few of the many different ways you can use them in game programming.

7.1 What Are Vectors?

A plain old number, such as 2 or 9102, is useful for describing the size or *magnitude* of something: like the temperature, for example, or the length of a piece of sheet metal, or the number of votes a candidate wins from a local precinct. Single numbers are not very good, however, for describing *directions*. If I asked you where the airport was, for example, you might tell me it was 27.1 miles away, but that single number -- though helpful -- is not nearly enough information for me to find the airport.

That is where vectors come in. A vector can be *very loosely* defined as *an entity that has both a direction and a magnitude.* That is not perhaps the best definition for vectors since there are entities that have both direction and magnitude and yet are not vectors -- in fact, entities must satisfy other properties as well before you can call them vectors -- but for now this definition will suffice.

Returning to the airport example, you could tell me the airport was 27.1 miles in the direction you were pointing. That direction, combined with the magnitude of 27.1 miles, together constitute a vector.

Similarly in physics, *velocities* are vectors. A velocity describes the direction of movement and how fast that movement is. For example, a train might be moving at 70 mph in the northeast direction.

Since directions can exist in any number of dimensions, so can vectors. You can have one-dimensional vectors, two-dimensional vectors, three-dimensional vectors, and so on. Although not used frequently in computer science, there are occasions when higher dimensional vectors are useful (especially in physics).

Vectors can be visually represented by a so-called *directed line segment*, which is nothing more than a portion of a straight line with an arrowhead stuck on the end. The direction of both the line segment and the arrowhead designate the direction of the vector, while the length of the vector represents its magnitude. Figure 7.1 shows you how this looks with a number of different vectors.

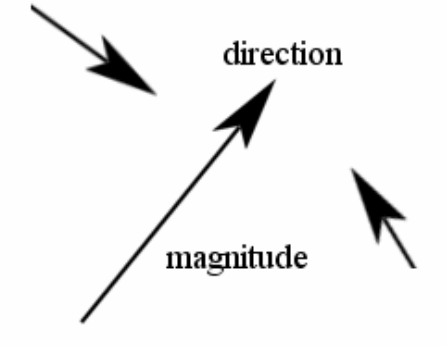


Figure 7.1: Examples of vectors.

Vectors are usually represented by lower case English letters, oftentimes in bold or italic face (such as **a** or *b*, or even ***c***), and occasionally (when the typesetter is feeling extraordinarily ambitious) by a letter with an arrow placed directly overhead (e.g. $\vec{v}$).

7.2 Elementary Vector Math

For a thing to be a vector, it must not only have a direction and magnitude, but it must also have certain properties and obey certain rules. These properties and operations are what make vectors so incredibly useful across a broad range of mathematical, physical, and computational disciplines.

Addition is probably the most basic of all vector operations. The operation of addition takes two vectors and returns another vector, called the *resultant*. Geometrically, you can think of addition as taking the base of the first vector and sticking it at the tip of the second; the resultant vector is the vector formed by the line segment that travels from the base of the second vector to the tip of the first. This is shown in Figure 7.2.

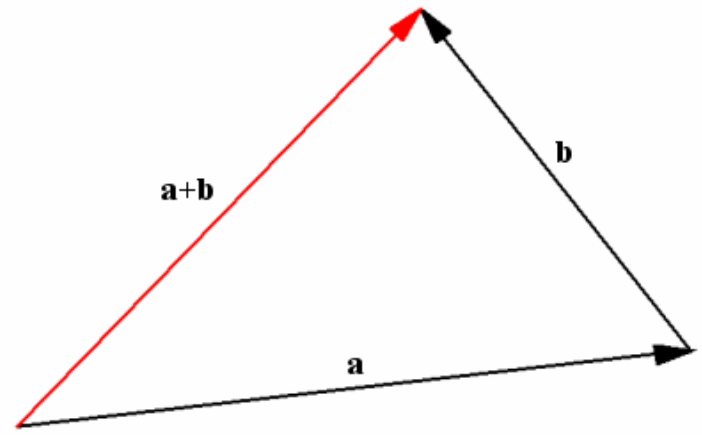


Figure 7.2: Adding two vectors.

As you can probably guess from the definition, vector addition is commutative (**a** + **b** = **b** + **a**), as well as associative ((**a** + **b**) + **c** = **a** + (**b** + **c**)).

The so-called *zero* or *null vector* is a vector with no direction or magnitude (weird, but true), designated by **0**. It exists so that we can write equations like **a** + **0** = **a**.

Subtraction is defined for vectors too. If you have two vectors **a** and **b**, then to subtract **b** from **a**, place the bases of the vectors at the same point, then draw a line segment from the tip of **b** to the tip of **a**. The vector formed by this line segment defines the result of the subtraction. Figure 7.3 shows how this looks.

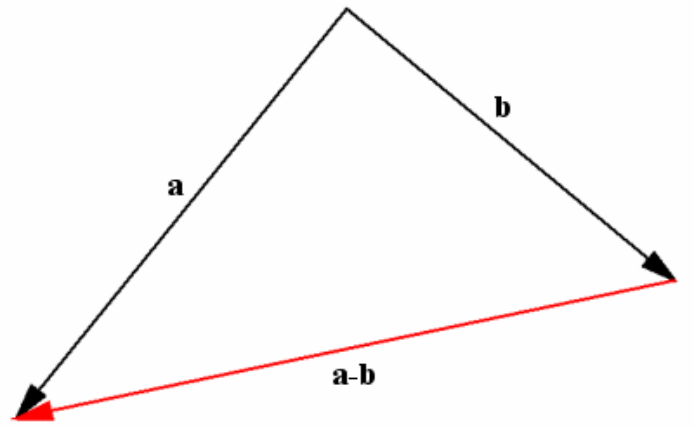


Figure 7.3: Vector subtraction.

Like vector addition, subtraction is both commutative and associative. Furthermore (and this is not an accident) any vector minus itself (**a - a**) is the zero vector, a consequence of the way subtraction is defined.

You can multiply a vector by any real number, called a *scalar* because it *scales* the vector. The new vector has the same direction as the old one if the scalar is positive, but the opposite direction if the scalar is negative. The magnitude of the new vector is equal to its old magnitude times the absolute value of the scalar.

For one thing, this definition of scalar multiplication allows you to write **a** + (-1 **a**) = **0**, since -1 **a** points in the opposite direction as **a** but has the same magnitude, so adding the two vectors will result in the null vector (see Figure 7.4).

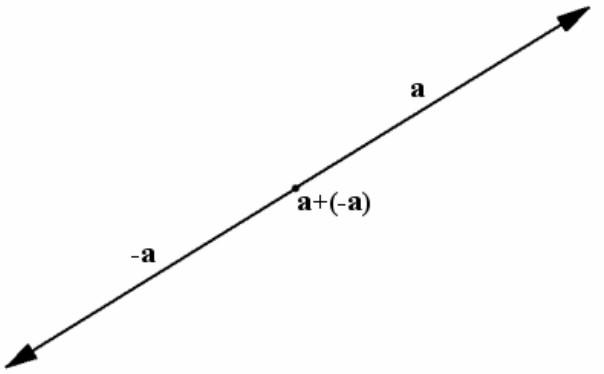


Figure 7.4: Adding the vectors a and -a.

Scalar multiplication is distributive with respect to vector addition, so $a(\mathbf{u}+\mathbf{v}) = a\mathbf{u}+a\mathbf{v}$ (where a is a scalar and **u** and **v** are vectors). It is also associative, since $a\mathbf{u} = \mathbf{u}a$, and commutative as well.

One of the things we take for granted in the arithmetic of ordinary numbers is that we can easily write $x + (-x)$ as $x - x$. For vectors, however, the first would be a vector plus a scalar times a vector, and the second would be a vector minus another vector. Fortunately, by the way scalar multiplication is defined, it turns out this property holds true for vector arithmetic as well, but if we had defined scalar multiplication in any other way (e.g. the new vector has a length equal to its old length *plus* the scalar) the property would not hold. The moral of this story is that you need to be careful whenever you create a new algebra not to ascribe properties of real number arithmetic to the new algebra.

You can divide a vector by a scalar by multiplying it by the reciprocal of the real number. This intuitive way of defining scalar division allows you to write equations like $5(\mathbf{a}/5) = \mathbf{a}$.

The magnitude (i.e. length) of a vector comes up so frequently that there is a special designation for it: two vertical bars ('|') that enclose the vector. Thus the magnitude of vector **a** can be designated by $|\mathbf{a}|$. In many math texts, You will also see the magnitude of a vector represented by the letter standing in for the vector, without boldface, and possibly italicized (in a text where a vector is written **v**, for example, the magnitude of the vector might be written *v*, or maybe just v).

To *normalize* a vector is to make its length 1. You can do this by taking the vector and dividing it by its own length. The resulting vector is called a *normalized* vector.

That about does it for the elementary vector operators. You may be wondering if you can multiply a vector by another vector, or divide two vectors. As we will see in the next section, the answer to the first question is "sort of". Division of one vector by another, however, cannot be defined in any meaningful way. We will briefly see why this is true after defining the two operations of vector multiplication.

7.2.1 Vector Multiplication

It is not immediately obvious how to define multiplication for vectors. In the case of real numbers, *ab* is defined as the sum of a certain number of *a*'s -- namely, *b* of them (or conversely, as a sum of *a b*'s). So the first thing you might think of is to define the vector product **ab** as the sum of a certain number of **a**'s -- namely, **b** of them -- but what does it mean to say **b** of them, since **b** is a vector and not a number?

The answer is that it does not mean anything, so it is not possible to define vector multiplication in a way that parallels real number multiplication. So instead, what mathematicians have done is define two operations, called the *dot product* and the *cross product*, that act like multiplication in some respects (they are distributive with respect to vector addition, for example, just like real number multiplication), but do not have an exact parallel in the algebra of real numbers.

In the next two sections, we will look at both of these operations in some depth.

The Dot Product / Vector Projection

The dot product operator, designated by the symbol '•', tells you something about the angle between two vectors. Here you can understand "angle between two vectors" as the angle formed by geometrically dragging the base of one vector to the base of the other (see Figure 7.5).

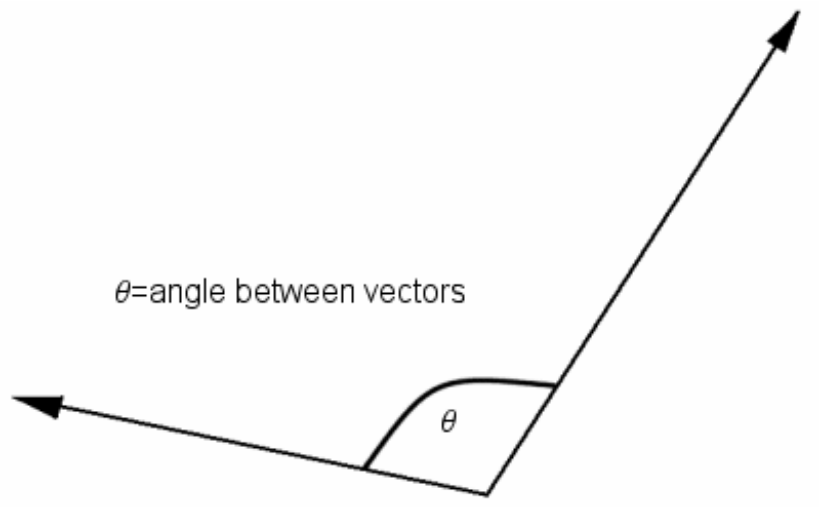


Figure 7.5: The angle between two vectors.

More precisely, **a•b** (read "a dot b") is defined as $|\mathbf{a}||\mathbf{b}|\cos(\theta)$; that is, the magnitude of **a** times the magnitude of **b** times the cosine of the angle between the vectors.

This may seem an odd way to define the dot product. After all, if we are interested in the angle between two vectors, why not simply define the dot product as just that? Actually there is a reason for the strange definition: it is very easy to derive a formula for the dot product of two vectors given their Cartesian representations (more on this later).

In any case, if you want to find the angle between two vectors, all you have to do is solve the equation $\mathbf{a}\cdot\mathbf{b}=|\mathbf{a}||\mathbf{b}|\cos(\theta)$ for θ, which gives you $\theta=\cos^{-1}(\frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|})$ (which is true only assuming you choose θ so that it falls in the interval $[0,\pi]$, the range of the inverse cosine function).

The dot product is commutative and associative with respect to vector addition -- the latter property giving it some similarity to multiplication. Of course, the result of the dot product operation is not a vector, but a scalar, whereas the product of two real numbers is again a real number, so the parallel between the two breaks down.

The definition of the dot product leads to an interesting (and sometimes useful) way to calculate the magnitude of a vector: $\sqrt{a\cdot a}=\sqrt{|\mathbf{a}||\mathbf{a}|\cos(0)}=\sqrt{|\mathbf{a}||\mathbf{a}|(1)}=\sqrt{|\mathbf{a}|^2}=|\mathbf{a}|$.

The dot product leads naturally to an operation known as *vector projection*. The projection of some vector **a** onto another vector **b**, sometimes denoted $proj_{\mathbf{b}}(\mathbf{a})$, is a vector pointing in the direction of **b**, whose magnitude is equal to the part of **a** that points in the direction of **b**.

Figure 7.6 graphically depicts what vector projection is all about. The figure also suggests a way to determine what the projected vector is.

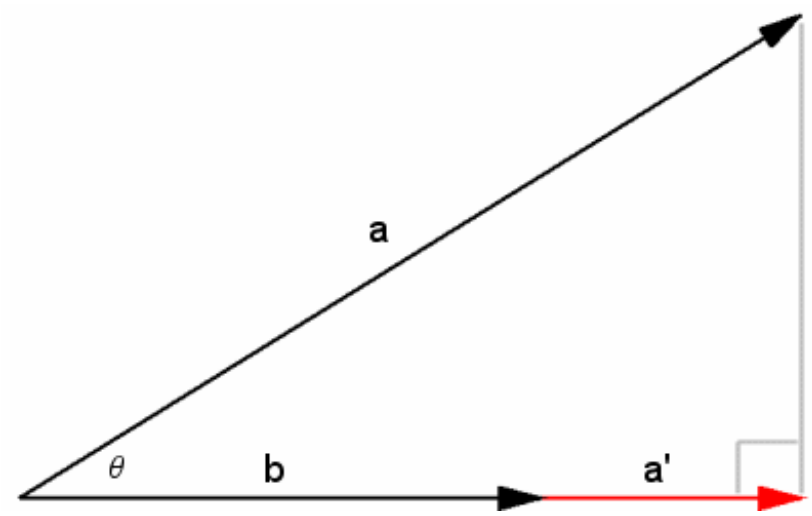


Figure 7.6: Vector projection.

Suppose we call the projected vector **a'**. We know **a'** will have the same direction as **b**, but what about its magnitude? From the right triangle drawn in Figure 7.6, we know that $\cos(\theta) = \frac{|\mathbf{a}'|}{|\mathbf{a}|}$, so solving for |**a'**|, we find $|\mathbf{a}'| = |\mathbf{a}|\cos(\theta)$. From the definition of the dot product, we also know that $\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta)$, so dividing both sides of the equation by |**b**|, we see that $|\mathbf{a}|\cos(\theta) = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{b}|}$. Together, these two equations imply that $|\mathbf{a}'| = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{b}|}$. If we multiply this number (which is the magnitude of **a'**) by the normalization of **b** (which is just $\frac{\mathbf{b}}{|\mathbf{b}|}$), then we will have the vector **a'**. Thus our formula for vector projection becomes,

$$proj_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{b}|^2}\mathbf{b}$$

However, note that $|\mathbf{b}|^2 = \mathbf{b}\cdot\mathbf{b}$, so we can also write the formula as follows:

$$proj_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\mathbf{b}$$

Vector projection is often used in physics simulations and collision detection. In physics, for example, you use vectors to represent forces, and sometimes you need to figure out what part of a force points in a given direction. Vector projection allows you to easily calculate this result.

The Cross Product

The cross product operation is a bit strange in that it is only defined for three-dimensional vectors. That means that within the algebra of two-dimensional vectors or four-dimensional vectors, for example, the cross product operation does not exist.

The cross product of two vectors **a** and **b**, written $\mathbf{a} \times \mathbf{b}$, is a vector, not a scalar (in contrast to the dot product of two vectors). What sort of vector? Consider that if you move the base of one of these vectors to the base of the other vector, as done in Figure 7.7, then the two vectors determine a three-dimensional plane (that is, there is one and only one plane that passes through both the common base and the tips of the two vectors). The cross product is a vector that is perpendicular to the plane (and hence, to the two vectors).

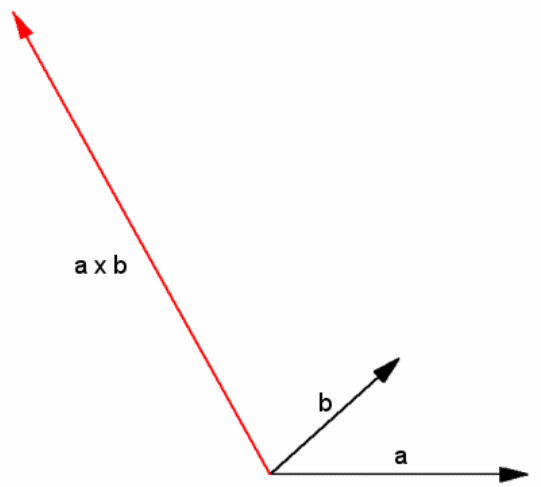


Figure 7.7: The cross product operation.

We have left two things out in this description of the cross product: (1) what side of the plane the cross product vector points away from (since it could point away from either side of the plane and still be perpendicular to the plane) and (2) what the magnitude of the cross product vector is.

The cross product vector points in the direction determined by the *right-hand rule*. To use the right-hand rule, make a flat hand, then stick your thumb straight out at a 90-degree angle from the direction of your fingers. Then angle your hand so that you can rotate your fingers from the direction of the first vector to the direction of the second (from **a** to **b** in the above example, although if we were computing $\mathbf{b} \times \mathbf{a}$, it would be from **b** to **a**) without rotating more than 180 degrees (see Figure 7.8). The cross product vector points in the same general direction as your thumb. (A variant of this rule involves turning a screw from the first vector to the second in such a way that you do not turn more than 180 degrees; the screw will go in the same direction the cross product vector points.)

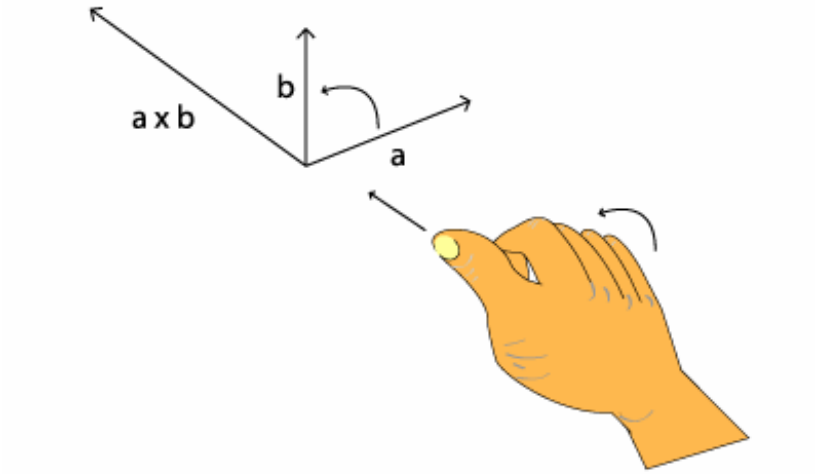


Figure 7.8: The right-hand rule for cross products.

The magnitude of the cross product vector is defined to be $|\mathbf{a}||\mathbf{b}|\sin(\theta)$, where **a** and **b** are the vectors being crossed (note the similarity to the dot product) and θ is the smallest angle between the two vectors, as illustrated back in Figure 7.7. You can interpret the magnitude of the cross product vector as the area of the parallelogram formed by the two vectors. Figure 7.9 shows you why this is so.

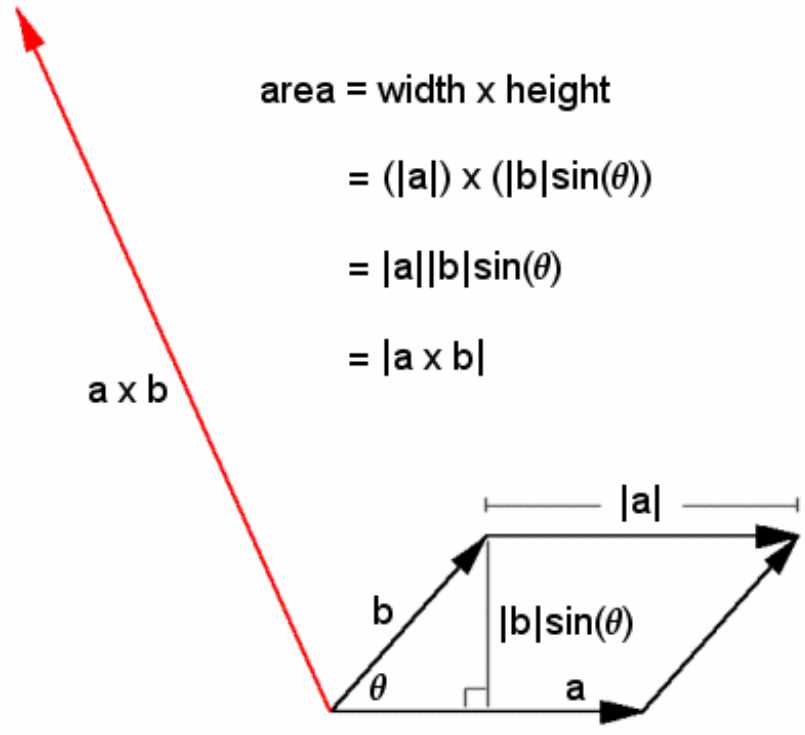


Figure 7.9: The parallelogram interpretation of the magnitude of the cross product vector.

The cross product is associative with respect to addition, so $\mathbf{a}\times(\mathbf{b}+\mathbf{c})=\mathbf{a}\times\mathbf{b}+\mathbf{a}\times\mathbf{c}$. However, it is *not* commutative: the vector $\mathbf{a}\times\mathbf{b}$ does have the same magnitude as the vector $\mathbf{b}\times\mathbf{a}$, but as the right-hand rule should convince you, it points in the *opposite direction*. This means that $\mathbf{a}\times\mathbf{b}=-(\mathbf{b}\times\mathbf{a})$. As we will

see later in this course, the cross product is but one of many mathematical operations that is not commutative (the idea of a non-commutative operation is strange at first, but you get used to it after a while and stop taking commutativity for granted).

It should be clear by now that neither the dot product nor the cross product is really a substitute for a "multiplication" of vectors. The operation that is associative and commutative results in a number, not a vector, and the operation that results in a vector is associative but not commutative. As we will see in the next section, we cannot have anything even remotely approaching division.

7.2.2 Vector Division

One of the most fundamental uses for division is solving an equation for an unknown. In the equation $5x = 10$, for example, all we need to do is divide both sides of the equation by 5 in order to determine the value of *x*. So it makes sense to define division in vector math in a way that lets us solve vector equations.

Consider the equation $\mathbf{a} \cdot \mathbf{x} = b$. Ideally, we would divide both sides of the equation by the vector **a**, giving us, $\mathbf{x} = \left(\frac{1}{\mathbf{a}}\right) b$ for some definition of 1/**a** (which must be a vector, since *b* is a scalar). The problem with this approach is that there are *infinitely many* solutions to the equation $\mathbf{a} \cdot \mathbf{x} = b$ for any given **a** and *b*. Remember that $\mathbf{a} \cdot \mathbf{x} = b$ simply implies that $|\mathbf{a}||\mathbf{x}|\cos(\theta) = b$. This equation is true for any choice of **x** and θ such that $|\mathbf{x}|\cos(\theta) = \frac{b}{|\mathbf{a}|}$, and there are infinitely many such choices, so there is no single vector we can assign to 1/**a**. This means that division in the case of the dot product is not possible -- at least, no division is possible that would allow us to solve vector equations.

You run into the same problem with cross products. Consequently, division in any form is not defined for vectors.

7.3 Linear Combinations

If you add up a bunch of vectors, each multiplied by some arbitrary scalar, then you will get back a vector. Such a vector is called a *linear combination* of the others. For example, the following vector:

$$\mathbf{v} = \alpha_1 \mathbf{a}_1 + \alpha_2 \mathbf{a}_2 + \alpha_3 \mathbf{a}_3 + \cdots + \alpha_n \mathbf{a}_n$$

is a linear combination of the vectors $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ (all the α's are scalars).

Of special note is the following linear combination:

$$\mathbf{0} = \alpha_1 \mathbf{a}_1 + \alpha_2 \mathbf{a}_2 + \alpha_3 \mathbf{a}_3 + \cdots + \alpha_n \mathbf{a}_n$$

Using the geometric definition of vector addition, this case looks something like that shown in Figure 7.10. As you can see, all the vectors form a closed loop. If you add $a_2\mathbf{a}_2$ to $a_1\mathbf{a}_1$ and then add $a_3\mathbf{a}_3$ to that, and continue doing the process, you will eventually wind up exactly where you started: the base of the vector $a_1\mathbf{a}_1$. The sum of all those vectors equals the null vector.

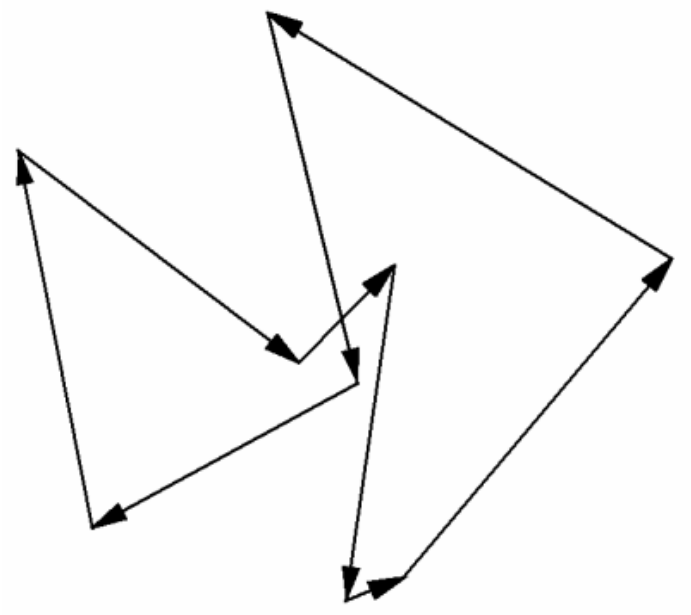


Figure 7.10: The visual depiction of $\mathbf{0} = \alpha_1\mathbf{a}_1 + \alpha_2\mathbf{a}_2 + \alpha_3\mathbf{a}_3 + \cdots + \alpha_n\mathbf{a}_n$.

If the only way the equation:

$$\mathbf{0} = \alpha_1\mathbf{a}_1 + \alpha_2\mathbf{a}_2 + \alpha_3\mathbf{a}_3 + \cdots + \alpha_n\mathbf{a}_n$$

can be satisfied for a given list of vectors $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ is if all the α's are zero, then the vectors are called *linearly independent*. The term "linearly independent" might conjure up images of vectors yelling, "None of you other vectors can represent me! I'm an independent!" Surprisingly, that is exactly what it means: in a linearly independent set of vectors, *no one vector can be represented as a linear combination of the others*.

To see why this is so, suppose the vectors $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ are linearly independent, and that we can represent $\mathbf{a_1}$ as a linear combination of the others (if it cannot be done, then we will get a contradiction). Then we can write $\mathbf{a_1}$ as:

$$\mathbf{a}_1 = \alpha_2\mathbf{a}_2 + \alpha_3\mathbf{a}_3 + \cdots + \alpha_n\mathbf{a}_n$$

for some scalars α_2, α_3, up to α_n. Now subtract $\mathbf{a_1}$ from both sides of the equation, giving us,

$$0 = -\mathbf{a}_1 + \alpha_2\mathbf{a}_2 + \alpha_3\mathbf{a}_3 + \cdots + \alpha_n\mathbf{a}_n$$

Here we have a linear combination of the vectors adding to zero, with $\mathbf{a_1}$ multiplied by the scalar -1. But that is a contradiction, since the vectors are linearly independent, and therefore, the only way linear combinations of the vectors can add to zero is if *all* the scalars are zero. Hence, in a linearly independent set of vectors, no one vector can be represented as a linear combination of the others.

We are teetering on the edge of vector analysis here, so we will steer the concept of linear independence toward a definite application: *n* linearly independent vectors, each of dimension *n*, form a so-called *basis* for *n*-dimensional space (almost sounds like it comes from an episode of *Star Trek*, does it not?). By a basis, we mean that *any* vector of dimension *n* can be *uniquely* represented as a linear combination of these basis vectors. These basis vectors, which are themselves *not unique*, are said to *span* the *n* dimensions.

Take, for example, the case of two dimensions: define a vector **u** as pointing along the positive *x*-axis, having length 1, and a vector **v** as pointing along the positive *y*-axis, also having length 1. Then any two-dimensional vector can be represented as a linear combination of the vectors **u** and **v** (remember, these vectors are not unique -- any two linearly independent vectors would work as a basis).

The *standard basis* for *n* dimensional space consists of a list of *n* vectors: the first one points along the positive direction of the first axis, the second, along the positive direction of the second axis, and so on. Each of the vectors has length 1. These vectors are used so often they have standard notations: typically they're called $\mathbf{i_1}$, $\mathbf{i_2}$, $\mathbf{i_3}$, ..., $\mathbf{i_n}$, or, for the 2D and 3D cases, **i** and **j**, and **i**, **j**, and **k**, respectively. These basis vectors are called *orthonormal* because they are all at right angles to each other and each has length 1.

7.4 Vector Representations

So far, all of the discussion about vectors has been pretty abstract: we have talked about directed line segments, but directed line segments are, after all, *geometric objects*, and consequently of little use to computers. In Chapter Six we learned that many geometric shapes have mathematical representations, and that expressing them in these mathematical forms is the basis of analytic geometry -- an immensely useful tool in math, physics, computer science, and computer game development. Vectors are no exception. If we want to wield their power, we have to find some way to represent them analytically, with numbers instead of shapes, so we can manipulate those numbers according to the rules of math.

There is more than one way to do this. One way we could represent vectors is to denote their orientation with a number of angles (one angle for two-dimensions, two angles for three-dimensions, and so on), and then represent their magnitude with a real number. This is done often in math and physics but is not the preferred representation in computer science because it is so difficult to work with (try to find a way to add two vectors in such a form, for example -- it is not pretty).

A much better representation is based on the Cartesian coordinate system, and is therefore referred to as the *Cartesian representation of vectors*. In this scheme, each point in the Cartesian coordinate system represents a vector: where the vector is the directed line segment drawn from the origin to the point. The magnitude of a vector is simply the distance from the origin to the point representing the vector. Figure 7.11 shows you the Cartesian representations of a vector.

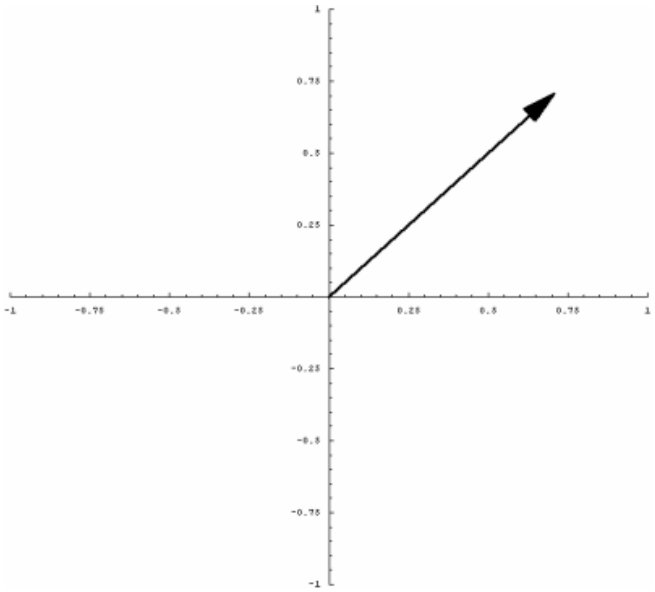


Figure 7.11: The Cartesian representation of vectors.

Points, as you have seen in past lessons, are represented by ordered *n*-tuples: lists of numbers enclosed in parentheses (such as the 3D point (*x*,*y*,*z*)). To differentiate points from vectors, vectors have their own notation: they are represented by lists of numbers enclosed in angular brackets ('<' and '>'). Hence (1, 3, -5) is a point, while <2, -1> is a vector (specifically, it is the vector represented by a directed line segment drawn from the origin to the point (2,-1)).

The standard basis vectors **i**, **j**, **k** for 3D space are represented as <1,0,0>, <0,1,0>, and <0,0,1>, respectively. Any vector written in the form <*x*,*y*,*z*> can also be written as $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$.

The reason why the Cartesian representation is better is because all the operations we have discussed are easy to implement for Cartesian vectors. In the next sections, we will revisit the operations of vector math, this time with attention to how you actually perform the operations with three-dimensional Cartesian vectors (you can easily derive the results for *n* dimensional vectors yourself).

7.4.1 Addition/Subtraction

The best way to determine how to add two vectors **u** and **v** is to add their standard basis representations. If $\mathbf{u} = \langle u_1, u_2, u_3 \rangle$, and $\mathbf{v} = \langle v_1, v_2, v_3 \rangle$, then we can write $\mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$, and $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. Thus $\mathbf{u} + \mathbf{v} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k} + v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k} = u_1\mathbf{i} + v_1\mathbf{i} + u_2\mathbf{j} + v_2\mathbf{j} + u_3\mathbf{k} + v_3\mathbf{k}$ (by the commutative property of vector addition) $= (u_1+v_1)\mathbf{i} + (u_2+v_2)\mathbf{j} + (u_3+v_3)\mathbf{k}$ (since scalar multiplication is distributive with respect to vector addition) $= \langle u_1+v_1, u_2+v_2, u_3+v_3 \rangle$.

Cleaning the result up a bit, we have found the following:

$\mathbf{u} + \mathbf{v} = \langle u_1+v_1, u_1+v_2, u_3+v_3 \rangle$

By the same process, we could also determine,

$\mathbf{u} - \mathbf{v} = \langle u_1 - v_1, u_1 - v_2, u_3 - v_3 \rangle$

This is the kind of clean, elegant result that gives the Cartesian representation its good name. To add two vectors, as we have now discovered, merely requires adding their Cartesian components.

7.4.2 Scalar Multiplication/Division

We will figure out how to multiply a vector by a scalar in the same way we determined vector addition: by giving the vector a standard basis representation.

Suppose we want to compute $a\mathbf{v}$, where a is the scalar and $\mathbf{v}$, the vector. If $\mathbf{v}$ has a Cartesian representation $\langle v_1, v_2, v_3 \rangle$, then we can write $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. Then $a\mathbf{v} = a(v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}) = av_1\mathbf{i} + av_2\mathbf{j} + av_3\mathbf{k} = \langle av_1, av_2, av_3 \rangle$. Thus we have,

$a\mathbf{v} = \langle av_1, av_2, av_3 \rangle$

Similarly,

$\mathbf{v}/a = \langle v_1/a, v_2/a, v_3/a \rangle$

7.4.3 Vector Magnitude

Recall that the magnitude of a vector is its length, which, in the Cartesian representation of vectors, corresponds to the distance from the origin to the tip of the vector. So if $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$, then the magnitude of the vector is given by,

$$|\mathbf{v}| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

7.4.4 The Dot Product

The dot product is trickier to compute than the other operations we have covered so far. To derive a formula, we will have to refer back to Chapter Five and use the very handy *Law of Cosines*. To refresh your memory, that law describes a relationship between the lengths of the sides of some arbitrary triangle and one of the angles of that triangle. Specifically, if a, b, and c are the lengths of the sides of the triangle, and θ is the angle between the sides a and b, then the law states the following (graphically depicted in Figure 7.12):

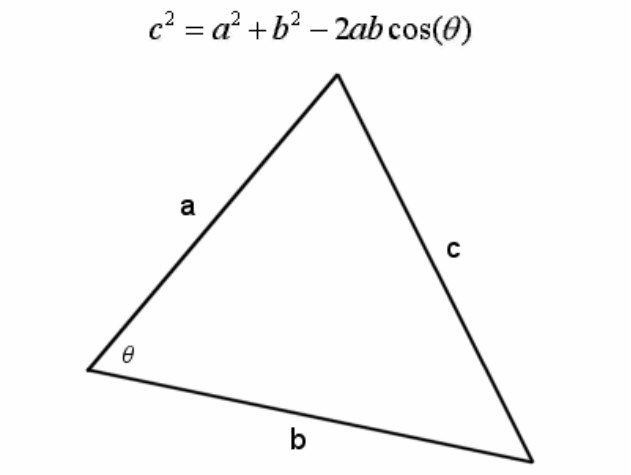


Figure 7.12: The Law of Cosines.

$$c^2 = a^2 + b^2 - 2ab\cos(\theta)$$

The right hand side of this equation looks *suspiciously* like the right hand side of the dot product equation, which states that $\mathbf{a}\cdot\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta)$, which should give you a clue as to how to solve the problem.

If we represent side *a* of the triangle with *vector* **a**, and side *b* with *vector* **b**, then we can represent side *c* with the vector **b**-**a** (which is, graphically, the vector that connects the tip of **a** to the tip of **b**). This state of affairs is shown in Figure 7.13 (it looks exactly like Figure 7.12 except vectors form the edges of the triangle).

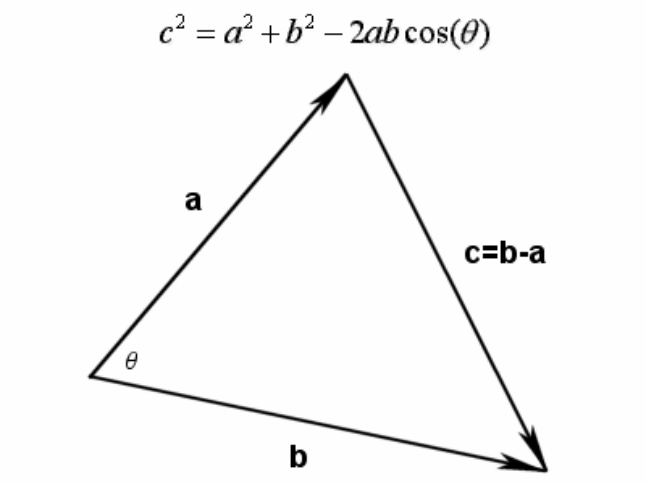


Figure 7.13: The Law of Cosines -- with vectors

Now we are in business, because we can apply the Law of Cosines to the magnitudes of these vectors, giving us the following relation:

$$|\mathbf{b}-\mathbf{a}|^2 = |\mathbf{a}|^2 + |\mathbf{b}|^2 - 2|\mathbf{a}||\mathbf{b}|\cos(\theta)$$

Solving for the quantity $|\mathbf{a}||\mathbf{b}|\cos(\theta)$ (which is what we are interested in), we find the following:

$$|\mathbf{a}||\mathbf{b}|\cos(\theta) = \frac{|\mathbf{a}|^2 + |\mathbf{b}|^2 - |\mathbf{b}-\mathbf{a}|^2}{2}$$

The only thing left to do now is to give each vector a Cartesian representation. If **a** = $\langle a_1, a_2, a_3 \rangle$ and **b** = $\langle b_1, b_2, b_3 \rangle$, then we can write the above equation like so:

$$|\mathbf{a}||\mathbf{b}|\cos(\theta) = \frac{\left(\sqrt{a_1^2 + a_2^2 + a_3^2}\right)^2 + \left(\sqrt{b_1^2 + b_2^2 + b_3^2}\right)^2 - \left(\sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + (b_3 - a_3)^2}\right)^2}{2} =$$

$$\frac{\left(a_1^2 + a_2^2 + a_3^2\right) + \left(b_1^2 + b_2^2 + b_3^2\right) - \left((b_1 - a_1)^2 + (b_2 - a_2)^2 + (b_3 - a_3)^2\right)}{2} =$$

$$\frac{\left(a_1^2 + a_2^2 + a_3^2\right) + \left(b_1^2 + b_2^2 + b_3^2\right) - \left((b_1^2 - 2b_1a_1 + a_1^2) + (b_2^2 - 2b_2a_2 + a_2^2) + (b_3^2 - 2b_3a_3 + a_3^2)\right)}{2} =$$

$$\frac{a_1^2 + a_2^2 + a_3^2 + b_1^2 + b_2^2 + b_3^2 - b_1^2 + 2b_1a_1 - a_1^2 - b_2^2 + 2b_2a_2 - a_2^2 - b_3^2 + 2b_3a_3 - a_3^2}{2} =$$

$$\frac{(a_1^2 - a_1^2) + (a_2^2 - a_2^2) + (a_3^2 - a_3^2) + (b_1^2 - b_1^2) + (b_2^2 - b_2^2) + (b_3^2 - b_3^2) + 2b_1a_1 + 2b_2a_2 + 2b_3a_3}{2} =$$

$$\frac{2b_1a_1 + 2b_2a_2 + 2b_3a_3}{2} =$$

$$b_1a_1 + b_2a_2 + b_3a_3$$

Whew! This should teach you that no matter how ugly an equation gets, you should persevere -- if there is any justice in life, it will simplify greatly at the end.

All of these calculations establish the following important formula for calculating the dot product:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta) = a_1b_1 + a_2b_2 + a_3b_3$$

This is a nice result because it allows you to find out something about the angle between two vectors just by multiplying their Cartesian components together.

7.4.5 The Cross Product

Compared to the dot product operation, computing the cross product is a breeze thanks in large part to the distributive property of the cross product operation. Given two vectors **u** and **v** with standard representations, their cross product can be computed as follows:

$$
\begin{aligned}
\mathbf{u}\times\mathbf{v} &= (u_1\mathbf{i}+u_2\mathbf{j}+u_3\mathbf{k})\times(v_1\mathbf{i}+v_2\mathbf{j}+v_3\mathbf{k}) \\
&= (u_1\mathbf{i}+u_2\mathbf{j}+u_3\mathbf{k})\times v_1\mathbf{i} + \\
&\quad (u_1\mathbf{i}+u_2\mathbf{j}+u_3\mathbf{k})\times v_2\mathbf{j} + \\
&\quad (u_1\mathbf{i}+u_2\mathbf{j}+u_3\mathbf{k})\times v_3\mathbf{k} \\
&= u_1\mathbf{i}\times v_1\mathbf{i} + u_2\mathbf{j}\times v_1\mathbf{i} + u_3\mathbf{k}\times v_1\mathbf{i} + \\
&\quad u_1\mathbf{i}\times v_2\mathbf{j} + u_2\mathbf{j}\times v_2\mathbf{j} + u_3\mathbf{k}\times v_2\mathbf{j} + \\
&\quad u_1\mathbf{i}\times v_3\mathbf{k} + u_2\mathbf{j}\times v_3\mathbf{k} + u_3\mathbf{k}\times v_3\mathbf{k} \\
&= u_1v_1(\mathbf{i}\times\mathbf{i}) + u_2v_1(\mathbf{j}\times\mathbf{i}) + u_3v_1(\mathbf{k}\times\mathbf{i}) + \\
&\quad u_1v_2(\mathbf{i}\times\mathbf{j}) + u_2v_2(\mathbf{j}\times\mathbf{j}) + u_3v_2(\mathbf{k}\times\mathbf{j}) + \\
&\quad u_1v_3(\mathbf{i}\times\mathbf{k}) + u_2v_3(\mathbf{j}\times\mathbf{k}) + u_3v_3(\mathbf{k}\times\mathbf{k}) \\
&= u_1v_1(\mathbf{0}) + u_2v_1(-\mathbf{k}) + u_3v_1(\mathbf{j}) + \\
&\quad u_1v_2(\mathbf{k}) + u_2v_2(\mathbf{0}) + u_3v_2(-\mathbf{i}) + \\
&\quad u_1v_3(-\mathbf{j}) + u_2v_3(\mathbf{i}) + u_3v_3(\mathbf{0}) \\
&= -u_2v_1(\mathbf{k}) + u_3v_1(\mathbf{j}) + u_1v_2(\mathbf{k}) - u_3v_2(\mathbf{i}) - u_1v_3(\mathbf{j}) + u_2v_3(\mathbf{i}) \\
&= -u_3v_2(\mathbf{i}) + u_2v_3(\mathbf{i}) + u_3v_1(\mathbf{j}) - u_1v_3(\mathbf{j}) - u_2v_1(\mathbf{k}) + u_1v_2(\mathbf{k}) \\
&= (u_2v_3-u_3v_2)\mathbf{i} + (u_3v_1-u_1v_3)\mathbf{j} + (u_1v_2-u_2v_1)\mathbf{k}
\end{aligned}
$$

Notice two things about the above derivation: first, it required us to know that $\alpha\mathbf{u}\times\beta\mathbf{v} = \alpha\beta(\mathbf{u}\times\mathbf{v})$ (you should verify this is true by using the definition of the cross product); and second, it required us to compute nine cross products directly, such as $\mathbf{i}\times\mathbf{j}$ (fortunately these cross products are so easy you can do them in your head).

In the end we get the following formula for computing cross products:

$$\mathbf{u}\times\mathbf{v} = |\mathbf{u}||\mathbf{v}|\sin(\theta) = < u_2v_3-u_3v_2, u_3v_1-u_1v_3, u_1v_2-u_2v_1 >$$

It is not very pretty (there is no simple way to remember it), but as with the dot product formula, the cross product formula is very powerful and it gets used all the time in 3D graphics programming.

7.5 Applications of Vectors

Vectors have a wide variety of applications in computer games: they are used in geometry transformation, in animation, in physics simulation, in collision detection, in projecting 3D data onto a two-dimensional screen, in optimizing the performance of 3D games, and much more.

In this section, we will look at some of the most important (but basic) applications, including vector-based representations of lines and planes, useful formulas for calculating distances, and a vector-based method of efficiently rotating, scaling, and skewing points. This is really just a teaser of what vectors are capable of. In the coming lessons we will look at many more applications of vectors.

7.5.1 Representing Lines

We noted in Chapter Six that vectors lead to a very elegant and intuitive way of mathematically representing lines. Suppose we know two points on the line, located by the vectors $\mathbf{p_0}$ and $\mathbf{p_1}$. Then we can define a function from the set of real numbers to the set of all points on the line in the following way:

$P(t) = \mathbf{p_0} + (\mathbf{p_1} - \mathbf{p_0})t$

Here $P(t)$ is the point corresponding with the real number t (actually, it is a vector that describes the point, but that is really a technicality). By varying t from 0 to 1, you get all the points from $\mathbf{p_0}$ to $\mathbf{p_1}$. Notice how this one simple vector equation takes the place of the *three* parametric equations introduced in Chapter Six. Such is the elegance of vector math.

7.5.2 Vectors and Planes

Vectors can be used to describe the orientation of a plane. Such vectors are called *surface normal vectors* or sometimes just *normal vectors*, and are perpendicular to the surfaces of the planes they describe (the word *normal* means *perpendicular*).

Figure 7.14 shows an example of a normal vector.

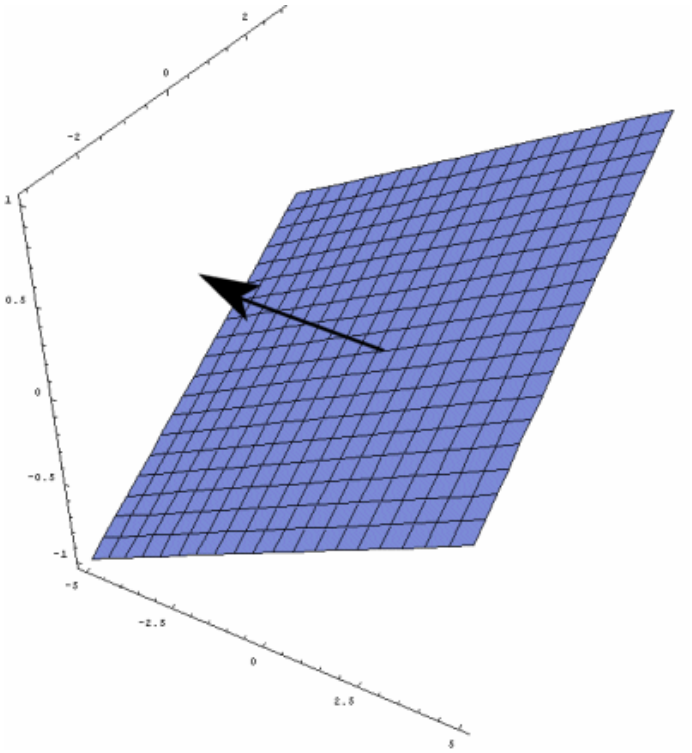


Figure 7.14: A normal vector.

You can create a normal vector given any three unique points on a plane. If you represent these points with the vectors $\mathbf{p_0}$, $\mathbf{p_1}$, and $\mathbf{p_2}$, then you can form two vectors $\mathbf{s} = \mathbf{p_0} - \mathbf{p_1}$ and $\mathbf{t} = \mathbf{p_2} - \mathbf{p_1}$ that both lie on the surface of the plane. The cross product of these vectors is the normal vector of the plane (Fig 7.15).

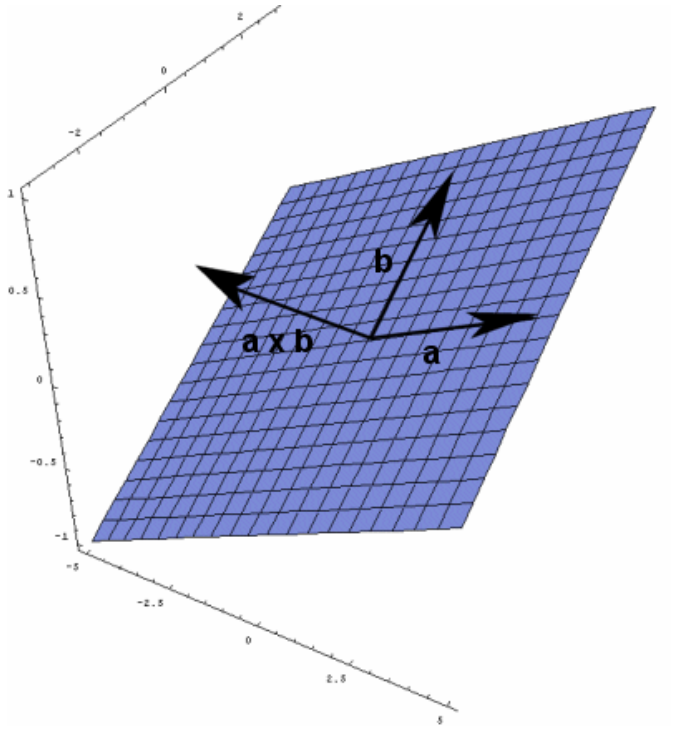


Figure 7.15: The cross product allows you to compute the normal vector for a plane.

Recall that there are two ways to cross the vectors **s** and **t**. The resulting vectors will both be perpendicular to the plane, but they will point in opposite directions. In some cases this will not matter to you, but if your 3D game uses polygons, then which one you choose can be important, as the next section demonstrates.

Backface Culling

If your game uses polygons, then you can describe the orientation of each polygon by using the normal of the polygon's plane (the plane the polygon lies in). Usually, only one side of a polygon can be seen, due to the construction of the game world (walls, for example, are represented with two polygons so that they do not appear infinitely thin, and the inner sides of these two polygons are never seen). If you know whether the surface normal points on the side of the polygon that is seen or the side that is never seen, then you can use an optimization known as *backface culling* to speed up the processing of your game.

Suppose you know that the surface normal of a polygon points on the side of the polygon that the viewer can see. Then you can construct a vector from the viewer to a point on the polygon. Now take the dot product of the polygon's normal vector and that vector that you just created. If the result is positive (indicating the angle between both vectors is 90 degrees or less), then the viewer is looking at the backside of the polygon and you do not need to display it. On the other hand, if the result is negative (indicating the angle between both vectors is greater than 90 degrees), then the viewer is looking at the front side of the polygon and so you *do* need to display it (of course, the polygon may be obstructed by other polygons, but as far as backface culling is concerned, there is no way to eliminate the polygon).

Figure 7.16 shows you what this looks like.

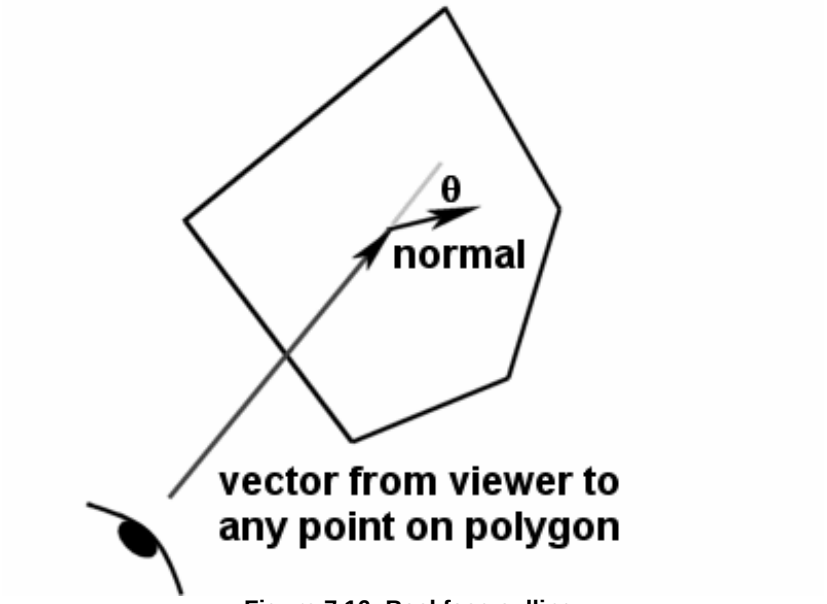


Figure 7.16: Backface culling.

A Vector-Based Representation of Planes

Another useful thing we can do with normal vectors is mathematically represent planes, regardless of their orientations.

Suppose $\mathbf{p_0}$ is a definite, known point on the plane, and $\mathbf{p}$ is *any* point on the plane. Then the vector $\mathbf{s} = \mathbf{p} - \mathbf{p_0}$ lies on the surface of the plane. The angle between this vector and the normal vector is $\pi / 2$ radians (see Figure 7.17). More importantly, the *cosine* of this angle is 0, *hence the dot product is zero also*. If $\mathbf{n}$ is the normal vector, we can express this mathematically as $\mathbf{s} \cdot \mathbf{n} = (\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = \mathbf{p} \cdot \mathbf{n} - \mathbf{p}_0 \cdot \mathbf{n} = 0$. If we let $\mathbf{p} = \langle x, y, z \rangle$, $\mathbf{n} = \langle A, B, C \rangle$, and $D = -\mathbf{p}_0 \cdot \mathbf{n}$, then we can rewrite this equation as follows:

$Ax + By + Cz + D = 0$

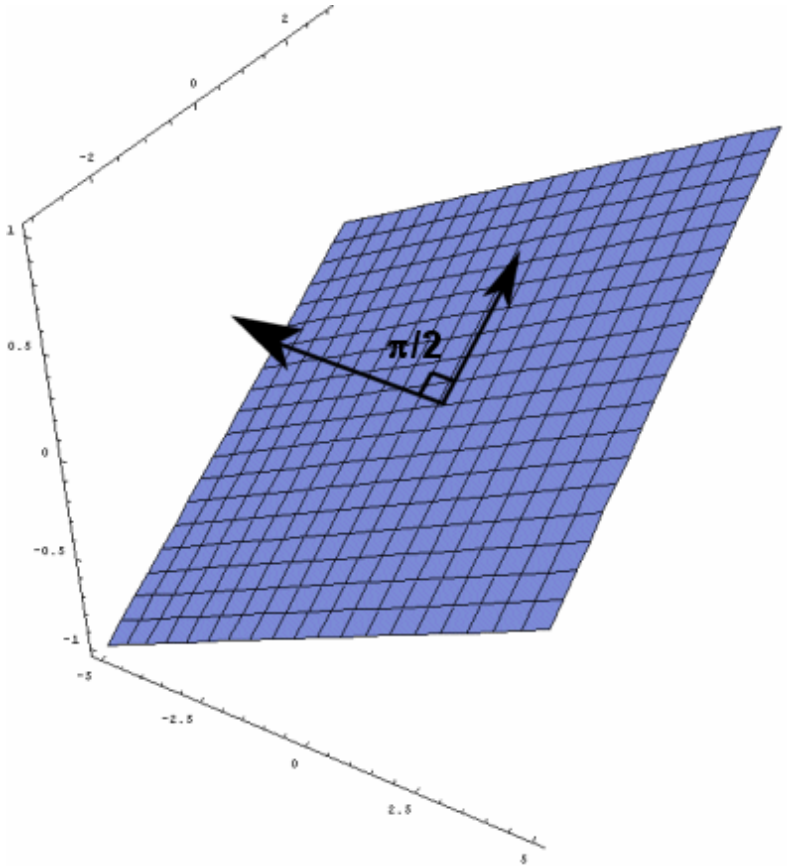


Figure 7.17: The relationship between a vector lying in a plane and the normal of that plane.

This is the famous *general form of the plane equation* mentioned in Chapter Six. This equation tells us that all points (*x*, *y*, *z*) lying on the surface of the plane must satisfy the equation. Unlike the plane equations introduced in Chapter Six, this equation can represent all planes, regardless of orientation.

The plane equation gets used in many ways in game development. For example, if you want to determine which side of a plane a certain point lies on, plug it into the plane equation: the result will be positive for points lying on one side of the plane equation (thanks to the cosine of the angle between the

normal and the vector $\mathbf{p} - \mathbf{p_0}$ being positive), zero for points directly lying on the plane, and negative for points lying on the other side of the plane (thanks to the cosine being negative).

Since it is impossible for *A*, *B*, and *C* to all be zero (otherwise the triplet would be the null vector and describe the orientation of no plane), you can always solve the plane equation for either *x*, *y*, or *z* in terms of the other two. So you can also use the plane equation to generate points on the plane, as was done in Chapter Six.

Another way to represent planes with vectors (which we will not cover here) is to find two non-parallel vectors lying on the surface of the plane. The plane can then be defined as the sum of a point on the plane and all linear combinations of those two vectors.

7.5.3 Distance between Points, Planes, and Lines

Vectors allow you to calculate the distance between a plane and a point, or between a plane and a line. In both cases, "distance" is defined as the minimum distance.

Given a point **p** on a plane with normal **n**, the minimum distance from the plane to the point **q** can be found with the following formula:

$$D = | proj_{\mathbf{n}}(\mathbf{q} - \mathbf{p}) |$$

Figure 7.18 justifies the formula.

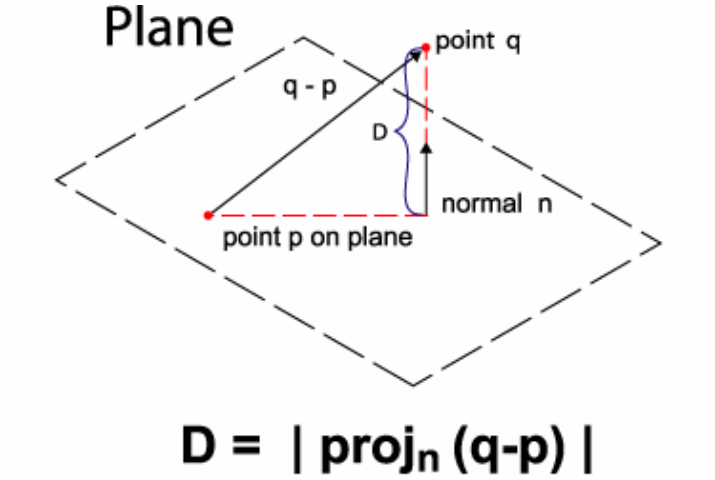


Figure 7.18: Determining the distance between a point and a plane.

Given a point **p** on a line whose direction is given by the vector **u**, the minimum distance from the line to the point **q** can be found with the following formula:

$$D = \frac{|(\mathbf{q} - \mathbf{p}) \times \mathbf{u}|}{|\mathbf{u}|}$$

7.5.4 Rotating, Scaling, and Skewing Points

Vectors can be used to rotate, scale, and *skew* (scale differently for each axis) points efficiently and easily. This is possible all thanks to the magic of bases.

Recall that any vector in 3D space can be represented as a linear combination of three linearly independent 3D vectors. So suppose we have three linearly independent vectors, $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$. Then a given vector **v** in 3D space can be represented as,

$\mathbf{v} = x\mathbf{e_1} + y\mathbf{e_2} + z\mathbf{e_3}$

The amazing thing occurs when you realize that each of the vectors $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$ can, in turn, be represented by a *linear combination of the standard basis vectors* **i**, **j**, and **k**. That is, we can write:

$$\mathbf{e_1} = \alpha_1\mathbf{i} + \alpha_2\mathbf{j} + \alpha_3\mathbf{k}$$
$$\mathbf{e_2} = \beta_1\mathbf{i} + \beta_2\mathbf{j} + \beta_3\mathbf{k}$$
$$\mathbf{e_3} = \gamma_1\mathbf{i} + \gamma_2\mathbf{j} + \gamma_3\mathbf{k}$$

Plugging this into our equation for **v**, we see:

$$\begin{aligned}
\mathbf{v} = \quad & x(\alpha_1\mathbf{i} + \alpha_2\mathbf{j} + \alpha_3\mathbf{k}) + \\
& y(\beta_1\mathbf{i} + \beta_2\mathbf{j} + \beta_3\mathbf{k}) + \\
& z(\gamma_1\mathbf{i} + \gamma_2\mathbf{j} + \gamma_3\mathbf{k}) \\
= \quad & (x\alpha_1 + y\beta_1 + z\gamma_1)\mathbf{i} + \\
& (x\alpha_2 + y\beta_2 + z\gamma_2)\mathbf{j} + \\
& (x\alpha_3 + y\beta_3 + z\gamma_3)\mathbf{k} \\
= \quad & x'\mathbf{i} + y'\mathbf{j} + z'\mathbf{j} \qquad \left(\begin{array}{l} x' = x\alpha_1 + y\beta_1 + z\gamma_1 \\ y' = x\alpha_2 + y\beta_2 + z\gamma_2 \\ z' = x\alpha_3 + y\beta_3 + z\gamma_3 \end{array} \right)
\end{aligned}$$

This result relates a vector <*x*, *y*, *z*> in the $\mathbf{e_1}$, $\mathbf{e_2}$, $\mathbf{e_3}$ basis to a vector <*x*', *y*', *z*'> in the **i**, **j**, **k** basis. If the $\mathbf{e_1}$, $\mathbf{e_2}$, $\mathbf{e_3}$ basis is rotated, scaled, or skewed in relation to the standard basis, then we can effectively use the above equation to rotate, scale, and skew points.

We will not pursue this method because it requires additional vector math in order to produce a useable result, but you should at least be aware of it -- it is in fact, the reason why matrices can be used to transform points.

Conclusion

In our next lesson, we will introduce matrices -- what they are and what operations are defined for them -- with the eventual goal of using matrices to transform points. Your new background in vector math will help immensely, because from a certain point of view, a matrix is really just a list of vectors.

Exercises

1. Perform the following vector operations:

a. $< 1,-4,7 > + < 6,8,2 >$

b. $< -9,1,5 > - < 2,-7,-7 >$

c. $< 4,2,1 > \cdot < 3,3,2 >$

d. $< 5,9,-1 > \times < -8,-2,3 >$

*2. Suppose the player's location is described by the vector **p**, and the player's orientation is described by the vector **v**. If everything the player can see falls within a cone of angle θ, then derive an expression that is positive for points the player can see, and negative otherwise. (You can also use this technique to determine what the non-player characters can see.)

*3. Determine the intersection between a line (in parametric vector form) and a plane (in general form).

*4. One optimization that 3D game programmers use involves looking for polygons that are completely obscured by nearer polygons. These polygons do not need to be displayed since they will not be visible in the final scene. One way you can check to see if one polygon obscures another is by casting out rays from the viewer through the vertices of the first polygon (see Figure E7.1). These rays will form a 3D shape that the second polygon will fall within if it is obscured by the first. Using this information and the general form of the plane equation, devise a method to determine if one polygon obscures another. (This method is too slow for real-time deployment, but the data can be pre-computed for some specially chosen subset of viewer locations; that is precisely what many game engines on the market do.)

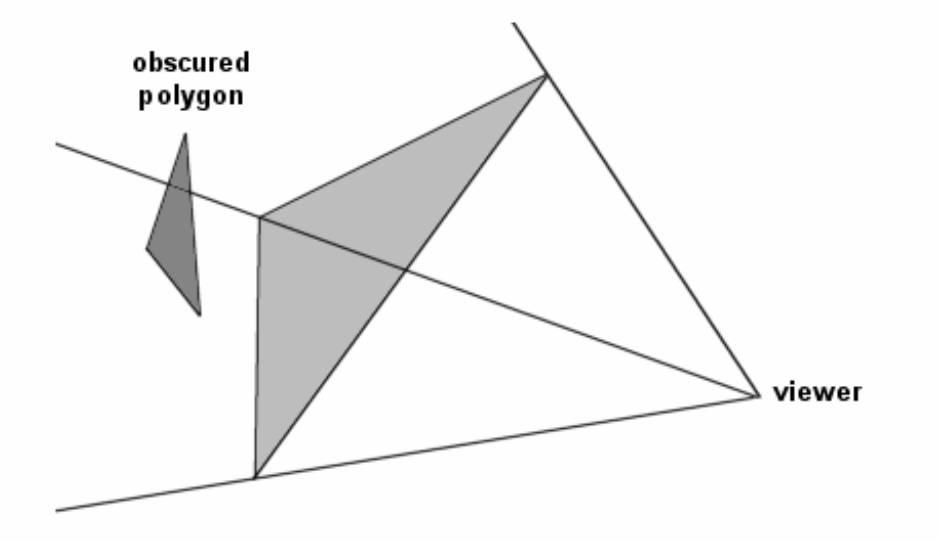


Figure E7.1: One polygon obscures another.

*5. Compute the area of a triangle if two of its edges are spanned by the vectors **u** and **v**.

*6. Using vector projection, compute the distance between a point and a line in parametric vector form. Compare this to the formula given in Chapter Six.

*7. In Chapter Three, you created a polynomial that mapped angles to light intensities. In this problem, you will make that result more useable. Suppose a light source is located at position **p**, with orientation **v**. Further, suppose the light falls on a nearby plane with normal **n**. Using both the general form of the equation for a plane and the polynomial you derived in Chapter Three, create a function that expresses the intensity of the plane as a function of the plane's normal **n**.

*8. One 3D animation technique involves having the artist create a bunch of 3D models (key frames) and then morphing between them to achieve fluid real-time animation. The heart of this method is interpolating between the points in one frame and the points in the next. Suppose the location of the points at the *n*th frame of the animation is given by the vectors $\mathbf{v}_{1n}, \mathbf{v}_{2n}, ..., \mathbf{v}_{3n}$. Using linear interpolation, derive a function that computes the location of the *n*th point at time *t*. (Hint: The function must be defined piecewise.)

*9. Revisit problem 8, only this time, use quadratic interpolation to achieve a smoother fit with the data. Explain why you cannot use quadratic interpolation for both the beginning and the end of the animation. Where does your own quadratic interpolation function fail?

Chapter Eight

Matrix Mathematics I

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2m} & b_2 \\ \vdots & & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} & b_n \end{bmatrix}$$

Introduction

I first started dabbling in computer game programming when I was 8 years old. My Dad purchased a Commodore 64™, which came with a very simple BASIC programming language and a manual designed to introduce kids to the wonderful world of computer programming. The manual described the exploits of the robot Gortek and the other robots of his civilization, illustrating basic computer programming ideas with lavish cartoons and intriguing stories.

The first game I wrote (which I never actually finished) was a simple text-based adventure game with primitive ASCII graphics. About the only thing I can remember about that game is that the setting was a mansion, and that there was some mystery to solve (I was a huge fan of the mystery genre in those days).

A few years later, I graduated to C programming on an IBM PC, where, after spending a year or two learning 2D game programming, I delved into 3D game programming. Although it was far more challenging and engaging than anything I had done prior, at that relatively young age I was still studying algebra, so more often than not I would end up using "black boxes" (equations I did not fully understand) supplied by the authors of books or by fellow game programmers to write my programs.

The biggest of all such black boxes were the equations that involved mathematical entities called "matrices", used in virtually all three-dimensional geometry processing. To me, this was quite amazing: How could a rectangular block of numbers (which is essentially all a matrix is) be used to perform such feats as scaling, rotation, coordinate transformation, and projection?

I eventually found the answer to that question (and many more I did not ask!) in later mathematics courses in college -- but even to this day I have retained that initial sense of wonder at the power of matrices. Like vectors, but to an even greater degree, matrices are simple to define but extraordinarily powerful.

In this chapter we will introduce the concept of a matrix and look at how to add and multiply them, among other operations. In the next chapter we will actually see specifically how matrices are used in game development by introducing the concept of linear transformations. The material we cover in this lesson is necessary for understanding matrices, but will only surface intermittently during actual game development projects.

8.1 Matrices

Fundamentally, a matrix is a table of numbers. Matrices are usually denoted with bold capital letters, such as **A** or **B**. A matrix with *m* rows and *n* columns is called an $m \times n$ matrix.

To explicitly list the contents of a matrix, just draw the table of numbers so that the columns are aligned and then enclose the table in square brackets ('[' and ']').

A few examples of matrices are shown below:

$$\begin{bmatrix} -1 & 0 & 3 \\ 0 & 3 & 2 \\ 3 & 2 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 9 & \pi & -8 \\ -2 & 3 & 6 \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The elements in a matrix have a shorthand notation: if **A** is the matrix, then the element at the *i*th row and *j*th column is typically denoted a_{ij}, $(\mathbf{A})_{ij}$, or even A_{ij}.

The *i*th row of a matrix **A** is typically denoted $\mathbf{A}_i$, and the *j*th column, as $\boldsymbol{A}_j$. The rows and columns can be thought of as one-dimensional matrices (matrices with size $n \times 1$ and $1 \times m$, respectively) or as vectors.

There are two matrices so frequently used that they have special symbols: the *null* or *zero matrix*, denoted by **0** and defined by the rule $a_{ij} = 0$ for all *i* and *j*; and the so-called *identity matrix*, denoted by **I** and defined by the rules $a_{ij} = 1$ for all $i = j$, and $a_{ij} = 0$ otherwise. Only square matrices can be identity matrices, although any size matrix can be a zero matrix.

Here are some examples of identity matrices:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The identity matrix is so named because if you multiply it by any other matrix of the appropriate size, using the definition of matrix multiplication we will introduce later, then the result is equal to that same

matrix. Hence the identity matrix is the matrix equivalent of the number '1', since if you multiply 1 by any number, you get back that same number.

In the sections that follow we will describe some of the things you can do with matrices.

8.1.1 Matrix Relations

You can compare two matrices of the same size to see if they are equal. The equals sign ('=') is used, just like when comparing real numbers or vectors.

By definition, two matrices are defined to be equal if their corresponding elements are equal.

8.1.2 Matrix Operations

Much like real numbers, you can add, subtract, and multiply matrices. Like vectors, you can also do a whole lot more with matrices, including transpose and invert them or take their determinant.

The first operation we will look at is addition.

Addition/Subtraction

You can add two matrices (done with the standard '+' symbol) providing they are the same size. If **A** and **B** are matrices, and **C** = **A** + **B**, then $c_{ij} = a_{ij} + b_{ij}$. In words, the *ij*th element of the sum (the element located at the *i*th row and *j*th column of **C**) is equal to the *ij*th element of **A** plus the *ij*th element of **B**.

Here's an example of matrix addition:

$$\begin{bmatrix} 4 & 2 \\ 1 & -2 \end{bmatrix} + \begin{bmatrix} 5 & 10 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 0 & 0 \end{bmatrix}$$

Matrix addition is both commutative and associative, which comes from the commutativity and associativity properties of real numbers. Also, we can write **A** + **0** = **A** (where **0** is the zero matrix introduced previously), which has parallels in real number arithmetic as well as vector arithmetic.

Subtraction is defined analogously, so that if **A** and **B** are matrices, and **C** = **A** - **B**, then $c_{ij} = a_{ij} - b_{ij}$.

Scalar Multiplication

You can multiply a matrix by a real number (called a scalar, just as with vectors), and scale the entries in the matrix. If **A** is a matrix and c is a scalar, then $(c\mathbf{A})_{ij} = c(\mathbf{A})_{ij}$. That is, the ijth element of the matrix $c\mathbf{A}$ is merely c times the ijth element of the matrix **A**. Scalar division (division of a matrix by a real number) is defined in the same manner.

The following is an example of scalar multiplication:

$$(-2)\begin{bmatrix} -1 & 5 & 8 \\ 0 & -6 & 2 \\ -3 & 2 & 0 \\ 3 & 1 & 1 \\ 6 & -10 & 4 \end{bmatrix} = \begin{bmatrix} -1(-2) & 5(-2) & 8(-2) \\ 0(-2) & -6(-2) & 2(-2) \\ -3(-2) & 2(-2) & 0(-2) \\ 3(-2) & 1(-2) & 1(-2) \\ 6(-2) & -10(-2) & 4(-2) \end{bmatrix} = \begin{bmatrix} 2 & -10 & -16 \\ 0 & 12 & -4 \\ 6 & -4 & 0 \\ -6 & -2 & -2 \\ -12 & 20 & -8 \end{bmatrix}$$

Note that $c\mathbf{0} = \mathbf{0}$, for any scalar c.

Matrix Multiplication

The most difficult of the elementary operations to perform is matrix multiplication -- the product of two matrices. Based on the way addition and subtraction were defined, you might suspect that for two matrices **A** and **B**, the product **AB** would be a matrix **C** such that $c_{ij} = a_{ij}b_{ij}$. It is certainly possible to define matrix multiplication like this, but doing so is not entirely helpful.

It turns out that the best way to define matrix multiplication is not anything like the definitions of matrix addition or subtraction. It is also a difficult definition to master, and quite tedious to use if you need to multiply moderately sized matrices by hand. But the definition goes a long way: the number of applications for this particular definition of matrix multiplication is enormous (using matrices, you can solve systems of linear equations, transform geometry, or perform any linear transformation, for example).

The first thing to note is that you cannot multiply just any two matrices together, regardless of their dimensions. Now you *can* multiply any square matrices together (of the same size), but the definition goes beyond that and allows you to multiply matrices of *different* sizes together, provided that they meet certain requirements. What requirements? Well if you want to compute the matrix product **AB** (which, note, is not the same as **BA**), then **A** must be an $m \times n$ matrix, and **B** must be an $n \times p$ matrix; m and p can be anything you want, but notice that the number of columns in **A** must be equal to the number of rows in **B**. If this is not the case, then you cannot multiply the two matrices together.

If we denote the size of a matrix with a subscript, then we can restate the above rule by saying that you can only compute products of the form $\mathbf{A}_{\mathbf{m}\times\mathbf{n}}\mathbf{B}_{\mathbf{n}\times\mathbf{p}}$ (the n's are on the inside and close together -- this should help you remember the requirement).

The product **AB** is another matrix called **C**. The size of **C** is $m \times p$; it has the same number of rows as **A**, but the same number of columns as **B**. So using the subscript notation again, we can write: $\mathbf{C_{m\times p}} = \mathbf{A_{m\times n}B_{n\times p}}$. For square matrices of dimension *n*, this reduces to, $\mathbf{C_{n\times n}} = \mathbf{A_{n\times n}B_{n\times n}}$.

The *ij*th element of **C** is defined as the dot product of the *i*th row of **A** and the *j*th column of **B**. Dot product, you ask? By dot product, we mean that you should think of both the *i*th row of **A** and the *j*th column of **B** as vectors, and then compute the dot product of these vectors as discussed in the last chapter. The result goes into the *ij*th element of **C**.

In symbols, we can describe the *ij*th element of **C** as follows:

$$c_{ij} = \mathbf{A}_i \cdot \mathbf{B}^j = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

The notation on the far right side is shorthand for indicating summation: it can be read, "the sum of all $a_{ik}b_{kj}$ for *k* from 1 to *n*." You can think of it as a simple FOR loop in mathematics.

Here are some examples of matrix multiplication:

$$\begin{pmatrix} -1 & -1 \\ -1 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & -4 & 1 & 2 \end{pmatrix} = \begin{pmatrix} -1 & 4 & -1 & -2 \\ 2 & -8 & 2 & 4 \\ 1 & -4 & 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 & -2 & 1 & 2 \\ -2 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & -3 & 1 \\ 1 & -2 & 1 \\ 2 & 3 & 0 \\ 1 & -3 & -3 \end{pmatrix} = \begin{pmatrix} 6 & -5 & -6 \\ 0 & 4 & -4 \end{pmatrix}$$

$$\begin{pmatrix} -1 & 0 & 1 & 1 \\ -3 & 2 & 2 & -3 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ -2 & -2 \\ 2 & -3 \\ -1 & -2 \end{pmatrix} = \begin{pmatrix} 3 & -6 \\ 9 & -7 \end{pmatrix}$$

$$\begin{pmatrix} -2 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -2 \\ 0 & 1 \end{pmatrix}$$

Matrix multiplication is *not* commutative: **AB** is not, in general, the same as **BA** (you can easily see this by noting that, to begin with, these matrices are usually not even the same size!). However, matrix multiplication is associative and also distributive with respect to matrix addition (or subtraction). With the distributive law, however, you must be careful, since while it is true you can write **A**(**B** + **C**) as **AB** + **AC**, you cannot write it as **BA** + **CA**. Similarly, you can write (**B** + **C**)**A** as **BA** + **CA**, but not as **AB** + **AC**.

Notice that as mentioned before, the identity matrix times any matrix equals that matrix.

You can raise a matrix to a non-zero integer power *n* by multiplying the matrix by itself *n* times.

Transpose

The transpose of a matrix **A**, denoted $\mathbf{A}^\mathbf{T}$, is a matrix such that $(\mathbf{A}^\mathbf{T})_{ij} = (\mathbf{A})_{ji}$. One of the implications of this definition is that if **A** is $m \times n$, then $\mathbf{A}^\mathbf{T}$ is $n \times m$.

The transpose of a simple matrix is shown below:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & -2 & 5 \end{bmatrix}^\mathbf{T} = \begin{bmatrix} 1 & 3 \\ 0 & -2 \\ 0 & 5 \end{bmatrix}$$

Here are a few important properties of the transpose operation:

$(\mathbf{A}^\mathbf{T})^\mathbf{T} = \mathbf{A}$
$(\mathbf{A} + \mathbf{B})^\mathbf{T} = \mathbf{A}^\mathbf{T} + \mathbf{B}^\mathbf{T}$
$(\mathbf{AB})^\mathbf{T} = \mathbf{B}^\mathbf{T}\mathbf{A}^\mathbf{T}$

The proofs of these properties are quite easy and follow directly from the definition of the transpose.

Determinant

The determinant of a matrix **A** is a function, denoted by det(**A**). The domain of the determinant function is the set of all *square* matrices, and the range is the set of all real numbers. Thus the determinant function associates a real number with every square matrix, a real number that tells you something about that matrix.

A whole chapter could be spent discussing exactly what the determinant is, and the various (rather complicated) ways to calculate it. But for our purposes, it suffices to know just this: that the determinant evaluates to non-zero if the rows of the matrix are linearly independent, as defined by vector math, and zero otherwise.

If you need to calculate the determinant of a matrix (you should not typically need to; the determinant is mainly used in proofs rather than in applications), then you can always use a CAS (Computer Algebra System) or a matrix library (such as that supplied with this course).

Inverse

You will notice that we have not said a word about division of one matrix by another: that is because such an operation does not exist (it would not be possible to assign any meaning to division in the general case).

However, to take its place in certain situations, we do have the *inverse*. The inverse of a square matrix **A**, denoted $\mathbf{A}^{-1}$, is a matrix such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ (or $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ -- in this special case multiplication *does* commute).

Not all square matrices have inverses. In order for a given square matrix to have an inverse, its determinant must be non-zero.

Important properties of inverses are as follows:

$(\mathbf{A}^{-1})^{-1} = \mathbf{A}$
$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$
$(\mathbf{A}^{T})^{-1} = (\mathbf{A}^{-1})^{T}$

There are a number of ways to calculate the inverse of a matrix, but as with the determinant, in this course we will not explore the very complex mathematics involved. Suffice to say that most math libraries (including the one that ships with this course) will calculate the inverse of a matrix for you. Inverse matrices are often used to cancel out the effects of a set of operations that has taken place and are stored in the original matrix. The goal is to allow one to use the matrix to enter the local coordinate space of a given entity. Since we have not talked about transformations or local coordinate spaces yet, we will save this discussion for the next lesson.

8.2 Systems of Linear Equations

You may wonder what matrices are actually used for. The full answer to that question will have to wait until the next chapter when we discuss the topic of linear transformations, but for now we can at least look at what matrices were originally intended to do. This will provide some context to some of the matrix operations, as well as provide a foundation upon which physics and certain areas of computer science can build on (linear transformations are the primary use of matrices in game development, but certainly not the only one).

A system of n linear equations in m unknowns is a set of *n* linear equations, each of which involves one or more of the *m* variables. For example, the following is a system of two linear equations in two unknowns:

$x + y = 2$
$3y = 5$

Allowing zero coefficients, we can write all systems of linear equations in the following form:

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \\
&\vdots \\
&a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n
\end{aligned}
$$

where the *x*'s are the unknowns. Cast in the above form, we can represent this system of linear equations with the following matrix equation:

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1m} \\
a_{21} & a_{22} & \cdots & a_{2m} \\
\vdots & & & \\
a_{n1} & a_{n2} & \cdots & a_{nm}
\end{bmatrix}
\begin{bmatrix}
x_1 \\ x_2 \\ \vdots \\ x_n
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\ b_2 \\ \vdots \\ b_n
\end{bmatrix}
$$

If there are more unknowns than equations, then the system does not have a unique solution. If there are more equations than unknowns, then the system might have a solution, but then again, maybe not (you cannot impose more constraints than you have unknowns). If there are as many equations as unknowns ($m = n$, so the coefficient matrix – the matrix on the left that stores the coefficients -- is square), then providing the rows of the coefficient matrix are linearly independent, the system has a unique solution.

One way to solve a system with a unique solution is to find the inverse of the coefficient matrix, and then multiply both sides by that. This gets the job done, but it is inefficient, and can only be used for systems that have a single solution (as opposed to many solutions).

A better way involves the so-called *augmented matrix of the system*. This matrix is what you get when you add the column of *b*'s to the coefficient matrix. The augmented matrix of the above system is shown below:

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1m} & b_1 \\
a_{21} & a_{22} & \cdots & a_{2m} & b_2 \\
\vdots & & & & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nm} & b_n
\end{bmatrix}
$$

Once you create this matrix, solving the system involves the application of a process known as *Gaussian elimination* on the matrix.

8.2.1 Gaussian Elimination

Consider the system of *n* equations in *m* unknowns:

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \\
&\vdots \\
&a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n
\end{aligned}$$

Gaussian elimination is based on the following three principles:

1. You can multiply (or divide) any equation by a non-zero real number without changing the solution set.
2. You can swap two equations without changing the solution set.
3. You can multiply (or divide) any equation by a non-zero real number, and then add the resulting equation to another equation without changing the solution set.

The first two principles are obviously true (we use them all the time in algebra when solving for unknowns), but the third one may not be, so we will take some time to explain it now.

Suppose we have the following two equations:

$a + b = c$

$d + e = f$

Then we can add the first equation to the second as follows:

$(a + b) + (d + e) = c + f$

This is justified because we are adding the same quantity to both sides of the second equation -- namely, the quantity $(a + b)$, which is the same number as c.

The three principles above also apply to augmented matrices, since augmented matrices are just systems of linear equations written in compact form. Restated with matrices in mind, the principles become:

1. You can multiply (or divide) any row by a non-zero real number without changing the solution set.
2. You can swap two rows without changing the solution set.
3. You can multiply (or divide) any row by a non-zero real number, and then add the resulting row to another row without changing the solution set.

Gaussian elimination can now be defined as the process of using these three operations (referred to as *elementary row operations*) on an augmented matrix in order to find the solution set. For a square

matrix, the easiest way to do this is to manipulate the augmented matrix so that it is in the following form (called *echelon form*):

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} & b'_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & b'_2 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & a'_{nn} & b'_n \end{bmatrix}$$

The associated system of equations for the above matrix looks like this:

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{(n-1)(n-1)}x_{(n-1)} + a'_{(n-1)n}x_n &= b'_{n-1} \\ a'_{nn}x_n &= b'_n \end{aligned}$$

We can solve this system by using a process known as *back-substitution*. First you solve for x_m, using the last equation, which tells you $x_n = b'_n/a'_{nn}$. Then you substitute this into the second-to-last equation, and solve for x_{n-1}. You repeat the process until you have solved for all the unknowns. Of course, this process can fail if certain coefficients are zero. For example, if $a'_{nn} = 0$, then since division by zero is undefined, you cannot compute x_n. More generally, if a'_{ii} is zero for any i, then you cannot solve the system for all the unknowns.

For non-square matrices, the process works much the same, except you may either get a contradiction (it is possible if there are more equations than unknowns) or you will not be able to solve for all unknowns (if there are more unknowns than equations). In the latter case, you can still solve for *some* of the unknowns *in terms of the other unknowns*.

For an example of the latter, consider the simple system $x + y = 1$. Solving for x, we get $x = 1 - y$. This is all we can say about x, since we have no additional equation that relates y to x (if we did, we could solve this equation for y and plug the result into the equation $x = 1 - y$ to determine the value of x). Thus the system $x + y = 1$ has no single solution for x; rather, it has an *infinite number of solutions* -- one for each value of y.

As game developers, unsolvable systems, and non-square matrices in general, will not be of concern to us. We are interested in problems we can solve for *all* the unknowns and get *definite* values from -- that is, in the rare case we are interested in solving systems of equations at all (physics simulation calls for this sometimes, but usually game developers use matrices in other ways).

Conclusion

That concludes our rather brief introduction to the world of matrix math. In our next lesson, we will study an alternate way of looking at matrix math that provides the basis for virtually every three-dimensional game on the market.

Exercises

1. Compute the following matrix operations:

$$a.\ \begin{bmatrix} 1 & -5 \\ 4 & 2 \end{bmatrix} + \begin{bmatrix} 1 & -6 \\ 6 & -3 \end{bmatrix} = ???$$

$$b.\ 4\begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} = ???$$

$$c.\ \begin{bmatrix} 4 & 6 & 2 \\ -3 & 2 & -3 \\ 8 & 5 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 4 \\ 2 \end{bmatrix} = ???$$

*2. Simulating the physics of cloth is one of many uses for matrices. Read the paper located at http://www.cs.cmu.edu/~baraff/papers/sig98.pdf and report on your findings. You will not understand everything, since the paper assumes you know calculus, but the paper should give you an idea of the sheer breadth of applications that matrices have. The Cloth Animation seminar here at Game Institute will also provide you with some interesting insight (although you may wish to wait until you have completed the Graphics Programming course series before enrolling in the seminar).

*3. Compute the intersection of two lines in general form using Gaussian elimination.

!4. Markov chains allow you to represent the probability (in the range of [0, 1]) of transitioning from one state to another with a matrix. This has applications in artificial intelligence, where a computer character may have a number of different states (such as attacking, retreating, defending -- or even emotional states, such as happy, sad, angry), and different probabilities of transitioning from one state to another. Read the paper located at http://www.mech.bee.qut.edu.au/men170/smsch6.html and construct two matrices to model a character's behavior. One matrix should be the initial state of the character, and the other should be the transition probability matrix. (Note the author of the paper uses the period '.' symbol to designate matrix multiplication.)

*5. Take the matrix you constructed in problem 4, and multiply it by itself 10 times (use a Computer Algebra System if you have access to one). Did the matrix change? If so, how did it change? What does this new matrix represent, anyway?

*6. Gaussian elimination is exceptionally adept at solving resource allocation problems. Suppose you are designing a resource management game and that your income comes from four species: the Grendals, the Hymlocks, the Wyrotts, and the Shails. Suppose the user wants to collect 10 million credits from all four species combined, but that the species will tolerate differing levels of taxation according to the following rules: the Grendals are greedy and will only pay half of what the Hymlocks pay; the Wyrotts and the Shails have formed an alliance and will pay only 4 million combined; the Hymlocks and Shails are small populations, and cannot afford to pay more than 2 million combined. Using Gaussian elimination, find out how much each species pays. Which species pays zero taxes?

Chapter Nine

Matrix Mathematics II

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Introduction

In our last lesson we learned the ground rules for matrix math, and were shown how matrices can be used to solve systems of linear equations. As it turns out, that is just one of the many ways we can use matrices. Another use, more relevant to computer game programming, involves the concept of linear transformations.

In this lesson we will talk about linear transformations and how matrices can be used when performing them. Then we will see a wide variety of matrices that can be used to rotate, translate, scale and skew points in any conceivable way.

9.1 Linear Transformations

Suppose $L(\mathbf{v})$ is a function whose domain is the set of n dimensional vectors, and whose range is the set of m dimensional vectors. If $L(\mathbf{v})$ obeys the following two properties (where c and d are scalars and $\mathbf{u}$ and $\mathbf{v}$ are n dimensional vectors):

$L(c\mathbf{v}) = cL(\mathbf{v})$
$L(\mathbf{u} + \mathbf{v}) = L(\mathbf{u}) + L(\mathbf{v})$

then L is called a *linear transformation from* $\boldsymbol{R^n}$ *to* $\boldsymbol{R^m}$ (the set of ordered real n-tuples to the set of ordered real m-tuples).

Let us look at few examples of linear transformations. Suppose we define a function $f(\mathbf{v}) = 4v_1\mathbf{i} + (v_1+v_2)\mathbf{j}$, where $\mathbf{v}$ is the vector $v_1\mathbf{i} + v_2\mathbf{j}$. Is this a linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$? To determine that, we have to see if it satisfies the two properties listed above.

For a scalar c, $c\mathbf{v} = c(v_1\mathbf{i} + v_2\mathbf{j}) = cv_1\mathbf{i} + cv_2\mathbf{j}$. So f($c\mathbf{v}$) = $4cv_1\mathbf{i} + (cv_1+cv_2)\mathbf{j} = c(4v_1\mathbf{i} + (v_1+v_2)\mathbf{j})$. Now $cf(\mathbf{v})$, on the other hand, is $c(4v_1\mathbf{i} + (v_1+v_2)\mathbf{j}$. Thus $f(c\mathbf{v}) = cf(\mathbf{v})$, so the function satisfies the first property.

Let us check the second property for vectors $\mathbf{u}$ and $\mathbf{v}$,

$$f(\mathbf{u}) + f(\mathbf{v}) = (4v_1\mathbf{i} + (v_1 + v_2)\mathbf{j}) + (4u_1\mathbf{i} + (u_1 + u_2)\mathbf{j})$$

$$= (4v_1 + 4u_1)\mathbf{i} + ((v_1 + v_2) + (u_1 + u_2))\mathbf{j}$$

$$= 4(v_1 + u_1)\mathbf{i} + ((v_1 + u_1) + (v_2 + u_2))\mathbf{j}$$

On the other hand, $\mathbf{u} + \mathbf{v} = (v_1 + u_1)\mathbf{i} + (v_2 + u_2)\mathbf{j}$. So $f(\mathbf{u} + \mathbf{v}) = 4(v_1 + u_1)\mathbf{i} + ((v_1 + u_1) + (v_2 + u_2))\mathbf{j}$. This is the same as $f(\mathbf{u}) + f(\mathbf{v})$, so we can conclude that f satisfies the second property as well.

Thus, f is a linear transformation from $\mathbf{R^2}$ to $\mathbf{R^2}$.

Consider another transformation g that takes vectors in $\mathbf{R}^3$ and drops the "z" component, thus bringing them into $\mathbf{R}^2$. Mathematically, you can express this as $g(\mathbf{v}) = v_1\mathbf{i} + v_2\mathbf{j}$, where $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. This transformation can be viewed as a projection of vectors onto the *x*-*y* plane, as shown in Figure 9.1.

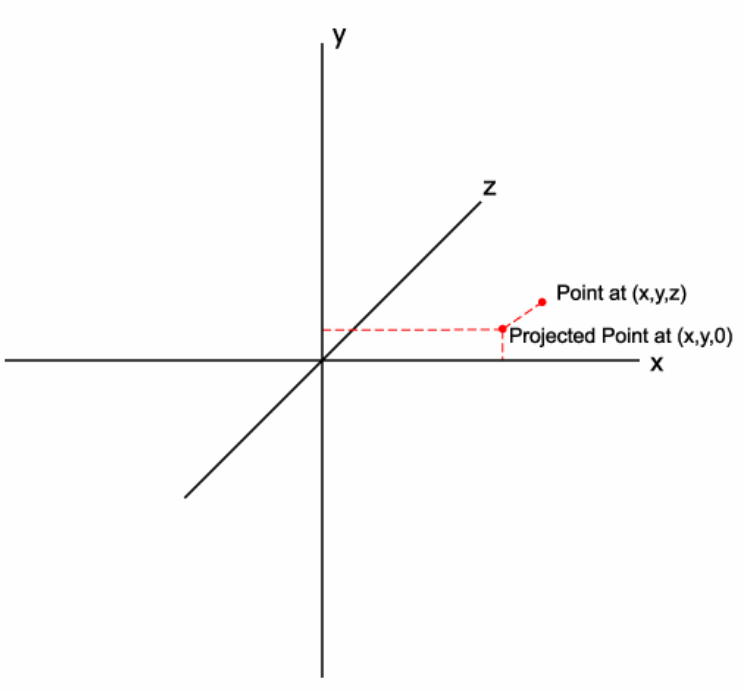


Figure 9.1: Projecting vectors onto the *x*-*y* plane.

Is this a linear transformation from $\mathbf{R}^3$ to $\mathbf{R}^2$? The answer is yes, although it will be left to you to verify that mathematically (or by thinking about the transformation as a projection).

Another example of a linear transformation is rotation of a vector around the origin of a coordinate system. Scaling and skewing are also linear transformations. All three of these operations are very important for computer games.

Translation of points (represented by vectors) is also important, but unfortunately, translation is not a linear transformation. This is an important point, so we will prove it here. Let $T(\mathbf{v})$ be a function from $\mathbf{R}^3$ to $\mathbf{R}^3$ that adds t_x, t_y, and t_z onto the x, y, and z components of the vector $\mathbf{v}$, respectively. Thus $T(\mathbf{v}) = (v_1 + t_x)\mathbf{i} + (v_2 + t_y)\mathbf{j} + (v_3 + t_z)\mathbf{k}$.

Now $c\mathbf{v} = c(v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}) = cv_1\mathbf{i} + cv_2\mathbf{j} + cv_3\mathbf{k}$. So $T(c\mathbf{v}) = (cv_1 + t_x)\mathbf{i} + (cv_2 + t_y)\mathbf{j} + (cv_3 + t_z)\mathbf{k}$. On the other hand, $cT(\mathbf{v}) = c(v_1 + t_x)\mathbf{i} + c(v_2 + t_y)\mathbf{j} + c(v_3 + t_z)\mathbf{k} = (cv_1 + ct_x)\mathbf{i} + (cv_2 + ct_y)\mathbf{j} + (cv_3 + ct_z)\mathbf{k}$. Clearly then, the function T is not a linear transformation.

The application to matrices of this discussion of linear transformation is simple, but profound. *Any linear transformation whatsoever from $\mathbf{R}^n$ to $\mathbf{R}^m$ can be represented by an $m \times n$ matrix*. To transform a vector, just represent it as an $n \times 1$ matrix, and compute $\mathbf{Lv}$, where $\mathbf{L}$ is the linear transformation matrix (sometimes called the *induced matrix*, since it is induced by the linear transformation function) and $\mathbf{v}$ is the vector matrix. The result is the transformed vector in $\mathbf{R}^m$.

The critical question now is how to compute a matrix that corresponds to a given linear transformation function. That is the topic of the next section.

9.1.1 Computing Linear Transformation Matrices

Suppose we have a linear transformation function T that maps vectors from $\mathbf{R^n}$ to $\mathbf{R^m}$. Further, let us suppose we represent vectors as *column matrices* (matrices with only one column). Then we can express any given vector v in $\mathbf{R^n}$ as shown below:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} = v_1\mathbf{i}_1 + v_2\mathbf{i}_2 + v_3\mathbf{i}_3 + \cdots + v_n\mathbf{i}_n = v_1\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + v_2\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + v_n\begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Now let us apply the transformation to **v**, using the fact that T is a linear transformation:

$$\begin{aligned} T(\mathbf{v}) &= T(v_1\mathbf{i}_1 + v_2\mathbf{i}_2 + v_3\mathbf{i}_3 + \cdots + v_n\mathbf{i}_n) \\ &= T(v_1\mathbf{i}_1) + T(v_2\mathbf{i}_2) + T(v_3\mathbf{i}_3) + \cdots + T(v_n\mathbf{i}_n) \\ &= v_1T(\mathbf{i}_1) + v_2T(\mathbf{i}_2) + v_3T(\mathbf{i}_3) + \cdots + v_nT(\mathbf{i}_n) \end{aligned}$$

All the $T(\mathbf{i}_j)$'s are $m \times 1$ column matrices. They are the m dimensional transformations of the n dimensional standard basis vectors. With this in mind, if you study the above equation for a while, you will see that we can represent $T(\mathbf{v})$ (itself an $m \times 1$ column matrix) as the product of an $m \times n$ matrix **L** and the vector **v**, where the jth column of **L** is $T(\mathbf{i}_j)$. This product can be represented with the following notation:

$$\begin{aligned} T(\mathbf{v}) &= [T(\mathbf{i}_1) \;\vdots\; T(\mathbf{i}_2) \;\vdots\; T(\mathbf{i}_3) \;\vdots\; \cdots \;\vdots\; T(\mathbf{i}_n)]\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix} \\ &= \mathbf{L}\mathbf{v} \end{aligned}$$

(The symbol '⋮' merely separates the columns.)

With very little effort, we have found the induced matrix of the linear transformation T. Moreover, we have also indirectly proved that all linear transformations have induced matrices (a statement we made in the last section without proof), since we found exactly what those matrices look like!

To summarize these results, if *T* is a linear transformation that maps vectors from $\mathbf{R^n}$ to $\mathbf{R^m}$, then to find the corresponding matrix, we transform the standard basis vectors, and use the transformed vectors as columns in a matrix. This matrix is the induced matrix of **T**.

To solidify this process in your mind, let us look at an example.

In the last section, we introduced the linear transformation $g(\mathbf{v}) = v_1\mathbf{i} + v_2\mathbf{j}$, where $\mathbf{v} = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k}$. This is a transformation from $\mathbf{R^3}$ to $\mathbf{R^2}$, so the induced matrix will be a 2×3 matrix. The columns of this matrix will be the transformed standard basis vectors for $\mathbf{R^3}$. The standard basis vectors for $\mathbf{R^3}$ are, in matrix form:

$$\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \text{ and } \begin{bmatrix}0\\0\\1\end{bmatrix}.$$

Sending all of these vectors to *g*, our transformation function, we get the following vectors in $\mathbf{R^2}$:

$$\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \text{ and } \begin{bmatrix}0\\0\end{bmatrix}.$$

These form the columns of our 2×3 matrix. So our induced transformation matrix is:

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\end{bmatrix}.$$

Let us check to make sure this is correct by multiplying the matrix by the vector **v**:

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\end{bmatrix}\begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix} = \begin{bmatrix}v_1\\v_2\end{bmatrix} = v_1\mathbf{i} + v_2\mathbf{j} = g(\mathbf{v})$$

It worked! We found the induced matrix of *g*.

These are impressive results. Shortly, we are going to take them and apply them to all the common transformations that computer games require. But first, there is one issue we need to address – translation. We saw that translation is not a linear transformation, but it is, nevertheless, extremely important in game programming. In the next section we will address just this issue.

9.1.2 Resistance is Futile: Making Translation Comply

The first step in finding some way of making translation a linear transformation is figuring out why it is not *already* a linear transformation. In the last section, we calculated both $T(c\mathbf{v})$ and $cT(\mathbf{v})$ (where T was our translation transformation function), and found the following:

$$T(c\mathbf{v}) = (cv_1 + t_x)\mathbf{i} + (cv_2 + t_y)\mathbf{j} + (cv_3 + t_z)\mathbf{k}$$

$$cT(\mathbf{v}) = (cv_1 + ct_x)\mathbf{i} + (cv_2 + ct_y)\mathbf{j} + (cv_3 + ct_z)\mathbf{k}$$

As you can see, $T(c\mathbf{v})$ and $cT(\mathbf{v})$ differ in that $T(c\mathbf{v})$ lacks a factor of c in the translation terms t_x, t_y, and t_z. This problem stems from our definition of T, which we defined as $(v_1 + t_x)\mathbf{i} + (v_2 + t_y)\mathbf{j} + (v_3 + t_z)\mathbf{k}$. When we compute $cT(\mathbf{v})$, we multiply all of the terms by c, which gives rise to a factor of c in both the v's and the t's. But when we compute $c\mathbf{v}$, we introduce a factor of c only in the v's. Essentially, what is going wrong is that the terms t_x, t_y, and t_z are not multiplied by any of the v's.

We can solve this problem, but it requires a slight leap of faith.

Suppose for a moment that $\mathbf{v}$ is not a three-dimensional vector, but a *four-dimensional* vector. Further suppose that we define the translation function as follows:

$$T(\mathbf{v}) = (v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k}$$

Now what happens when we compute $T(c\mathbf{v})$? We get the following result:

$$c\mathbf{v} = c((v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k}) = (cv_1 + ct_x v_4)\mathbf{i} + (cv_2 + ct_y v_4)\mathbf{j} + (cv_3 + ct_z v_4)\mathbf{k}$$

This introduces a factor of c in the v's, which is good, but we still need to know what $cT(\mathbf{v})$ is under this new definition of T. This computation is shown below:

$$cT(\mathbf{v}) = c((v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k}) = (cv_1 + ct_x v_4)\mathbf{i} + (cv_2 + ct_y v_4)\mathbf{j} + (cv_3 + ct_z v_4)\mathbf{k}$$

The result is exactly the same! So we have established that under our new definition of T, $T(c\mathbf{v}) = cT(\mathbf{v})$ (You can verify on your own that $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$). We are not done yet, however, since it is unclear how we can use this result (which assumes we send the function a four-dimensional vector) to translate three-dimensional vectors.

The new transformation equation maps from $\mathbf{R}^4$ to $\mathbf{R}^3$. We defined the new transformation as:

$$(v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k}$$

where the old transformation was:

$$(v_1 + t_x)\mathbf{i} + (v_2 + t_y)\mathbf{j} + (v_3 + t_z)\mathbf{k}.$$

You can see that if $v_4 = 1$ (that is, if the fourth component of our 4D vector is 1), then the new transformation reduces to the old one. To confirm this, let us transform the vector $\langle v_1, v_2, v_3, 1\rangle$:

$$\begin{aligned} T(\langle v_1, v_2, v_3, 1\rangle) &= (v_1 + t_x(1))\mathbf{i} + (v_2 + t_y(1))\mathbf{j} + (v_3 + t_z(1))\mathbf{k} \\ &= (v_1 + t_x)\mathbf{i} + (v_2 + t_y)\mathbf{j} + (v_3 + t_z)\mathbf{k} \\ &= \langle v_1 + t_x, v_2 + t_y, v_3 + t_z\rangle \end{aligned}$$

This is the precise result we needed! What we have learned from all this is that if we represent our 3D vectors $\langle v_1, v_2, v_3\rangle$ in the form $\langle v_1, v_2, v_3, 1\rangle$, then translation of the 3D vector becomes a linear transformation. That means it has an induced matrix.

The size of the induced matrix is 3×4. A square matrix is nicer to work with than a non-square one (a square matrix can have a determinant and an inverse, for example), so most games do not actually use 3×4 matrices to translate points. Rather, they use 4×4 matrices that map from $\mathbf{R}^4$ to $\mathbf{R}^4$. The following definition of T is one way of defining translation for such a matrix:

$$T(\mathbf{v}) = (v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k} + v_4\mathbf{l}$$

where $\mathbf{l}$ is the vector $\langle 0, 0, 0, 1\rangle$ (the fourth dimensional standard basis vector). This is a nice definition because the resulting induced matrix is invertible (the rows are linearly independent).

Since the transformation results in a 4D vector, and all you really need is a 3D vector, you can just ignore the fourth component after you have transformed your vector.

This method of incorporating translation into matrices is not new -- it has been known for decades. As a result, there is some information you might want to know: the 4D vectors in the form $\langle v_1, v_2, v_3, 1\rangle$ are called *homogenous coordinates*. The 4th component is often referred to as the w coordinate (since the letters x, y, and z are already taken). Although translation requires that $w = 1$, there are some cases where other values of w are allowed. In these cases, to convert to a 3D vector, just divide the vector by w, which will again put the vector in the form $\langle v_1, v_2, v_3, 1\rangle$ (the first three components are the coordinates of the vector in three dimensions). This is called *homogenizing* the coordinate.

With translation out of the way, we are now ready to compute the induced matrices for a wide variety of common operations.

9.2 Common Transformation Matrices

The operations commonly performed in 3D games include scaling, skewing, translation, rotation and projection. For reasons that will soon become clear, it is helpful to perform all of these operations using the same size matrices. Translation is the only operation that requires a 4×4 matrix; the others can be implemented with 3×3 matrices, but can still be made to work with 4×4 matrices. For that reason, we are going to make 4×4 matrices the standard for all operations. This means that all of our vectors must be four-dimensional, and as we learned in the last section, the 4th component of our vectors must be 1, or the translation matrix fails to translate.

9.2.1 The Scaling Matrix

The scaling function takes a vector and scales its *x*, *y*, and *z* components by some scalar. We can define such a function as $S(\mathbf{v}) = sv_1\mathbf{i} + sv_2\mathbf{j} + sv_3\mathbf{k} + v_4\mathbf{l}$, where *s* is the scaling constant.

We can create the induced matrix of this transformation by applying the transformation function to the columns of the following matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(The columns are just the standard basis vectors.)

The result is the following matrix:

$$\mathbf{S} = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can then scale any vector **v** by computing the product **Sv**.

9.2.2 The Skewing Matrix

Skewing is just non-symmetric scaling. That is, each axis is scaled by its own constant. Our linear transformation function will then look something like $K(\mathbf{v}) = s_x v_1\mathbf{i} + s_y v_2\mathbf{j} + s_z v_3\mathbf{k} + v_4\mathbf{l}$.

Applying this transformation to the columns of the 4×4 identity matrix, we obtain the following result:

$$\mathbf{K} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You can skew a vector **v** by computing **Kv**.

9.2.3 The Translation Matrix

Earlier we uncovered the form of the translation function:

$T(\mathbf{v}) = (v_1 + t_x v_4)\mathbf{i} + (v_2 + t_y v_4)\mathbf{j} + (v_3 + t_z v_4)\mathbf{k} + v_4\mathbf{l}$

All we have to do now is apply it to columns of the 4×4 identity matrix. This results in the following translation transformation matrix:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You can translate a vector **v** by computing **Tv**.

9.2.4 The Rotation Matrices

Way back in Chapter Five, we derived the following formulas for the rotation of a point (*x*, *y*) around the origin of the coordinate system:

$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = y\cos(\theta) + x\sin(\theta)$$

where (*x*', *y*') is the rotated point. The above rotation is actually a rotation around the *z*-axis. So if we were rotating a 3D point (*x*, *y*, *z*), then the new point would be (*x*', *y*', *z*) (the *z* coordinate does not change for rotation around the *z* axis).

It is helpful if you make your rotations obey the so-called right hand rule. This rule tells you to point your right hand's thumb in the positive direction of the axis you want to rotate around. Your fingers will curl in the direction that corresponds to positive rotation angles. Negative rotation angles specify rotations in the opposite direction.

As it stands now, the above equations do not obey the right hand rule (they are the exact opposite), but that is easy enough to fix. All we have to do is negate the angle, and use the properties of trig functions described in Chapter Five. The end result is shown below:

$$x' = x\cos(\theta) + y\sin(\theta)$$
$$y' = y\cos(\theta) - x\sin(\theta)$$

These equations lead to the following linear transformation:

$$\mathbf{R_z}(\mathbf{v}) = (v_1\cos(\theta) + v_2\sin(\theta))\mathbf{i} + (v_2\cos(\theta) - v_1\sin(\theta))\mathbf{j} + v_3\mathbf{k} + v_4\mathbf{l}$$

Applying this transformation to the 4×4 identity matrix, we obtain the induced *z*-axis rotation matrix:

$$\mathbf{R_z} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotating a vector **v** around the *z*-axis is as easy as computing $\mathbf{R_z v}$.

What about the other axes of rotation? The process for finding the induced matrices is exactly the same; just find formulas for the rotated points and apply those to the columns of the 4×4 identity matrix.

We will not perform the calculations here, but rather just present you with the results:

$$\mathbf{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_y} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is worth noting that the inverse of any rotation-only matrix is equal to its own transpose. This can be easily proven for the above matrices, and is not too hard to prove for the more general case. (The critical step is showing that the rows/columns of all rotation-only matrices are orthonormal. After that, an argument based on the dot product easily finishes the proof.)

9.2.5 The Projection Matrix

Recall that *projection* (at least as far as our graphics discussions have gone in the course) is defined as the operation of projecting 3D geometry onto a 2D screen. This is an absolutely critical function for 3D games, since inevitably, all the data describing the three-dimensional game world must be displayed on a two-dimensional computer screen.

We calculated equations for projection in Chapter Four for a viewer centered at the origin, looking down the positive *z*-axis. The equations were:

$x' = xd/z$
y' = yd/z

where *d* is the "distance" from the viewer to the plane the geometry is being projected onto (the computer screen). Notice this projects points of the form $\langle x, y, z \rangle$ to $\langle x', y', d \rangle$ (where only the first two components are really necessary). We can therefore define a transformation function as follows:

$T(\mathbf{v}) = v_1 d/v_3\mathbf{i} + v_2 d/v_3\mathbf{j} + d\mathbf{k}$

The division in this equation should set off an alarm for you. Linear equations do not involve divisions (by variables, anyway), and so you might suspect that the projection transformation is not linear. Let us check the first of the linearity properties to be sure:

$$\begin{aligned} T(c\mathbf{v}) &= (cv_1)d/(cv_3)\mathbf{i} + (cv_2)d/(cv_3)\mathbf{j} + d\mathbf{k} \\ &= v_1 d/v_3\mathbf{i} + v_2 d/v_3\mathbf{j} + d\mathbf{k} \end{aligned}$$

$$\begin{aligned} cT(\mathbf{v}) &= c(v_1 d/v_3\mathbf{i} + v_2 d/v_3\mathbf{j} + d\mathbf{k}) \\ &= cv_1 d/v_3\mathbf{i} + cv_2 d/v_3\mathbf{j} + cd\mathbf{k} \end{aligned}$$

As we can see, $T(c\mathbf{v})$ is not equal to $cT(\mathbf{v})$, so the projection transformation is not linear. This means, for one, that it has no induced matrix. What is needed then is another trick, like the one we used for translation. Recall that the homogenous point $\langle x, y, z, w \rangle$ represents the 3D point $\langle x/w, y/w, z/w \rangle$. If *w* "just happened" to be equal to *z*/*d*, then the 3D point would be equal to $\langle xd/z, yd/z, d \rangle$. This suggests that the following transformation function might work for projection:

$T(\mathbf{v}) = v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k} + v_3/d\mathbf{l}$

There is a division in this transformation, but it is only by a constant, not by a variable, so there is hope that the transformation is now linear. Let us check the two properties:

$T(c\mathbf{v}) = cv_1\mathbf{i} + cv_2\mathbf{j} + cv_3\mathbf{k} + cv_3/d\mathbf{l}$

$$\begin{aligned} cT(\mathbf{v}) &= c(v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k} + v_3/d\,\mathbf{l}) \\ &= cv_1\mathbf{i} + cv_2\mathbf{j} + cv_3\mathbf{k} + cv_3/d\mathbf{l} \end{aligned}$$

The first property is satisfied. One down and one to go:

$T(\mathbf{v} + \mathbf{u}) = (v_1 + u_1)\mathbf{i} + (v_2 + u_2)\mathbf{j} + (v_3 + u_3)\mathbf{k} + (v_3 + u_3)/d\mathbf{l}$

$$\begin{aligned} T(\mathbf{u}) + T(\mathbf{v}) &= (v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k} + v_3/d\mathbf{l}) + (u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k} + u_3/d\mathbf{l}) \\ &= (v_1 + u_1)\mathbf{i} + (v_2 + u_2)\mathbf{j} + (v_3 + u_3)\mathbf{k} + (v_3 + u_3)/d\mathbf{l} \end{aligned}$$

Thus the second property is satisfied!

Before we compute the induced matrix, it is worth pointing out what we have actually done here. The projection transformation modifies the *w* component of the homogenous vector so that it is no longer 1. In order to actually transform the point into the form <*x*', *y*', *d*>, we would still have to homogenize the result (i.e. divide each component by *w*). So in a sense, we really have not "projected" until we have homogenized the point.

Also, note that translation requires $w = 1$, so you cannot translate points after you project them unless you first homogenize (not that doing so would make much sense, anyway).

Now let us compute the induced matrix, applying the transformation function to the columns of the 4×4 identity matrix:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

By inspection, you can see the rows of this matrix are not linearly independent. This means that the projection matrix does not have an inverse. This is as it should be. Each projected point on a computer screen could be any one of an infinite number of 3D points. There is just no way to tell which one is which.

You are probably wondering exactly why we have gone to the trouble of expressing all of these different transformations as matrices, rather than just use the transformation functions directly on the points. The answer to that question is the topic of the next section.

9.3 Matrix Transformations

Nearly all computer games and most graphics tools use matrices for geometry transformation. Since matrices are not as intuitive as functions, there has to be something really attractive about them to warrant all of the attention they have received.

In fact, there is. One benefit is pretty obvious. You can just create a library of matrices, and no matter what (linear) operation you need to do to a vector, you can do it by multiplying the vector by the appropriate matrix. Matrices standardize all transformations to matrix multiplication.

Another benefit is that most new video cards on the market can perform matrix operations *in hardware* -- much faster than the CPU can do it. Since you can encode all linear transformations with matrices, this means that you can use the video card to perform all of your linear transformations (the Graphics Programming course series goes into more depth on exactly how you do this). Without matrix math, you would be stuck doing everything in software or perhaps taking advantage of a few functions the manufacturer chose to encode in hardware.

Perhaps the largest benefit to matrix math, however, comes into play when you need to perform a sequence of transformations on a number of points. For example, if you are animating a swinging door, then you need to translate the door so that its hinges are at the origin, then rotate it by the appropriate amount, then translate it back to its position, and finally, project it onto the computer screen. (Actually, you have to do even more operations than this, but we will cover why later.) If you were using functions to do the transformations, then you would have to send each point to the first transformation function, then to the second, then to the third, and so on, in sequence. *Matrix math allows you to perform all of the transformations at once*.

To see how, suppose we want to apply *n* transformations to a point represented by the vector **v** (each transformation can be any linear transformation at all). Further, suppose the induced matrices are denoted $\mathbf{T_1}$, $\mathbf{T_2}$, …, $\mathbf{T_n}$. To transform the point by the first transformation, we compute $\mathbf{T_1 v}$. Then we multiply that by the second transformation matrix, which gives us $\mathbf{T_2}(\mathbf{T_1 v})$. In the end, after all of the multiplications have been performed, we get the transformed vector $(\mathbf{T_n}(\mathbf{T_{n-1}}...(\mathbf{T_2}(\mathbf{T_1 v}))...))$, or $\mathbf{T_n T_{n-1}}...\mathbf{T_2 T_1 v}$.

Here is the critical part - since matrix multiplication is associative, we can throw in parentheses wherever we want. In particular, *we can add parentheses around the transformation matrices*. The transformed point then becomes $(\mathbf{T_n T_{n-1}}...\mathbf{T_2 T_1})\mathbf{v}$. Notice that $(\mathbf{T_n T_{n-1}}...\mathbf{T_2 T_1})$ is a matrix, and that multiplying it by **v** produces a vector transformed by *all of* the transformation matrices.

So now we can merge any number of linear transformation matrices into a *single* transformation matrix that does the job of all of them. Going back to the door example, we could compute one matrix that does all the operations we want it to, and then multiply this matrix by each of the door's points (i.e. the vertices of the polygons that comprise the 3D door model). This results in a dramatic performance gain.

Multiplying the matrices together is often called *concatenation*, *compounding*, *catenation*, or *composition*. Whatever you call it, the power of the operation is perhaps the most compelling argument for using matrices in your games.

There is one last important point that needs to be remembered: matrix multiplication is not commutative. So $(\mathbf{T_n T_{n-1}}...\mathbf{T_2 T_1})$ is, in general, not the same as $(\mathbf{T_1 T_2}...\mathbf{T_{n-1} T_n})$. The first one is the matrix $\mathbf{T_n}$ multiplied by the matrix $\mathbf{T_{n-1}}$, and so on. The second one is the matrix $\mathbf{T_1}$ multiplied by the matrix $\mathbf{T_2}$, and so on. If you want to apply the transformation $\mathbf{T_1}$ to the vector first, followed by $\mathbf{T_2}$, then $\mathbf{T_3}$, and so on, with $\mathbf{T_n}$ being the last transformation applied, then you must multiply the matrices together in the first order.

In the next section, we will discuss how to use what we have learned to write a 3D game.

9.4 Linear Transformations in 3D Games

Today's three-dimensional games usually represent objects as collections of polygons. Points define the vertices of these polygons. The points themselves can even be animated to animate the object.

You can easily animate points using the theory introduced in this chapter. Just use the appropriate matrices to transform the points according to the requirements of the animation. (Usually artists design the animations in 3D graphics programs, and these programs typically output the animation as a series of scale, rotation, and translation matrices – often called SRT matrices for short.)

A larger question is how to view all of our three-dimensional geometry from an arbitrary point-of-view. Players want to *move around* in 3D games, exploring the game world, killing bad guys and solving puzzles. This means that your games have to be able to display the three-dimensional game world as viewed through the eyes of the player, who can be at any location, looking in any direction.

In Chapter Four, we learned how to project 3D geometry onto a 2D screen for a viewer situated at the origin of the coordinate system, looking down the positive *z*-axis. The question is, what can we do when this is not the case?

Two solutions present themselves: (1) give the viewer an arbitrary position and orientation and then deduce new projection formulas from this information; (2) leave the viewer at the origin, looking down the positive *z*-axis, and move the game world around to simulate viewer movement.

Since the latter solution is the easiest, that is the one games use -- and the one we will cover here.

Motion is relative. If you are sitting in an airplane, looking out the window, you cannot tell (visually) whether the airplane is moving forward or the clouds are moving backwards. Similarly, when you walk, there is no visual difference between you moving forward and the world moving backward. So if we want to keep the viewer at the origin, then whenever the viewer wants to move in some direction, we can instead move the whole game world in the opposite direction. Similarly, if the viewer wants to rotate his or her head to look up, we can rotate the game world down.

This procedure can be formalized. The coordinate system where the game world exists is called *world space*. The viewer can have any position and orientation in this world space. To massage the geometry into a form we can easily project, we first change the underlying coordinate system by translating everything in the game world, including the viewer, by the *negative* of the viewer's position. This nicely places the viewer at the origin, but the viewer still has its own orientation -- one not necessarily looking down the positive *z*-axis. So we change the underlying coordinate system again, this time rotating everything in the game world, including the viewer, by the *opposite* of the viewer's orientation relative to the positive *z*-axis. This orients the viewer so that he is looking down the positive *z*-axis.

The final coordinate system created in this process is sometimes called *view space* or *camera space*. Once everything has been transformed into this space, projection is easy, and can be done with the matrix introduced in this chapter or the equations derived in Chapter Four.

Now let us get specific. Suppose the viewer has position **v**, and its orientation is described by the three rotation angles r_x, r_y, and r_z, which indicate the extent of the viewer's rotation along the *x*-, *y*-, and *z*-axes, respectively. Further, suppose we choose $r_x = r_y = r_z = 0$ to indicate the direction of the positive *z*-axis. Then, to transform the game world from world space into view space, we must perform the following two tasks:

1. Translate everything in the game world by **-v**.
2. Rotate everything in the game world by $-r_x$, $-r_y$, and $-r_z$, along the *x*-, *y*-, and *z*-axes, respectively.

Lastly of course, we would want to project the geometry onto the screen.

Using the linear transformation matrices defined earlier, the matrix that pulls all this magic off is going to look something like this:

$$\mathbf{P} \times \mathbf{S} \times \mathbf{R}_\mathbf{z} \times \mathbf{R}_\mathbf{y} \times \mathbf{R}_\mathbf{x} \times \mathbf{T}$$

The homogenous points transformed with this matrix have to be homogenized, and then they can be displayed on the computer screen.

Notice that we slipped the scaling matrix **S** into the above transformation. This allows you to control the final scale of the projected points (i.e. how big or small objects appear on the screen).

Also note that the order of rotation is *x*-axis rotation, followed by *y*-axis rotation, followed by *z*-axis rotation. The order is significant, as it gives meaning to the rotation angles that describe the viewer's orientation. The particular order we have chosen means that the viewer's rotation should be such that if the viewer is rotated first by $-r_x$ radians around the *x*-axis, followed by $-r_y$ radians around the *y*-axis, and then by $-r_z$ radians around the *z*-axis, the viewer should be facing the direction of the positive *z*-axis. It is not always obvious how the rotation angles should be chosen for a given viewer orientation, but this problem is solved nicely by the use of quaternions (the topic of our next lesson).

The above transformation matrix (or one very much like it) is at the heart of all computer games. It gives players the freedom to roam the game world at their pleasure. Knowing what it looks like and how it works, you could very easily create a basic 3D game right now (although these days you would probably end up using a 3D API like Direct3D for your games, both to take advantage of hardware acceleration and to simplify game development).

In the next section, we will discuss rotation matrices in a bit more depth, and end up developing an alternate method of transforming from world space to view space -- one that has some advantages to the composition of separate *x*, *y*, and *z* rotation matrices.

9.5 Another Look at Rotation

The rotation phase of the world-to-view transformation comes *after* the translation phase. That is, points are first translated by the negative of the viewer's position, and then rotated by the opposite of the viewer's orientation.

Suppose we say that, after points have been translated, they exist in *translated world space* (TW space for short). In this space, the viewer is situated at the origin, but can have any orientation.

Instead of using angles, we can describe the orientation of the viewer with three vectors. The first is the *direction* or *look-at* vector, which describes the direction the viewer is looking. The second is the *up vector*, which, if the viewer were wearing a cone-shaped party hat, would describe the direction the tip of the hat was pointing. (The up vector allows the viewer to tilt his or her head side to side. With only the direction vector, this would be impossible.) The third vector is the *right vector* and describes the direction the viewer's right side is facing (this is simply the cross product of the direction and up vectors, in that order). This method of describing the viewer's orientation is shown in Figure 9.2.

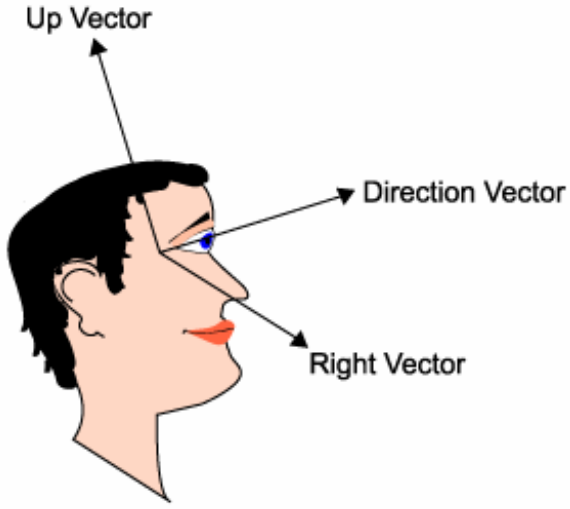


Figure 9.2: Describing viewer orientation with the direction, up, and right vectors.

The rotation phase is a transformation from TW space to view space. In this transformation, the viewer's direction vector gets aligned with the positive *z*-axis, the up vector gets aligned with the positive *y*-axis, and the right vector gets aligned with the positive *x*-axis.

The critical thing to notice is this: the viewer's direction, up, and right vectors can be used as the basis vectors for view space. That is, we can represent all points in view space as linear combinations of these three vectors. More formally, if we call these vectors **d**, **u**, and **r**, then we can write any vector **v** in view space in the following form:

$$\mathbf{v} = v_1\mathbf{r} + v_2\mathbf{u} + v_3\mathbf{d}$$

In view space (i.e. after the rotation transformation), the vectors **r**, **u**, and **d** are aligned with the standard basis vectors **i**, **j**, and **k**, respectively. In TW space, however, the vectors are not aligned with anything; they just encode the orientation of the viewer before the rotation transformation.

We need to relate vectors in view space to vectors in TW space (since this will provide a means of transforming vectors from one space to the other). The obvious way to do this is to relate the basis vectors in view space to the basis vectors in TW space. To do this, we are going to need an orthonormal basis for TW space. Suppose we call the basis vectors $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$. Then any vector **u** can be written as follows:

$$\mathbf{u} = u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}$$

where u_1, u_2, and u_3 are the coordinates of the vector in TW space.

Since the above result is true for any vector, in particular, we can represent our three basis vectors for view space as a linear combination of $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_3}$. Suppose these vectors have the following representations in TW space:

$$\mathbf{r} = \langle r_1, r_2, r_3\rangle = r_1\mathbf{e_1} + r_2\mathbf{e_2} + r_3\mathbf{e_3}$$
$$\mathbf{u} = \langle u_1, u_2, u_3\rangle = u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}$$
$$\mathbf{d} = \langle d_1, d_2, d_3\rangle = d_1\mathbf{e_1} + d_2\mathbf{e_2} + d_3\mathbf{e_3}$$

Then plugging these equations into our expression for a vector **v** in view space, we get the following results:

$$\begin{aligned}\mathbf{v} &= v_1(r_1\mathbf{e_1} + r_2\mathbf{e_2} + r_3\mathbf{e_3}) + v_2(u_1\mathbf{e_1} + u_2\mathbf{e_2} + u_3\mathbf{e_3}) + v_3(d_1\mathbf{e_1} + d_2\mathbf{e_2} + d_3\mathbf{e_3})\\ &= v_1r_1\mathbf{e_1} + v_1r_2\mathbf{e_2} + v_1r_3\mathbf{e_3} + v_2u_1\mathbf{e_1} + v_2u_2\mathbf{e_2} + v_2u_3\mathbf{e_3} + v_3d_1\mathbf{e_1} + v_3d_2\mathbf{e_2} + v_3d_3\mathbf{e_3}\\ &= v_1r_1\mathbf{e_1} + v_1r_2\mathbf{e_2} + v_1r_3\mathbf{e_3} + v_2u_1\mathbf{e_1} + v_2u_2\mathbf{e_2} + v_2u_3\mathbf{e_3} + v_3d_1\mathbf{e_1} + v_3d_2\mathbf{e_2} + v_3d_3\mathbf{e_3}\\ &= (v_1r_1 + v_2u_1 + v_3d_1)\mathbf{e_1} + (v_1r_2 + v_2u_2 + v_3d_2)\mathbf{e_2} + (v_1r_3 + v_2u_3 + v_3d_3)\mathbf{e_3}\end{aligned}$$

This is actually very cool. We have just expressed a vector **v** in view space as a vector in TW space. So the above equations transform a vector from view space into TW space.

That is not exactly the result we need, though. What we are really after is a way to transform a vector from TW space into view space, since that is what is required by the world-to-view transformation. But we can still use the above results. All we need to do is compute the induced matrix for the above transformation, and then invert it (which we can do by taking its transpose -- recall that the inverse of any rotation-only matrix is equal to the transpose of that matrix). Since the induced matrix transforms from view space into TW space, the inverse matrix will transform from TW space into view space. We can find the induced matrix by applying the transformation to the columns of the 3×3 identity matrix. The result is shown below:

$$\begin{bmatrix} r_1 & u_1 & d_1 \\ r_2 & u_2 & d_2 \\ r_3 & u_3 & d_3 \end{bmatrix}$$

The transpose of this matrix -- which is the transformation we really seek -- is shown below:

$$\begin{bmatrix} r_1 & r_2 & r_3 \\ u_1 & u_2 & u_3 \\ d_1 & d_2 & d_3 \end{bmatrix}$$

The equivalent 4×4 transformation matrix (for compatibility with all of the other matrices introduced in this lesson) is as follows:

$$\begin{bmatrix} r_1 & r_2 & r_3 & 0 \\ u_1 & u_2 & u_3 & 0 \\ d_1 & d_2 & d_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now let us recap our discovery. If you define the viewer's orientation by the right, up, and direction vectors, *all defined in TW space*, then the above matrix will transform vectors from TW space to view space. (Note that the translation part of the world-to-view transformation does not affect the viewer's orientation, so you can think of the right, up, and direction vectors as being defined in *world space*.)

This result is helpful in several ways. First, it frees you from having to keep track of angles. If you want, you can describe the orientation of the viewer entirely by vectors (rotating these vectors whenever the viewer changes orientation), and use these vectors to directly generate the rotation transformation matrix.

Second, even if you still want to use angles, you can use the above results to extract the right, up, and direction vectors from *any* rotation matrix -- even one created by concatenating separate *x*, *y*, and *z* rotation matrices. This is useful, for example, if you are using angles to generate the rotation matrix but for some reason you need a vector describing the direction of the viewer (this is found in the third row of the rotation matrix).

If you represent the viewer's position with the vector **v**, then the complete world-to-view transformation matrix can be found by multiplying the above rotation matrix with a translation matrix that translates points by the negative of **v**. This matrix is shown below:

$$\begin{bmatrix} r_1 & r_2 & r_3 & -v \cdot r \\ u_1 & u_2 & u_3 & -v \cdot u \\ d_1 & d_2 & d_3 & -v \cdot d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is a very clean result (using angles to represent viewer orientation produces a matrix wider than the width of this page!), and this is, perhaps, yet another reason why game developers favor this approach.

This lesson's material is nearly at an end. We have seen how to compute the induced matrix for any linear transformation, we have looked at common transformation matrices, and we have developed methods to display a 3D world given a viewer position and orientation (quite a feat for a single chapter).

In the next and final section, we will explore a slight difference between the matrices presented in this course and those you might find elsewhere -- a difference that has been the source of many a headache for the aspiring game developer.

9.6 Row versus Column Vectors

Throughout this chapter, we have been using single-column matrices to represent vectors. Due to the rules of matrix multiplication, this means that the only way we can multiply a matrix by a vector and get a vector is if the order of multiplication is *matrix-times-vector*.

However, we can also use single-row matrices to represent vectors. In this case, the only way we can multiply a matrix by a vector and get a vector is if the order is *vector-times-matrix*.

Most math texts, and a good deal of computer science texts, use single-column matrices to represent vectors. However, many sources do not, especially in the world of game development (for a reason we will discuss later, most game developers prefer using single-row matrices to represent vectors).

The transformation matrices used for column vectors will not, in general, be the same as the transformation matrices for row vectors. To see the relationship between them, suppose we have a column vector **v** that is equal to the product of multiple transformation matrices times some other vector **u**. If the transformation matrices are labeled $\mathbf{T}_1$, $\mathbf{T}_2$, ..., $\mathbf{T}_{n-1}$, $\mathbf{T}_n$, we can express this as follows:

$$\mathbf{v} = \mathbf{T}_n\mathbf{T}_{n-1}...\mathbf{T}_2\mathbf{T}_1\mathbf{u}$$

To make see what happens when **v** is a single-row vector, we can transpose both sides of the equation (which will change both **v** and **u** into row vectors):

$$\mathbf{v}^T = (\mathbf{T}_n\mathbf{T}_{n-1}\ldots\mathbf{T}_2\mathbf{T}_1\mathbf{u})^T = \mathbf{u}^T\mathbf{T}_1^T\mathbf{T}_2^T\ldots\mathbf{T}_{n-1}^T\mathbf{T}_n^T$$

Two effects are apparent: using row vectors transposes the transformation matrices and changes the order of multiplication. So the moral of this story is that if you want to use row vectors instead of column vectors, then you will have to transpose all the matrices listed in this lesson. Also, you will have to change the order in which you would otherwise perform multiplication.

Why do game developers tend to prefer using row vectors? Because the order of multiplication reflects the order in which the transformations are applied to the vector. Using column vectors, the first matrix in the product is actually the last transformation applied to the vector. Using row vectors, it is the first.

Conclusion

In our next lesson, we are going to introduce several new classes of numbers. One of these classes, called *quaternions*, will enable us to compute the world-to-view transformation matrix without having to bother with angles *or* the up, right, and direction vectors. Quaternions will also give us a means to rotate points around any axis we want (as opposed to just the *x*-, *y*-, and *z*-axes).

Exercises

1. Prove that the transformation $T(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x}$ is not linear.

2. Determine if the transformation $T(\mathbf{x}) = \text{Proj}_{\mathbf{n}}(\mathbf{x})$ (vector projection for a fixed vector **n**) is linear. If it is, construct the induced matrix of the transformation.

3. Determine if 2D rotation around an arbitrary point (c_x, c_y) is a linear transformation. If it is, construct the induced matrix of the transformation.

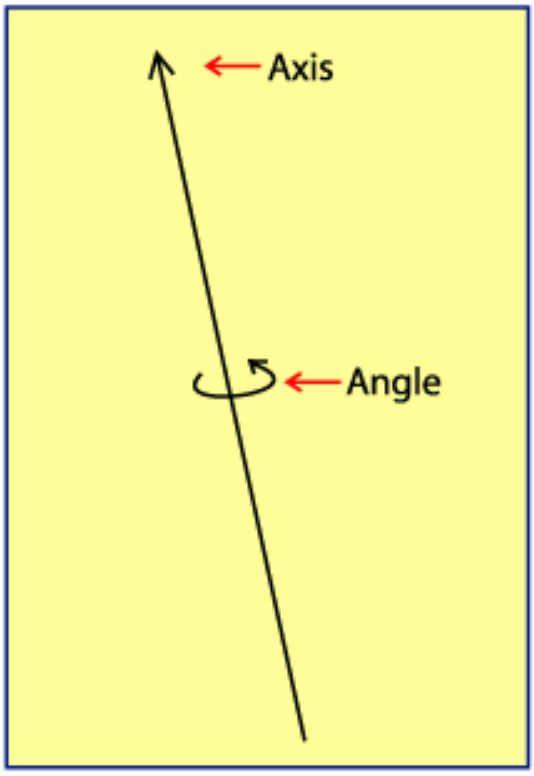
*4. Describe in detail how you could use matrices to animate a complex 3D shape (such as a character), using hierarchies to minimize redundant transformations.

*5. In the projection matrix introduced in this chapter, the factor of *d* encodes the field of view of the camera. This setup assumes the horizontal field of view is equal to the vertical field of view. Devise a projection matrix that allows for a vertical field of view that is different than the horizontal field of view. (Hint: Division/multiplication of individual vector components by constants does not change the linearity of a given transformation.)

!6. Find the induced matrix for a transformation that takes a 2D vector and reflects it off the *x*-axis.

Chapter Ten

Quaternion Mathematics



Introduction

When one of my younger brothers first started attending college, I persuaded him to take a course in pre-calculus, for the good of his soul (he was a business major). Math was not his favorite subject, and despite my hopes that pre-calculus would cure him of that (you learn to do so many neat things in pre-calc), he came out of that course with a newfound determination to bash mathematics at every opportunity. The reason? He stumbled onto the subject of *imaginary numbers*, and in his view, the fact that imaginary numbers were introduced to give meaning to square roots of negative numbers proved that mathematics is a hodge-podge; a collection of concepts so confused and inadequate that they require mathematicians in subsequent years to keep patching them up to prevent the whole of mathematics from crumbling. And I thought I was helping him by suggesting that he take that course!

This may all seem a bit confusing if you have not been introduced to imaginary numbers before. After all, we did just mention "the square root of a negative number". Prior to imaginary numbers, the domain of the square root function was strictly the set of non-negative numbers, such as 2 or 3.1415926. The reason for that is not too difficult to grasp; the square root of a negative number, such as -1, would have to be some number such that, when multiplied by itself, was equal to -1. But we learned way back in grade school that a negative number times a negative number is a positive number, and, of course, that a positive number times a positive number is also a positive number. What this means is that for any real number x, x^2 is always positive, and hence, there exists no real number such that, when multiplied by itself, is equal to -1 (or any other negative number, for that matter).

I can almost hear my brother screaming now, "Hole in math! Hole in math!" Is there really a hole in mathematics that we need to patch up? Not really. We have decided for various reasons that a negative real number times a negative real number is a positive real number. An implication of that decision is that there is no real number such that, when multiplied by itself, is equal to a negative number. You cannot have your cake and eat it too.

Now, we could always go back and by fiat decide that the product of two negative numbers is a negative number. But that would have some very unpleasant implications. Consider the distributive property of multiplication with respect to addition, which says that $a(b+c)=ab+ac$. Suppose we want to compute -5(6-4). Performing the addition inside the parentheses first, we see that this is -10. But if we use the distributive property with our new rule (a negative number times a negative number is negative), remembering that -5(-4) is really negative, then we get -30 + -20 = -50! As we can see, the assumption that the product of two negative numbers is a positive number is deeply ingrained in algebra, and any change to that assumption would radically alter all the rules that we have come to depend on.

So we cannot change the way things are defined already without losing the power of our existing definitions. One thing to ask is, "Why do we even need the square root of a negative number, anyway?"

Sometimes in mathematics, something is invented and then a use for it is found. But in the case of square roots of negative numbers, it seems the original motivation was categorizing the roots of polynomials. The roots (i.e. places where the polynomial evaluates to 0) of a quadratic, for example, are given by the quadratic formula introduced in Chapter Three. If the quadratic cannot possibly evaluate to zero (which is the case for x^2+1, for example), then the discriminant (the values inside the square root in

the quadratic formula) turns out to be a *negative number*. By counting these solutions too (even though there are no such real numbers), you can prove the so-called *Fundamental Theorem of Algebra*, which states that every polynomial of degree *n* has exactly *n* roots (some of them possibly duplicate or non-real). As its name suggests, this is an important theorem in algebra, one responsible for a number of other important mathematical results.

In any case, what you are really concerned with is whether or not square roots of negative numbers have any application to game development. The answer is yes! Such square roots come with an entirely new arithmetic, and have proven themselves fundamental to many real-world problems, such as quantum mechanics (where you cannot flip a page without encountering them), electricity and magnetism (yes, a knowledge of negative square roots really *did* go into building your computer), and much more. In our case, it will turn out that these strange square roots can be used to represent rotations around arbitrary axes in a slick and efficient manner that lends itself to the needs of game development.

This is all going to be made possible through the concept of *imaginary numbers*.

10.1 Imaginary Numbers

If you want to get accused of being a lame know-it-all (and who does not enjoy that?), here is a little trick you can play on some poor hapless soul: draw a number, say 5, on a card, and hold the card out to the person and ask, "What is this?" They will doubtless respond, "The number 5." Then tear up the card and exclaim in your most mocking voice, "Oh, no! What are we to do now that I have destroyed the number 5!"

Of course, by tearing up the card you really did not destroy the number five, because that symbol on the paper *was not, in fact, the number 5*. Rather, it *represented* the number five. So what, you ask, *is* the number 5? The number 5, just like all other numbers, is a concept, and it exists only in peoples' heads.

So in that sense, all numbers are imaginary. They are concepts, and although we can apply the concepts to things in the real world, which exist, the concepts themselves can never escape the confines of our minds.

The degree to which we can successfully apply the concept of numbers to the real world depends greatly on the context. We do not talk about negative quantities of cows in a herd, for example, because there is no obvious way to make sense of negative quantities of cows. On the other hand, we have no problem talking about negative amounts of money -- that is, deficits or debt -- as we throw around such phrases as "That car repair put me $4,500 in the hole." Similarly, there is no meaningful way to spell a word with six and a half letters, although you can easily measure the time to be six and a half hours past midnight.

What this means is that in some sense, mathematics exists on its own, and that as we are able, we apply its concepts to things in the real world in order to extract some benefit from doing so. Numbers and the whole of mathematics exist in an abstract realm -- a realm of the "imaginary".

Why do I philosophize on this point? I do so because we are about to encounter a new fundamental set of numbers; the set of *imaginary numbers*. Do not be misled by the name -- imaginary numbers are no more or less "imaginary" than real numbers are "real". Like real numbers, imaginary numbers are concepts, ones that exist in peoples' minds, and which happen to have their own set of properties. These properties, as we will see later on, turn out to give imaginary numbers an edge over real numbers in certain circumstances; that is, we can apply imaginary numbers to real world problems in places where we cannot apply real numbers.

An imaginary number is basically just the square root of a negative number. "But wait," you say, "I thought there is no real number such that, when multiplied by itself, is a negative number?" That is exactly correct; there is no such *real number*. But there is such an imaginary number. By definition, when you multiply an imaginary number by itself, the result is a *negative real number*. Specifically, the result is equal to whatever is inside the square root sign.

An example is in order. The number $\sqrt{-5}$ is an imaginary number. When you multiply this number by itself, you get a negative real number because of the way multiplication is defined for imaginary numbers. Notice how this result strongly contrasts with the real numbers, where if you multiply any number by itself, the result is positive. This is the *critical difference* between the two number systems.

There is a standard way to designate all imaginary numbers using the symbol *i*, which is defined to be $\sqrt{-1}$. The imaginary number $\sqrt{-5}$ can be written as $\sqrt{(5)(-1)} = \sqrt{5}\sqrt{-1} = \sqrt{5}i$. Thus, an alternate way of defining imaginary numbers is any number that can be expressed as the product of a non-zero real number and *i*, the fundamental imaginary number.

Imaginary numbers behave very much like real numbers, except for the peculiar property of imaginary numbers that says that the product of two of them is a negative real number. But nevertheless, some of the ways this rule manifests itself are not obvious, so in the next few sections we will examine the places where imaginary number arithmetic differs markedly from real number arithmetic.

10.1.1 Raising Imaginary Numbers to Powers

Suppose you want to raise an imaginary number in the form *ai* to some integer power, say *n*. This is written as $(ai)^n$, and expands as $a^n i^n$, just like in ordinary arithmetic. Now *a* is just a real number, so you can compute a^n just like you would in real number arithmetic. The interesting question is, what is the value of i^n?

Suppose $n = 2$. Then our number becomes $a^n i^2 = a^n(ii)$. What is the quantity *ii*? We can expand this as $\sqrt{-1}\sqrt{-1}$, which, by the definition of imaginary numbers, is equal to -1. Thus $a^n i^2 = -a^n$. As you can see, this process ultimately depends on what power of *n* we raise *i* to.

Table 10.1 shows the results for powers of *i* from 0 to 5. The pattern you see there continues.

n	i^n
0	1
1	*i*
2	-1
3	-i
4	1
5	*i*

Table 10.3: The powers of i.

You may also be wondering what happens if you raise *i* to fractional powers, like 0.1 or 5.92. It turns out the result is not an imaginary number at all, but a complex number (which we will introduce later). Game developers, however, are generally only interested in integer powers of *i*.

10.1.2 Multiplying/Dividing Imaginary Numbers

Suppose you have two imaginary numbers, $x = ai$ and $y = bi$. Then their product is simply:

$$xy = aibi = abi^2 = -ab$$

If you were to divide *x* by *y*, then you get just *x*/*y* (*i*/*i* is just 1).

From this discussion, you can see that both the product and the division of any two imaginary numbers is a real number.

10.1.3 Adding/Subtracting Imaginary Numbers

To add or subtract two imaginary numbers $x = ai$ and $y = bi$, just group like terms:

$$x + y = ai + bi = (a + b)i.$$

Subtraction is performed in like manner, so that $ai - bi = (a - b)i$. If $a = b$, then the result is a real number (namely zero), but otherwise, the result is imaginary.

10.2 Complex Numbers

Our tale of strange mathematical numbers does not end with imaginary numbers, since if it did, there would be little benefit to covering them since imaginary numbers in themselves have no direct application to game development.

Rather, it is with imaginary numbers that our story *begins*. Imaginary numbers are necessary components of complex and hypercomplex numbers -- and it is only after we have covered the latter topic that we start getting into results that game developers can use.

A complex number is the sum of a real number and an imaginary number. Complex numbers are written in the form $a + bi$, where both a and b are real numbers. Both real numbers and imaginary numbers are subsets of complex numbers, as you can see if you let $b = 0$ and $a = 0$, respectively. The peculiar trait of complex number arithmetic is also that peculiar trait of imaginary number arithmetic; the product of two imaginary numbers is a negative real number.

A complex number $a + bi$ can be viewed as the ordered point (a, b) on a Cartesian coordinate system (see Figure 10.1). This geometric interpretation is quite important.

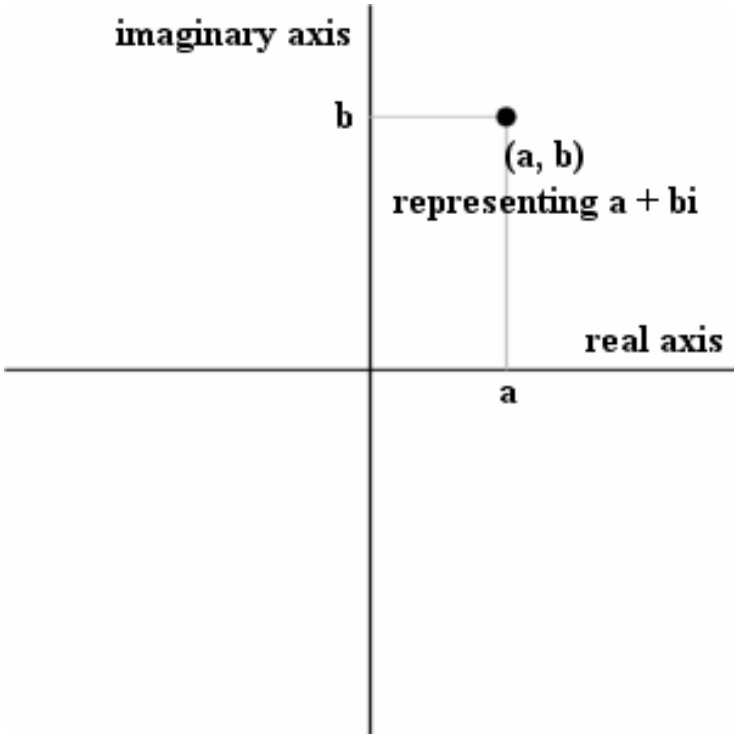


Figure 10.1: Representing points as complex numbers.

In the next few sections, we will discuss complex number arithmetic.

10.2.1 Adding/Subtracting Complex Numbers

Suppose you have two complex numbers, $x = a + bi$, $y = c + di$, and you want to compute $x + y$. All you have to do is add the real and imaginary components separately. Thus:

$x + y = (a + bi) + (c + di) = (a + c) + (b + d)i.$

Subtraction is performed in like manner:

$x - y = (a + bi) - (c + di) = (a - c) + (b - d)i.$

Geometrically, addition or subtraction of complex numbers corresponds to a translation.

10.2.2 Multiplying/Dividing Complex Numbers

To multiply two complex numbers together, you use the distributive property, remembering that the product of imaginary numbers is a negative real number.

If $x = a + bi$, $y = c + di$, then:

$$\begin{aligned} xy &= (a + bi)(c + di) \\ &= ac + adi + bci + bdi^2 \\ &= ac + (ad + bc)i - bd \\ &= (ac - bd) + (ad + bc)i. \end{aligned}$$

Thus, we have the following result:

$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$

Division is a bit trickier. Suppose we want to compute $(a + bi)/(c + di)$. The first step (which will seem to come out of nowhere, but will actually make more sense when we introduce complex conjugates) is to multiply both the top and bottom of the fraction by $(c - di)$, which you can do without changing the value of the fraction. This leads to the following result:

$$\frac{(a+bi)}{(c+bi)} = \frac{(a+bi)(c-bi)}{(c+bi)(c-bi)} = \frac{(ac+bd)+(bc-ad)i}{c^2+cbi-cbi-bi^2} =$$

$$\frac{(ac+bd)+(bc-ad)i}{c^2+b^2} = \left(\frac{ac+bd}{c^2+b^2}\right)+\left(\frac{bc-ad}{c^2+b^2}\right)i$$

Notice that the result is a complex number. Also note that if $c = b$, then the division is undefined (since it leads to a division by zero).

You can also multiply or divide a complex number by a real number. To do this, just multiply or divide each term of the complex number by that real number.

10.2.3 Raising Complex Numbers to Powers

To compute $(a + bi)^n$, all you have to do is multiply the quantity $(a + bi)$ by itself n times. Here are a few examples:

$(a + bi)^2 = a^2 + 2iab - b^2$
$(a + bi)^3 = a^3 + 3ia^2b - 3ab^2 - ib^3$
$(a + bi)^4 = a^4 + 4ia^3b - 6a^2b^2 - 4iab^3 + b^4$

10.2.4 The Complex Conjugate

The complex conjugate of a complex number (or even an entire expression involving complex numbers, for that matter) is what you get when you replace all occurrences of 'i' with '$-i$'. The complex conjugate of $x = a + bi$, for example, is $a - bi$. This is denoted x^*.

An interesting thing happens when you multiply a complex number by its complex conjugate -- the imaginary part disappears. For example, $(a + bi)(a + bi) = a^2 + b^2$. (When computing the division of complex numbers, in order to get rid of the imaginary part of the denominator, we multiplied both the top and bottom of the fraction $(a + bi)/(c + di)$ by the complex conjugate of $(c + di)$.)

10.2.5 The Magnitude of a Complex Number

The magnitude of a complex number $x = (a + bi)$, denoted $|x|$, is defined as follows:

$$| x |= \sqrt{x^* x} = \sqrt{(a - bi)(a + bi)} = \sqrt{a^2 + b^2}$$

Geometrically, you can interpret this as the distance from the origin to the point (a, b).

10.3 Introduction to Quaternions

On October 16th, 1843, William Rowan Hamilton was strolling with his wife along the Royal Canal in Ireland. At that moment, he was not thinking about his wife, the Royal Canal, or Ireland. Rather, he was thinking about a problem he had been working on, which involved the nature of hypercomplex numbers. Somewhere on his stroll, the solution to his problem came to him, and he was so excited with the result he carved it into the nearby Broome Bridge. What did he think was so important as to carve it into Broome Bridge? The formula he carved is shown below:

$i^2 = j^2 = k^2 = ijk = -1$

On the left, you see i^2, and on the right, -1. Look familiar? Yes, that is right -- i is indeed an imaginary number. In fact, i, j, and k are *all* imaginary numbers, but not exactly the kind of imaginary numbers that we have spent time studying. It is easiest to think of these imaginary numbers not as square roots of negative numbers, but as *symbols that obey Hamilton's rules*. (This may seem stranger than square roots of negative numbers, but it really is the easiest way to think about it.)

In order to understand the significance of Hamilton's rules, it is necessary to introduce the concept of hypercomplex numbers, the very thing Hamilton was studying when he had his revelation.

10.3.1 Hypercomplex Numbers and Quaternions

A hypercomplex number is the sum of one real number and several imaginary numbers. The hypercomplex numbers that are of interest to game developers are called *quaternions*. These are hypercomplex numbers of the following form:

$q_0 + q_1 i + q_2 j + q_3 k$

where i, j, and k are the imaginary numbers discussed in the last section, and the rest of the numbers are real numbers.

If $q_0 = 0$ in the above quaternion representation, then the quaternion is said to be a *pure* quaternion.

Hamilton's rules tell you what happens when you multiply the imaginary parts of hypercomplex numbers together. The quantity ijk, for example, is defined to be 1. Hamilton's rules also imply the following relations (which you should be able to derive easily):

$ij = k$
$jk = i$
$ki = j$
$ji = -k$
$kj = -i$
$ik = -j$

Generally speaking, as with complex and imaginary numbers, the arithmetic for quaternions is the same as the arithmetic for real numbers, except that you must use the above rules whenever imaginary components are multiplied together. In the few sections that follow, we will see how this is done.

10.3.2 Adding/Subtracting Quaternions

Adding two quaternions is exactly like adding two complex numbers. Simply add like terms.

For example, the addition of the quaternions $p = p_0 + p_1 i + p_2 j + p_3 k$ and $q = q_0 + q_1 i + q_2 j + q_3 k$ is shown below:

$$p + q = (p_0 + p_1 i + p_2 j + p_3 k) + (q_0 + q_1 i + q_2 j + q_3 k) = (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k$$

Subtraction is performed similarly:

$$p - q = (p_0 + p_1 i + p_2 j + p_3 k) - (q_0 + q_1 i + q_2 j + q_3 k) = (p_0 - q_0) + (p_1 - q_1)i + (p_2 - q_2)j + (p_3 - q_3)k$$

10.3.3 Multiplying Quaternions

Multiplication of two quaternions is a bit trickier, since we have to remember all the rules for multiplying the imaginary numbers (although we can still expand the product in the usual fashion).

If $p = p_0 + p_1 i + p_2 j + p_3 k$ and $q = q_0 + q_1 i + q_2 j + q_3 k$, then we can compute their product as follows:

$$pq = (p_0 + p_1 i + p_2 j + p_3 k)(q_0 + q_1 i + q_2 j + q_3 k)$$

$$\begin{aligned}
= \; & (p_0)q_0 + (p_0)q_1 i + (p_0)q_2 j + (p_0)q_3 k + \\
& (p_1 i)q_0 + (p_1 i)q_1 i + (p_1 i)q_2 j + (p_1 i)q_3 k + \\
& (p_2 j)q_0 + (p_2 j)q_1 i + (p_2 j)q_2 j + (p_2 j)q_3 k + \\
& (p_3 k)q_0 + (p_3 k)q_1 i + (p_3 k)q_2 j + (p_3 k)q_3 k
\end{aligned}$$

$$\begin{aligned}
= \; & (p_0 q_0) + (p_0 q_1)i + (p_0 q_2)j + (p_0 q_3)k + \\
& (p_1 q_0)i + (p_1 q_1)ii + (p_1 q_2)ij + (p_1 q_3)ik + \\
& (p_2 q_0)j + (p_2 q_1)ji + (p_2 q_2)jj + (p_2 q_3)jk + \\
& (p_3 q_0)k + (p_3 q_1)ki + (p_3 q_2)kj + (p_3 q_3)kk
\end{aligned}$$

$$\begin{aligned}
= \; & (p_0 q_0) + (p_0 q_1)i + (p_0 q_2)j + (p_0 q_3)k + \\
& (p_1 q_0)i + (p_1 q_1)(-1) + (p_1 q_2)k + (p_1 q_3)(-j) + \\
& (p_2 q_0)j + (p_2 q_1)(-k) + (p_2 q_2)(-1) + (p_2 q_3)(i) + \\
& (p_3 q_0)k + (p_3 q_1)(j) + (p_3 q_2)(-i) + (p_3 q_3)(-1)
\end{aligned}$$

$$
\begin{aligned}
= \quad & (p_0q_0) + (p_0q_1)i + (p_0q_2)j + (p_0q_3)k + \\
& (p_1q_0)i + (-p_1q_1) + (p_1q_2)k + (-p_1q_3)(j) + \\
& (p_2q_0)j + (-p_2q_1)(k) + (-p_2q_2) + (p_2q_3)(i) + \\
& (p_3q_0)k + (p_3q_1)(j) + (-p_3q_2)(\mathrm{i}) + (-p_3q_3)
\end{aligned}
$$

$$
\begin{aligned}
= \quad & (p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3) + \\
& (p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2)i + \\
& (p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1)j + \\
& (p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0)k
\end{aligned}
$$

From the above result, you can see that multiplication is not commutative (although it is associative and distributive with respect to quaternion addition). Despite this slight drawback, however, multiplication is one of the most important operations for quaternions, for reasons that will soon become clear.

10.3.4 The Complex Conjugate

The complex conjugate of a quaternion is very similar to the complex conjugate of a complex number. Just replace each imaginary number by its negative. So if $q = q_0 + q_1i + q_2j + q_3k$, then the complex conjugate of q, designated q^*, is $q_0 - q_1i - q_2j - q_3k$.

One important property of complex conjugation is that $(pq)^* = q^*p^*$. In words, the complex conjugate of the product of two quaternions is equal to the product of the individual complex conjugates, but in the opposite order.

The product q^*q, if you calculate the result, turns out to be $q_0^2 + q_1^2 + q_2^2 + q_3^2$. Note that the result is entirely real (which is not a coincidence, as Hamilton formulated his rules to produce this result). Also note that q commutes with q^* -- that is, $q^*q = qq^*$.

10.3.5 The Magnitude of a Quaternion

The magnitude of a quaternion q, denoted $|q|$, is defined as follows:

$$|q| = \sqrt{q^*q}$$

This also implies that $|q|^2 = q^*q$.

10.3.6 The Inverse of a Quaternion

The inverse of a quaternion is indeed a multiplicative inverse. Multiply it by the quaternion, and you get the real number 1. The inverse is unique.

For a quaternion q, the inverse is denoted q^{-1} and is given by the following equation:

$$q^{-1} = \frac{q^*}{|q|^2}$$

To prove this really is a multiplicative inverse, all we have to do is multiply it by q, and see if we get 1:

$$qq^{-1} = \frac{qq^*}{|q|^2} = \frac{|q|^2}{|q|^2} = 1$$

Notice that if the magnitude of q is already 1, then the inverse is equal to the complex conjugate of q.

10.4 Using Quaternions for Rotations

All the math you just learned about imaginary numbers is about to start paying dividends. Quaternions are not just odd looking mathematical constructs -- rather, hidden beneath the strange sum of a real number and three imaginary numbers is a very powerful and elegant way to rotate points around any axis. To uncover this important concept, let us take another look at the standard quaternion representation:

$q_0 + q_1 i + q_2 j + q_3 k$

It is possible to think of i, j, and k as "imaginary" vectors **i**, **j**, and **k** that obey Hamilton's rules. Under this interpretation, the components q_1, q_2 and q_3 become the components of a vector, say **q**, where $\mathbf{q} = q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$. Our quaternion then becomes:

$q_0 + \mathbf{q}$

If the quaternion is a pure quaternion, so that $q_0 = 0$, then we can think of the quaternion as representing a *vector* in three-dimensional space -- namely, the vector **q**.

Now suppose we have a vector represented by a pure quaternion v, and a *unit quaternion* (that is, a quaternion with magnitude 1) denoted by q. Then it is possible to show that the product qvq^* is the vector v rotated around an axis encoded in the quaternion q, by an angle also encoded in q (the product is a pure quaternion, and thus really can be interpreted as a vector).

This is a remarkable fact, but it is not sufficient. The question still remains -- exactly *how* does the quaternion q encode a rotation around an axis? Suppose q is expressed in the following form:

$$q = q_0 + \mathbf{q} = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)\mathbf{u}$$

where **u** is a unit vector. Then the product qvq* is the vector v rotated by an angle α around the axis **u**.

This is the exact result we seek, since it tells us how to construct a quaternion q in such a fashion that we can use it to rotate vectors around any axis we choose, by any angle we want (see Figure 10.2).

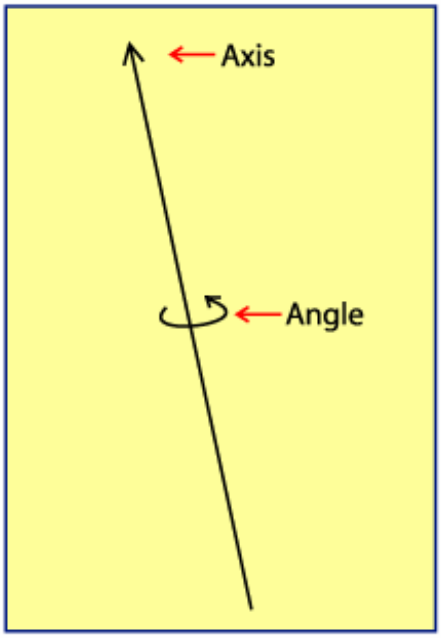


Figure 10.2: A quaternion rotation.

You can combine multiple quaternion rotations into a single quaternion. To show this, suppose we wanted to rotate the vector represented by the pure quaternion v by q, and then by p. The result of the first operation is qvq^*. Applying the second rotation, we get $p(qvq^*)p^*$. Since quaternion multiplication is associative, we can rewrite this as $(pq)v(q^*p^*)$. However, $(q^*p^*) = (pq)^*$, so we can write our rotated vector as $(pq)v(pq)^*$. If we let $r = pq$ (the quaternion product of p and q), then we see that the rotated vector is simply rvr^*. This is how you can combine quaternion rotations.

The next logical thing to do would be to develop a formula for qvq^*. But what is typically done instead is to calculate the matrix induced by qvq^* (the quaternion transformation is indeed linear), and then use this matrix to transform points. This is more efficient than transforming points directly, since you can combine the matrix with other matrices.

We will be skip the math derivation (since it is very ugly) and just look at the induced matrix of qvq^*:

$$\mathbf{Q} = \begin{bmatrix} 2q_0^2 - 1 + 2q_1^2 & 2q_1q_2 + 2q_0q_3 & 2q_1q_3 - 2q_0q_2 & 0 \\ 2q_1q_2 - 2q_0q_3 & 2q_0^2 - 1 + 2q_2^2 & 2q_2q_3 + 2q_0q_1 & 0 \\ 2q_1q_3 + 2q_0q_2 & 2q_2q_3 - 2q_0q_1 & 2q_0^2 - 1 + 2q_3^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

> **Note:** This particular matrix obeys the right-hand rule for determining the rotation direction. Point your thumb in the direction of the axis; positive angles correspond to the direction your fingers curl in, and negative angles correspond to the opposite direction.

Most of this chapter has been leading up to this result. With the above matrix, you can rotate points around any axis you want. Not only that, but you can combine any number of these rotations along with all the standard linear transformations (such as scaling and translation) into a single matrix.

What may not be clear right now is exactly how quaternions can benefit game developers. We will explore this in the next section.

10.4.1 Quaternions and the World-to-View Transformation

We noted in Chapter Nine that when you are creating the world-to-view transformation matrix, it is sometimes difficult to choose the *x*, *y*, and *z* rotation angles for the rotation matrices. You might think to set all these angles to zero, and then tie them to the keyboard (say) in the following way:

When the user presses the left or right arrow keys, increment or decrement the *y* rotation angle.
When the user presses the up or down arrow keys, increment or decrement the *x* rotation angle.
When the user presses some other two keys, increment or decrement the *z* rotation angle.

This simplistic scheme may not always give you the results you want. For example, suppose the user rotates 90 degrees around the *x*-axis, so he or she is facing the negative *y*-axis. Then any rotation performed on the *z*-axis will appear to the user as a rotation around the *y*-axis, and similarly, any rotation performed on the *y*-axis will appear to the user as a rotation around the *z*-axis. (See Figure 10.3.)

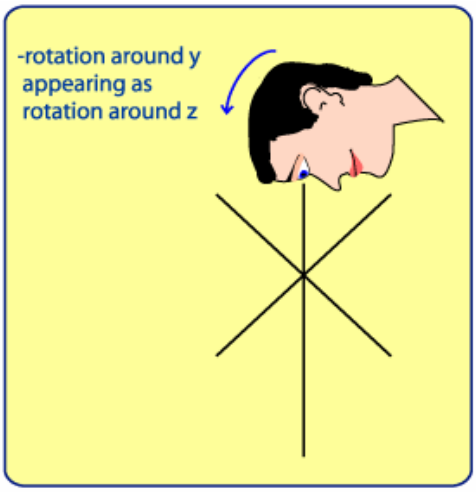


Figure 10.3: Problems using rotation angles.

This situation (called *gimbal lock*) occurs because each rotation is performed independently; first the *x*-axis rotation, then the *y*-axis rotation, and finally the *z*-axis rotation. Merging the rotation matrices into a single rotation matrix does not solve that problem. (Remember, the composite matrix does exactly the same thing as the three rotation matrices do separately when they are applied in sequence.)

You can fix this particular problem by changing the order of rotation, but no matter what order you ultimately select, there will still be many cases where rotation is not intuitive; where you should be rotating around one axis, you are actually rotating around another (or even several).

Not surprisingly (given the content of this chapter), quaternions are the solution to this problem. The reason is that quaternions perform rotation around a single axis, and by using a single angle, the problems associated with matrices simply cannot occur.

To create a quaternion-based rotation matrix for the world-to-view transformation matrix:

1. Create a quaternion that does not do anything to points. (It could rotate them around any old axis by 0 radians, for example.) Call this the *view quaternion*.

2. Whenever the user presses a key, generate a temporary quaternion for rotation around the axis corresponding to that key. The direction of rotation should be opposite to the direction the user is turning in. For example, if the user presses the *right* arrow key, generate a quaternion that rotates *leftward*, around the *y*-axis.

3. Multiply the view quaternion by the temporary quaternion. Use the result to update the view quaternion.

4. Create a rotation matrix that corresponds to the updated view quaternion, and use this matrix in the world-to-view transformation matrix.

This approach will work flawlessly, and yet still allow you to use matrices to transform your geometry. The only issue you have to be aware of is that after extended periods of time, the quaternion may cease having a magnitude of 1 (due to floating-point inaccuracies). To solve this problem, just renormalize it from time to time by dividing the components of the quaternion by its magnitude.

Note that under this scheme, there are no rotation angles, so you may wonder how you can figure out which direction the user is facing. This is easy. Either extract a direction vector from the rotation matrix, or use the inverse quaternion on the vector <0, 0, 1> (the positive *z*-axis direction). The latter approach works since the quaternion will transform the look-at vector to the vector <0, 0, 1>, so the inverse quaternion will transform the vector <0, 0, 1> to the look-at vector.

How do we know the inverse quaternion will undo the transformation of the quaternion? The proof is quite simple. Recall the form of the rotation quaternion:

$$q = q_0 + \mathbf{q} = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)\mathbf{u}$$

The quaternion that will undo this rotation has the same axis, but opposite angle. So plugging in $-\alpha$ into this equation, and calling the result *q*', we get the following:

$$q' = q_0^{'} + \mathbf{q}' = \cos\left(\frac{-\alpha}{2}\right) + \sin\left(\frac{-\alpha}{2}\right)\mathbf{u} = \cos\left(\frac{\alpha}{2}\right) - \sin\left(\frac{\alpha}{2}\right)\mathbf{u} = q_0^{'} - \mathbf{q}'$$

This is just the complex conjugate of *q*, by definition. Recall that for quaternions with magnitude 1 -- the only kind of quaternions that represent rotations -- the inverse is equal to the complex conjugate. So we have just shown that for a rotation quaternion *q*, the quaternion that will undo the rotation is equal to the inverse of *q*.

Quaternions have many other uses as well. For example, many high-end 3D graphics tools export animations using quaternions instead of rotation matrices. Further, it is possible to smoothly interpolate between two different quaternions, which is nice if you have a prerecorded series of orientations, stored as quaternions, and want to generate in between orientations. Other times you may just want to use quaternions to rotate points around arbitrary axes, for your own reasons.

Whatever kind of game you are interested in designing, chances are there are many places where you can incorporate quaternions instead of standard rotation matrices. And thanks to the elegance of quaternion rotation, doing just that will likely simplify your job considerably. Fortunately, you will not have to start from scratch. As part of the math library with this course, you will find a quaternion class that will make using quaternions a breeze.

Conclusion

In our next lesson, we tackle various problems in analytic geometry, using the all of the new math we have covered since Chapter Six.

Exercises

1. Construct the matrix for rotating points around the <1, 4, -1> axis.

2. What are the solutions of the polynomial equation $x^2 + 1 = 0$?

*3. You can use quaternions to represent the orientation of the camera in a 3rd person game like Tomb Raider™. One of the main advantages to doing so is that you can then smoothly change the camera orientation by interpolating the underlying quaternions. The interpolation algorithm commonly used is known as SLERP (**S**pherical **L**inear int**ERP**olation). Read about the technique at http://www.gamasutra.com/features/19980703/quaternions_01.htm and summarize your findings.

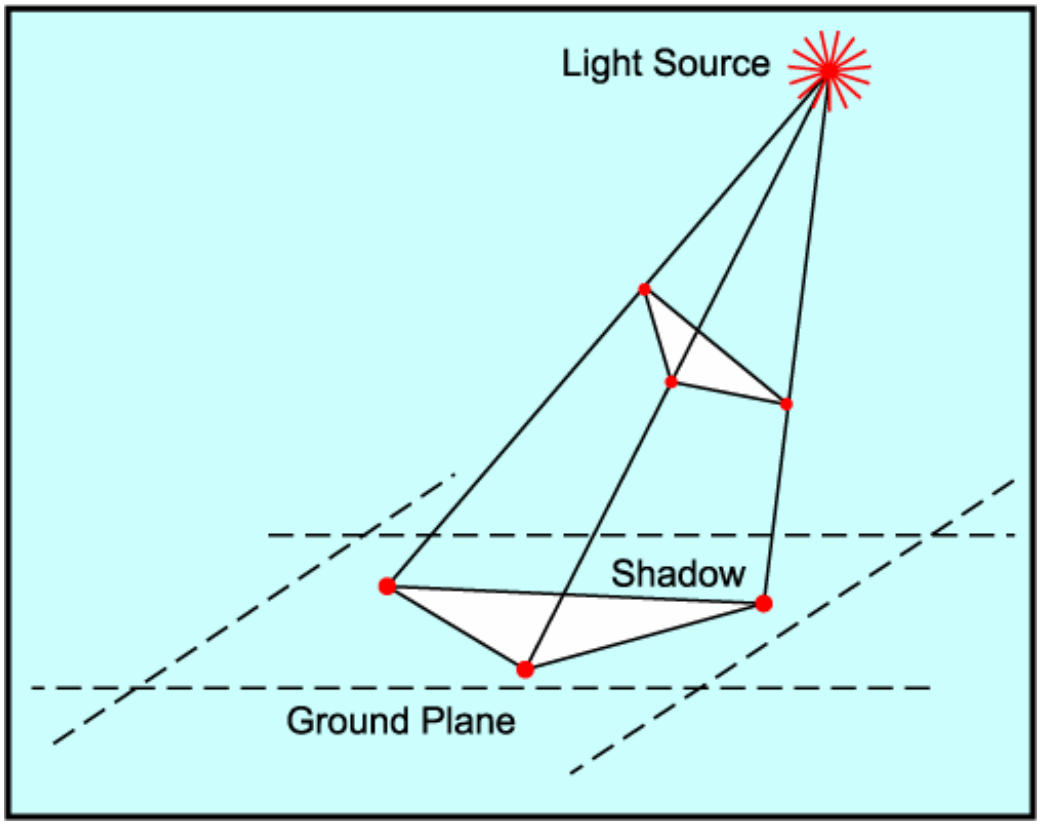
*4. What does multiplication of a complex number by *i* correspond to, geometrically?

!5. Describe how you can use the cross product operation and quaternion rotation to determine how a 3D vector **v** reflects off a plane with normal **n**. (This is not the actual method used for reflecting vectors, but it is instructive, nonetheless.)

*6. If you constructed a quaternion to rotate points around the positive *x*-axis, would the corresponding matrix (given by the quaternion-to-matrix formula shown in this lesson's material) be identical to the *x*-axis rotation matrix introduced in the last chapter? Why or why not? (Hint: Rotation matrices are invertible and all inverse matrices are unique.)

Chapter Eleven

Analytic Geometry II



Introduction

In Chapter Six, we saw some of the amazing things you can do when you give geometric shapes algebraic representations. Since that lesson, we have covered quite a bit of new material -- vectors, matrices, linear transformations, and quaternions. With that new material comes many new applications in the realm of analytic geometry. Some of these applications were introduced with the topics themselves, but many of them span multiple topics or could not be explained adequately with a half-page or so of material stuck on the end of a lesson. This chapter is dedicated to exploring those topics in a thorough and mathematically rigorous manner -- in a manner that will leave you fully prepared to pick up where this course leaves off, and take all the concepts and techniques you have learned and use them to solve your own problems.

This lesson is also going to be our last purely mathematical lesson. In our next lesson, we will come down from our ivory tower, venture into the real-world and start writing equations in C++ instead of math lingo -- creating, as we go, a 3D game, built using all of the math we have been talking about during this course.

11.1 Basic Collisions in 2D

Imagine that you are writing a pool (billiards) game. In this game you have a virtual cue that you use to strike balls in hopes of getting them into the six pockets lining the perimeter of the pool table. One of the physical facts that pool players take advantage of is that when the balls collide with the edges of the pool table or other balls, they bounce off at approximately the same angle they came in at. This observation, illustrated in Figure 11.1, opens up a world of possibilities for good pool players (as anyone who has watched a real-world competitive pool game knows).

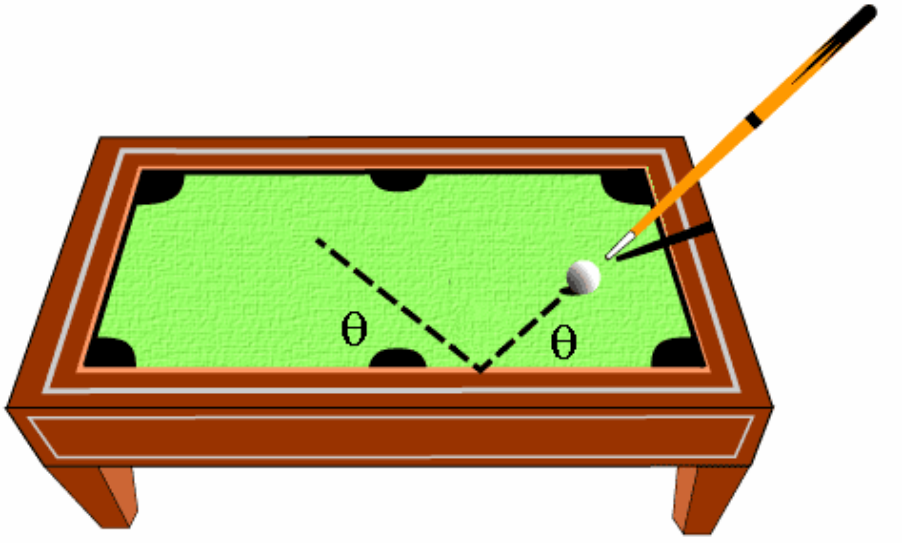


Figure 11.1: Recoiling in the game of pool.

As a game designer, you will face two obvious challenges in simulating the physics of the situation: (1) determining when one ball collides with another or with the edges of the pool table and (2) determining how balls should recoil when striking other objects or react when being struck.

When thinking about how to solve these problems, what you should not do is panic! You may not have a clear intuition about how to proceed, but if you approach the problems in a calm, systematic manner, you will gain a much better understanding about what is going on and what you have to do to solve the problems.

The first step in solving any problem is simple: *divide the problem into manageable tasks*. In the case of the pool game, we have two problems to solve, and each of those problems can be further divided. The choice of subdivisions is shown below:

- How to detect the collision of one ball with another ball.
- How to detect the collision of a ball with the edges of the pool table.
- How a ball should recoil when it hits another ball.
- How a ball should recoil when it hits the edges of the pool table.
- How a ball should react when it is hit.

The second step is to simplify each task as much as possible. For example, the pool table and the balls are three-dimensional objects, yet they really interact with each other only in two dimensions. Picture a plane drawn through the exact center of the balls. When balls collide with each other or with the edges of the pool table, the colliding parts will always be in this plane. What this suggests is that we can simplify the problems by reducing them to two dimensions.

The third step in solving the problem is simple. Draw a picture of what is happening in each task, and make sure the picture contains all of the information you know about the problem. Sometimes this is not possible, depending on the kind of problem you are trying to solve, but whenever it is possible, do it. Humans are visual creatures and generally have a much easier time understanding something when they can see it in front of them (hence all the figures and drawings in this course).

In our case, we have five individual problems to solve, and hence, at least five pictures to draw were we to solve all of them (in this lesson we will only solve three). The picture for the first problem is shown in Figure 11.2.

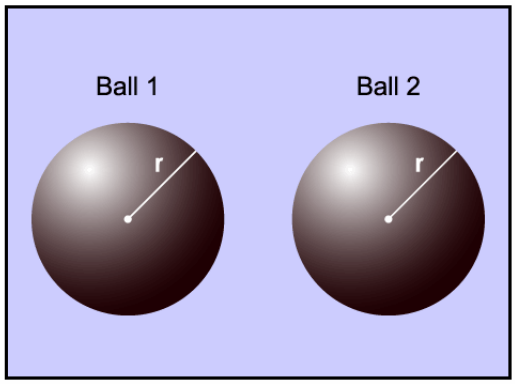


Figure 11.2: Detecting the collision of one ball with another ball.

In Figure 11.2 you can see two balls, each represented by a circle, and each having radius *r*. Task (1) is determining how to detect the collision of one ball with another ball. That translates into detecting when one of the circles either grazes the edges of the other or actually overlaps with it.

Task (1) depends on the distance between the balls. At far away distances, the balls will not penetrate each other. At closer distances, the balls will. So what we need to determine is the minimum distance that must separate the balls. If the balls are closer to each other than this minimum distance, then they have penetrated each other.

This raises the question, what do we mean by the distance between the balls? One way to define this is to say it means the distance between the *centers* of the balls. (You can define it other ways, however, and still solve the problem.) Presumably, we would represent the location of each ball as an (*x*, *y*) pair, which designates the location of the center of the ball. So if we designated the location of one ball by (x_1, y_1), and the location of the other ball by (x_2, y_2), then we could use the distance formula to calculate the distance between them. If we call this *d*, then we have,

$$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

Figure 11.3 shows the situation when the balls are so close that the edges touch each other. This would be considered a collision between the balls.

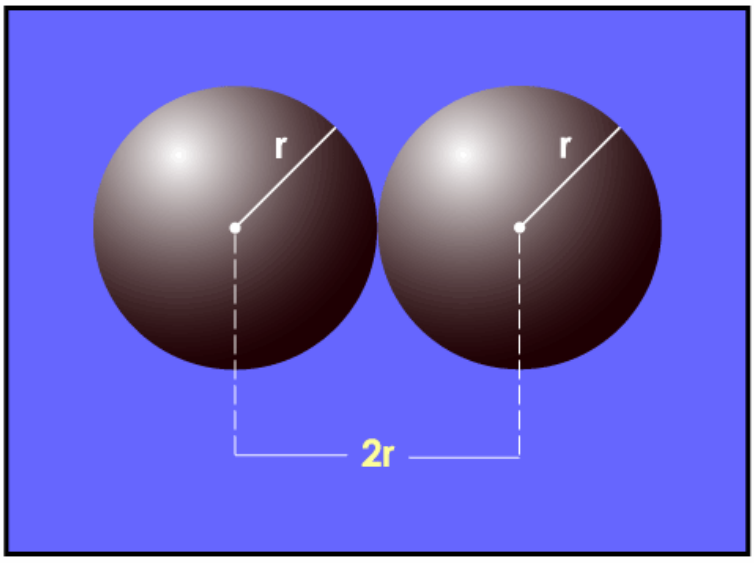


Figure 11.3: The case of collision.

Notice that from the way the figure is drawn, it is obvious that the minimum distance is 2*r*. (If this geometric deduction is not satisfying to you, you can represent each of the circles with an equation and prove that you will only get intersection points between the two circles if the distance between them is less than or equal to 2*r*.) Thus, if $d \leq 2r$, then we have a collision. Otherwise, we do not.

We have just solved task (1).

Figure 11.4 depicts task (2).

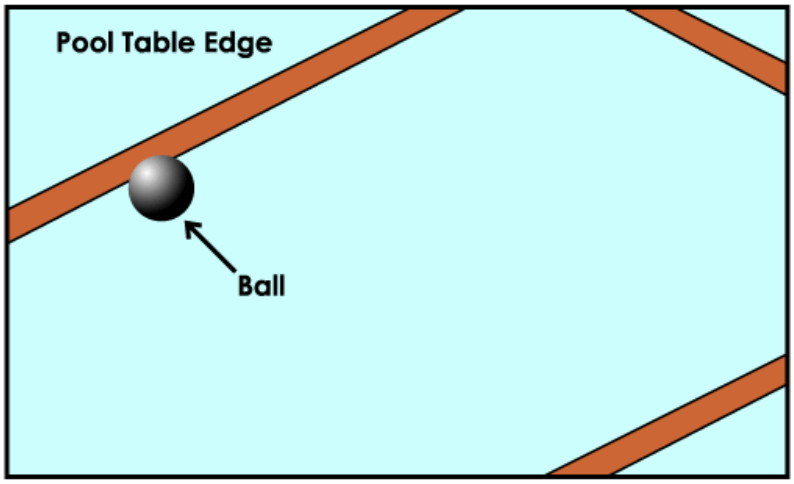


Figure 11.4: Detecting the collision of a ball with the edges of the pool table.

Task (2) is more complicated than task (1), since previously we just had to detect collisions between two circles. Now we have to detect collisions between a circle and the edges of the pool table.

We can represent the edges of the pool table with lines. The task then reduces to figuring out if a line and a circle intersect.

The solution to the last problem might suggest a solution to the current one. If the distance between the line and the center of the circle is less than the radius of the circle, then the ball has penetrated the edge of the pool table. Here the distance between the line and the center of the circle is defined as the "minimum distance" -- the distance along a vector perpendicular to the line. Figure 11.5 illustrates this idea.

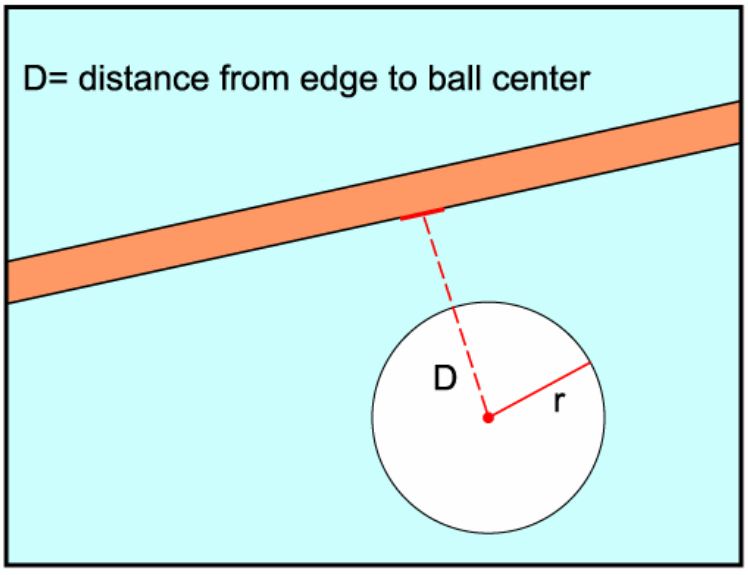


Figure 11.5: Detecting the collision of a ball with the edges of the pool table.

This is indeed a good way to solve the problem. We could determine what the distance is between a line and a point (it is not too hard), or we can just refer back to Chapter Seven, which stated that if you have a line with direction **u**, and a point on the line **p**, you can find the distance between the line and the point **q** with the following equation:

$$D = \frac{|(\mathbf{q}-\mathbf{p})\times\mathbf{u}|}{|\mathbf{u}|}$$

That wraps up task (2).

Tasks (3) and (5) require knowledge of college-level physics, so we will not solve them here. (The Game Physics course does indeed discuss and solve these problems, so you will get your chance to tackle this later in your studies.) But we can certainly solve task (4).

In task (4), the object is to figure out how a ball should rebound when it hits the edge of the pool table. If we ignore concepts like friction, the spin of the ball, and so on, the speed of the ball before impact is exactly the same as the speed of the ball after impact, and the outgoing angle is equal to the incoming angle. This is depicted in Figure 11.6.

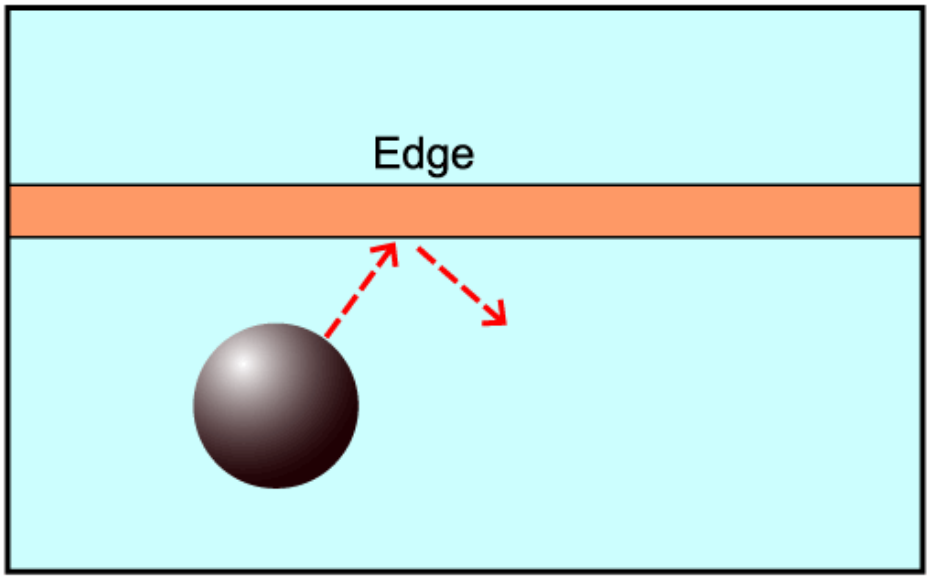


Figure 11.6: Determining how a ball should recoil when it hits the edges of the pool table.

We again need to represent the edges of the pool table with lines. The most convenient representation of a line here is the slope intercept form:

$y = mx + b$

A line perpendicular to the edge has slope $-1/m$ (from Chapter Six), so the vector **n** = <1, $-1/m$> is perpendicular to the edge. This vector, along with the incoming velocity of the ball, which we have designated **v**, and the outgoing velocity of the ball, which we have designated **v'**, are illustrated in Figure 11.7. (In physics, and here, velocities are vectors; they describe both direction and speed.)

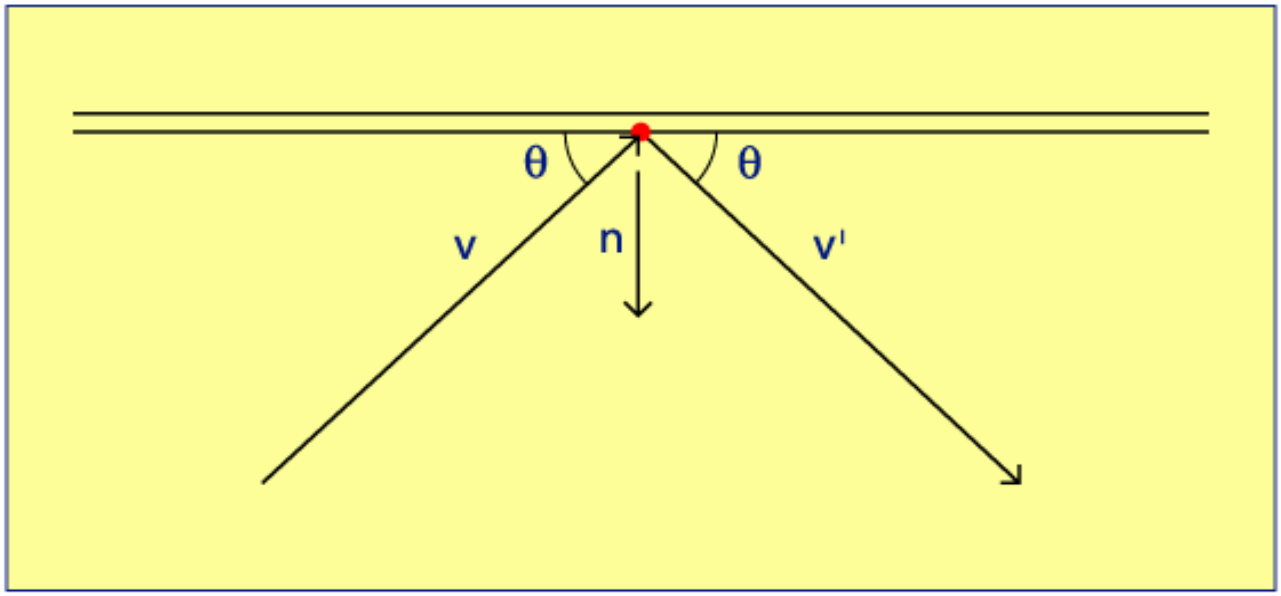


Figure 11.7: The vectors describing the situation.

After looking at the figure for a while, it may occur to you that if we project **v** onto **n**, we will get another vector, say **u**. Then if we form the difference (**u** - **v**), and add this result to **u**, we will get the vector **-v'**, and so we can just multiply this by -1 to get **v'**. This is indeed correct (see Figure 11.8).

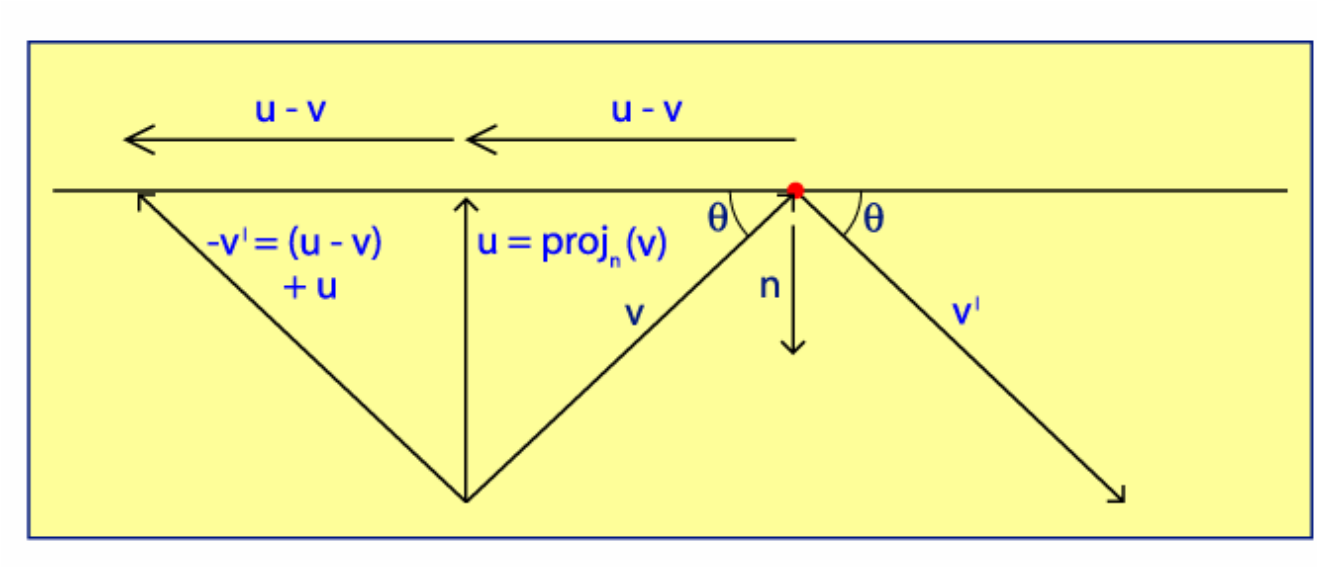


Figure 11.8: The solution to task (4).

So we can write **v'** as follows:

$$
\begin{aligned}
\mathbf{v}' &= \mathbf{v} - 2\,proj_{\mathbf{n}}(\mathbf{v}) = < v_1, v_2 > -2\left(\frac{\mathbf{n}\cdot\mathbf{v}}{\mathbf{n}\cdot\mathbf{n}}\right)\mathbf{n} \\
&= < v_1, v_2 > -2\left(\frac{< 1,-1/m > \cdot < v_1, v_2 >}{< 1,-1/m > \cdot < 1,-1/m >}\right) < 1,-1/m > \\
&= < v_1, v_2 > -2\left(\frac{v_1 - v_2/m}{1+1/m^2}\right) < 1,-1/m > \\
&= < v_1, v_2 > - < 2\left(\frac{v_1 - v_2/m}{1+1/m^2}\right), 2\left(\frac{v_1 - v_2/m}{1+1/m^2}\right)(-1/m) > \\
&= < v_1, v_2 > - < \frac{2m(mv_1 - v_2)}{1+m^2}, \frac{2(v_2 - mv_1)}{1+m^2} > \\
&= \cdots \\
&= < \frac{v_1(1-m^2) + 2mv_2}{1+m^2}, \frac{v_2(m^2-1) + 2mv_1}{1+m^2} >
\end{aligned}
$$

This formula is always valid, although keep in mind that using the slope-intercept form of the equation for a line, you can never represent perfectly vertical lines (and hence, you cannot have perfectly vertical edges for the pool table). However, you can represent lines that are as close to vertical as you want -- just increase *m* until the resulting line is indistinguishable from a vertical line. (Alternatively, you can use a different representation of the line, such as the vector form, and re-derive the equations, although the results will be somewhat more complicated.)

This result is applicable to more than just our pool game. If you wanted to create a two-dimensional ping-pong game, or any two-dimensional game that has some objects bouncing off of edges, then the equations above will do the job.

The next thing we could do is extend these physical reflection equations to the third dimension, but rather than do just that, we will instead look at how to do basic reflection (the equations are the same).

11.2 Reflection in 3D Games

One of the more amazing special effects that games are sporting these days is realistic reflection. Fly across a lake or look into a mirror in one of today's games, and you are more likely than ever to see something reflected back at you. How do the game developers pull off these feats? With magic of course -- the magic of mathematics.

If the reflective surface is flat, then games will often create a temporary viewer and place it in just the right place, with just the right orientation, so that what the temporary viewer sees is what the user sees reflected in the surface. The games will then place an image of what the temporary viewer sees into an *off-screen buffer* (a chunk of memory that holds graphics but is not visible to the user). Then when displaying the scene for the user, they will use the contents of this off-screen buffer to "paint" the

reflective surface, blending it with its own colors. (If the reflective surface was water, for example, the off-screen buffer would be blended with a picture of water.) Figure 11.9 shows the process.

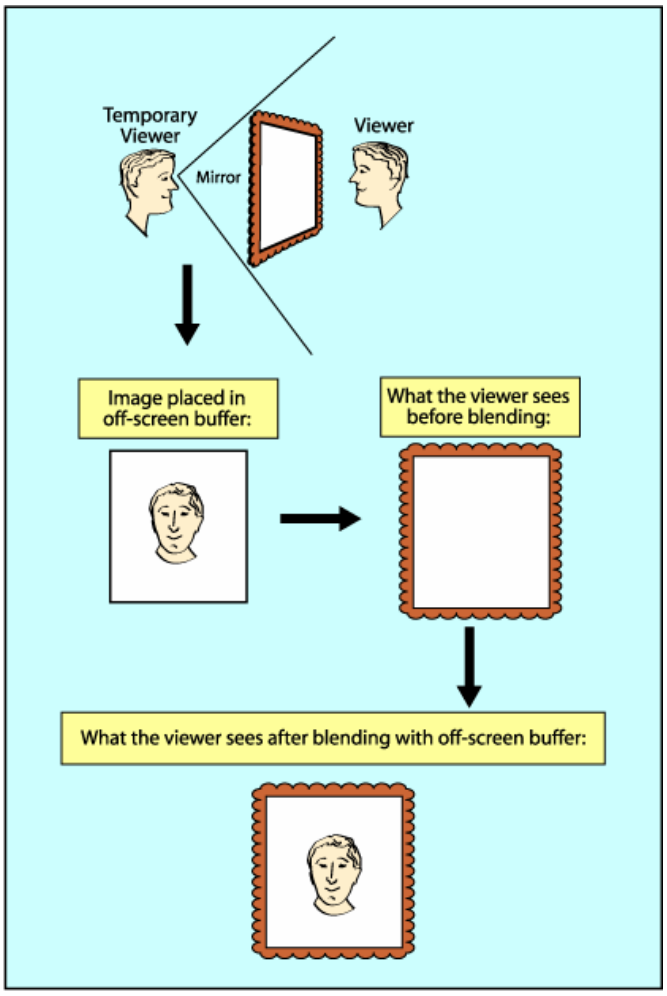


Figure 11.9: Reflection in games.

The challenging part of reflection is figuring out exactly where to place the temporary viewer and giving it an appropriate orientation. Of these problems, the former one is the easiest, so we will solve that one first.

When you look into a mirror that is positioned at a distance *d* from you, you see the reflected version of yourself at a distance 2*d* from you. (This can be proven with optical physics.) Another thing you will notice is that if you were to draw a straight line from your eyes to the eyes of your reflected self, the line would be perpendicular to the surface of the mirror. These facts suggest that we should place the temporary viewer a distance 2*d* from the real viewer, behind the reflective surface, along a line perpendicular to the normal of the reflective surface. This is illustrated in Figure 11.10.

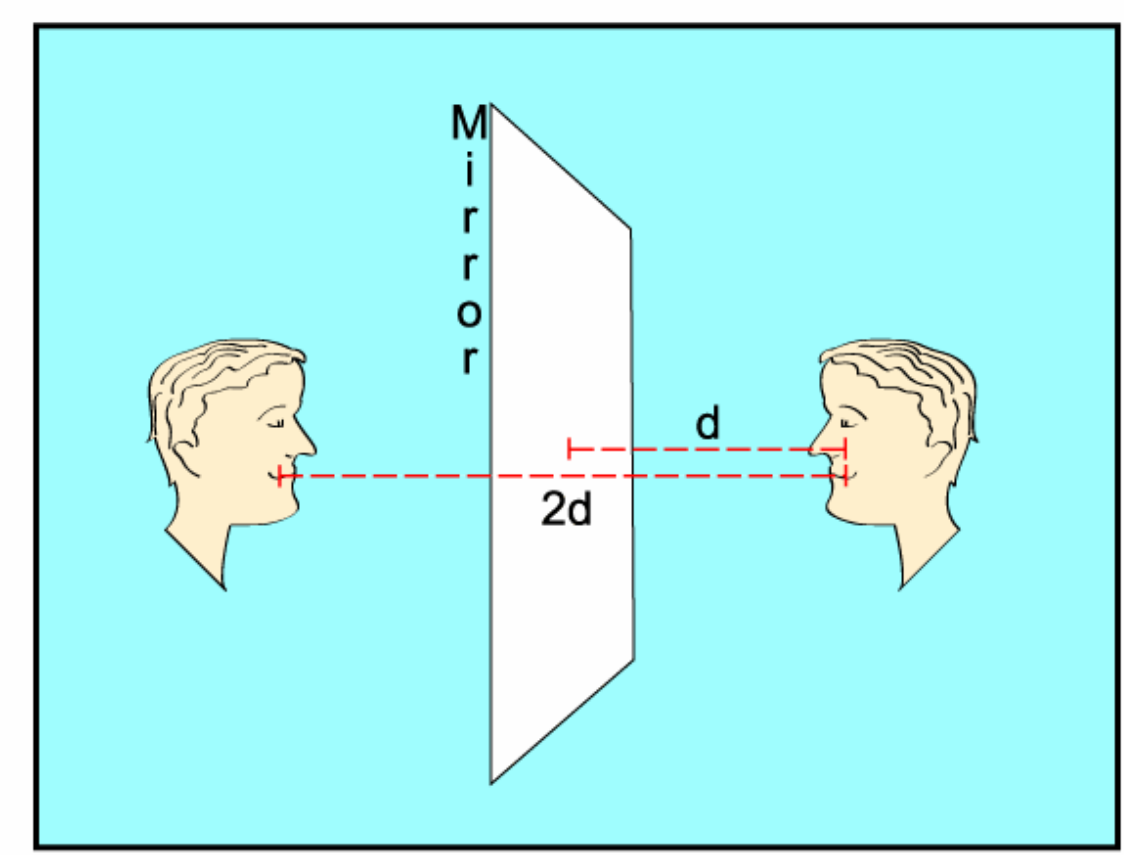


Figure 11.10: Placing the temporary viewer.

Note: Since the temporary viewer is placed behind the reflective surface, when creating the off-screen buffer, you must take care not to render the backside of the reflective surface.

Suppose the location of the viewer is given by the vector **v**, the normal of the reflective surface is given by **n**, and a point on the reflective surface is given by **p**. Then we can find the location of the temporary viewer in the following way. First, we project the vector (**v** - **p**) onto **n**. This creates a vector perpendicular to the reflective surface, with a magnitude equal to the distance between the viewer and the surface. Next, we multiply this vector by -2. This scales the vector so that its magnitude is equal to *twice* the distance between the viewer and the surface and changes its direction so that it points into the surface. Last, we add this vector to **v**, to give us **v'**, the location of the temporary viewer (see Figure 11.11).

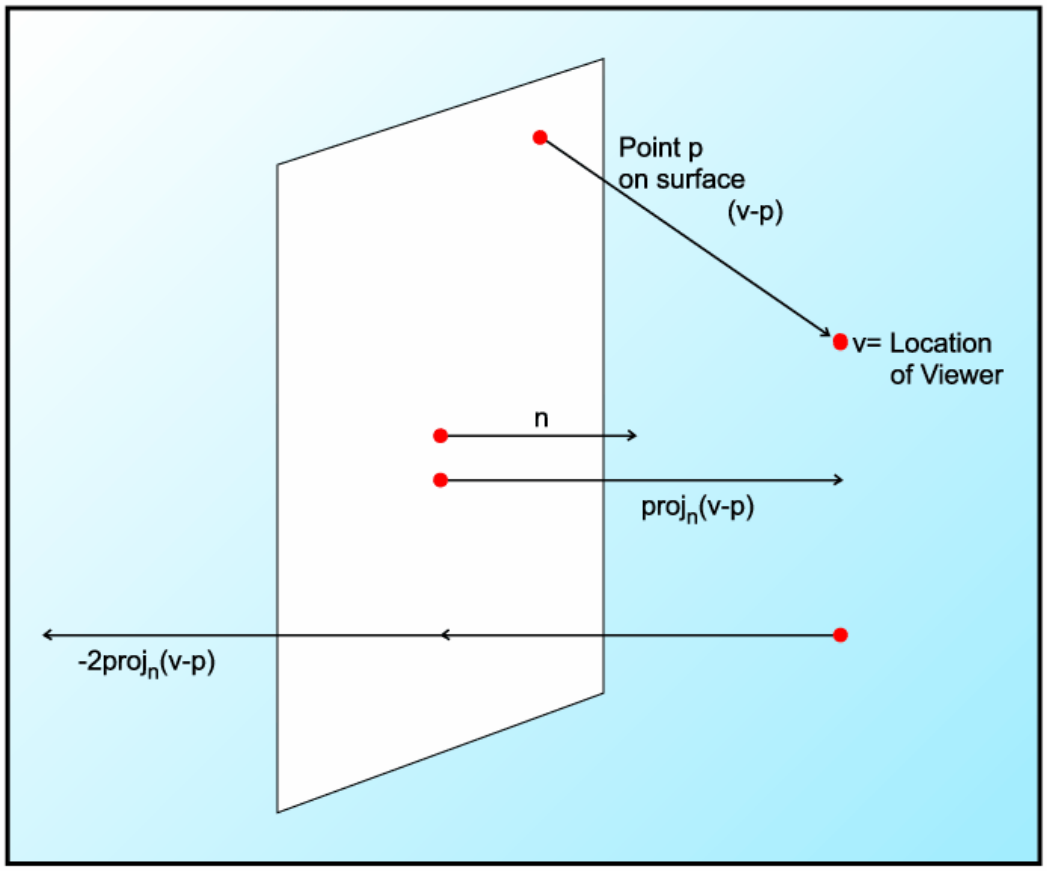


Figure 11.11: Deriving the location of the temporary viewer.

In mathematical notation, we can write the location of the temporary viewer as follows:

$$\mathbf{v}' = \mathbf{v} - 2\,proj_{\mathbf{n}}(\mathbf{v} - \mathbf{p})$$

Now we must solve the second problem. The easiest way to do this is to use the rotation matrix generated for the world-to-view transformation, since this matrix contains the viewer's up, right, and direction vectors (see Chapter Nine). To find the orientation of the temporary viewer, we just reflect each of these vectors on the surface. The result will be the up, right, and direction vectors for the temporary viewer, which we can use in *its* world-to-view transformation.

So now we need to know how to reflect a three-dimensional vector on a plane. This is not much harder than reflecting a two-dimensional vector on a line. Figure 11.12 shows the basic setup for the problem.

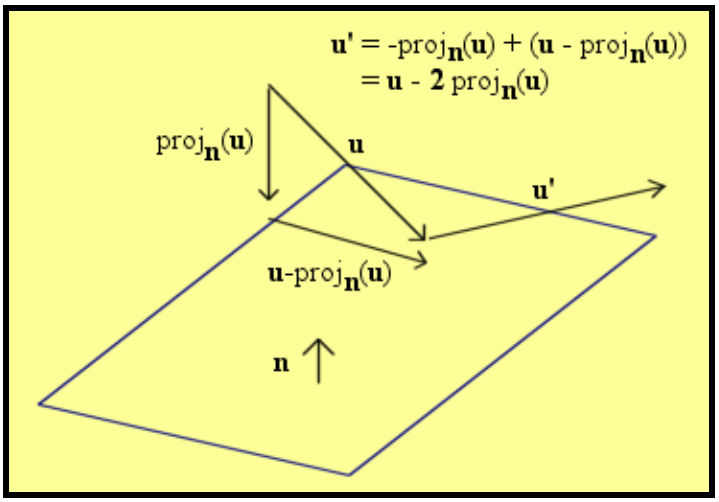


Figure 11.12: Reflecting a three-dimensional vector on a plane.

As suggested in the figure, given a vector **u**, a plane with normal **n**, and a point **p** on the plane, the reflected vector, **u'**, is given by the following equation:

$$\begin{aligned}\mathbf{u}' &= -(\mathbf{u} - 2(\mathbf{u} - proj_{\mathbf{n}}(\mathbf{u}))) \\ &= -\mathbf{u} + 2(\mathbf{u} - proj_{\mathbf{n}}(\mathbf{u})) \\ &= -\mathbf{u} + 2\mathbf{u} - 2\,proj_{\mathbf{n}}(\mathbf{u}) \\ &= \mathbf{u} - 2\,proj_{\mathbf{n}}(\mathbf{u})\end{aligned}$$

Notice this has the exact same form as the two-dimensional reflected vector.

Apply this formula to each of the rows of the rotation matrix, and you will get the rotation matrix for the temporary viewer. With this information, and the location of the temporary viewer (which we just calculated), you can create a new world-to-view transformation matrix. Use this to generate the scene from the point-of-view of the temporary viewer, but instead of displaying the result on the computer screen, store it in memory and use it to paint the reflective surface later on using texture mapping.

The operation of reflecting a vector is a linear transformation, so it has an induced matrix. Assuming **n** is the normal of the surface, the form of this matrix is shown below:

$$\begin{bmatrix} 1-\frac{2n_1^2}{\mathbf{n}\cdot\mathbf{n}} & -\frac{2n_1n_2}{\mathbf{n}\cdot\mathbf{n}} & -\frac{2n_1n_3}{\mathbf{n}\cdot\mathbf{n}} & 0 \\ -\frac{2n_1n_2}{\mathbf{n}\cdot\mathbf{n}} & 1-\frac{2n_2^2}{\mathbf{n}\cdot\mathbf{n}} & -\frac{2n_2n_3}{\mathbf{n}\cdot\mathbf{n}} & 0 \\ -\frac{2n_1n_3}{\mathbf{n}\cdot\mathbf{n}} & -\frac{2n_2n_3}{\mathbf{n}\cdot\mathbf{n}} & 1-\frac{2n_3^2}{\mathbf{n}\cdot\mathbf{n}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix form can be helpful if you want to reflect a collection of vectors on a given plane.

11.3 Polygon/Polygon Intersection

When objects in the real world collide, they either stop moving or ricochet off each other. When objects in a game world collide, however, they just sail right through each other -- unless, of course, you detect those collisions and prevent the objects from penetrating. That is what *collision detection* is all about.

The task is fairly straightforward once we simplify it to detecting collisions between two polygons. (Objects are made of polygons, so this simplification is no loss of generality.) In this case, collision detection becomes a matter of checking all the edges of the first polygon to see if they penetrate the second, and then all the edges of the second polygon to see if they penetrate the first. This is the only way the two polygons can penetrate each other (see Figure 11.13).

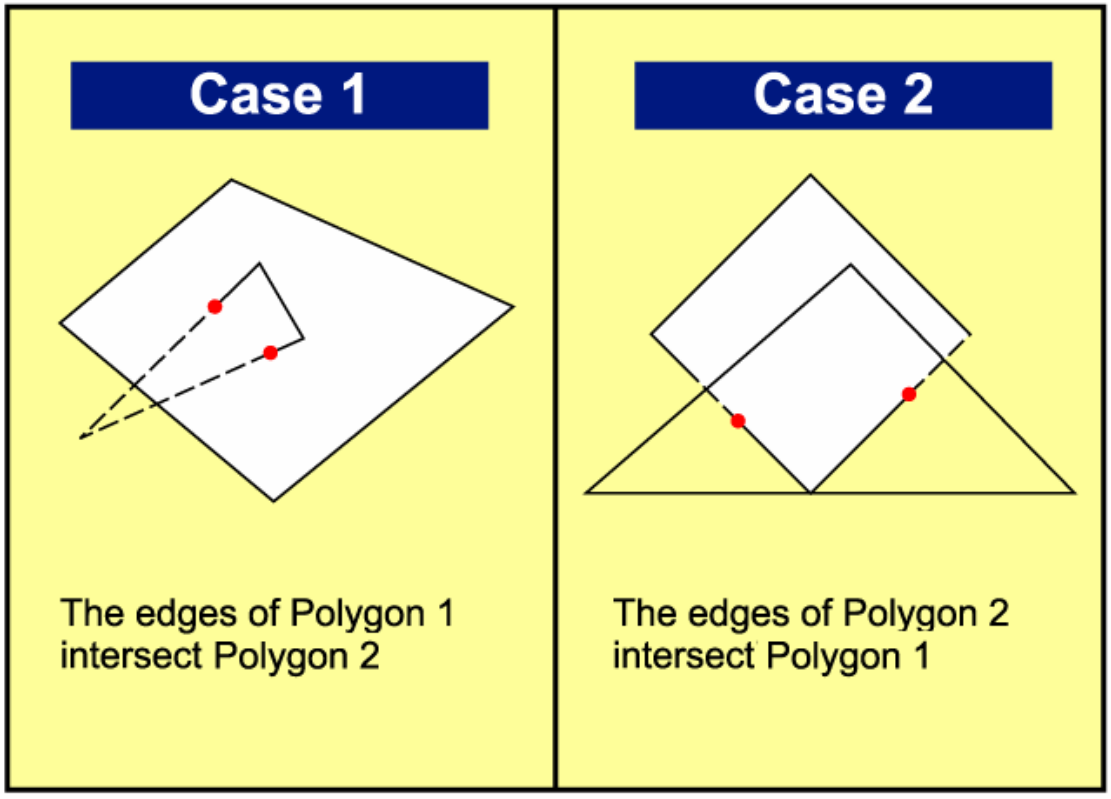


Figure 11.13: The two cases that lead to penetration.

Fundamentally, then, we need to be able to determine if a line segment (the edge of a polygon) penetrates a polygon. For someone who knows as much math as you, this is a proverbial walk in the park!

Here is the basic idea. First, check to see if the two points defining the line segment lie on opposite sides of the polygon. If they do not, then there is no possible way the line segment penetrates the polygon. If the points *do* lie on different sides of the polygon, however, then determine where the line segment intersects the plane containing the polygon. We call this the *intersection point*. Last, see if the intersection point actually lies within the boundaries of the polygon itself.

The first step is easy. Plug each of the points into the plane equation ($Ax + By + Cz + D = 0$). The equation will evaluate to zero for points on the plane, negative for points on one side, and positive for points on the other side (see Chapter Six). If the equation evaluates to zero for one of the points, then you have an intersection -- that point is the intersection point. Otherwise, just check the signs. If they are both negative or both positive, then both points lie on one side of the plane and there is no intersection. Otherwise, the points lie on opposite sides of the polygon and you must proceed to the next step.

The second step is easy too (we have done it before). Suppose the two points of the line segment are $\mathbf{p_1}$ and $\mathbf{p_2}$. We can create a parametric equation that describes all points on this line as follows:

$$P(t) = \mathbf{p_1} + (\mathbf{p_2} - \mathbf{p_1})t$$

Writing $\mathbf{p_1} = \langle p_{1x}, p_{1y} \rangle$, and $\mathbf{p_2} = \langle p_{2x}, p_{2y} \rangle$, the equation can be written:

$$p_x(t) = p_{1x} + (p_{2x} - p_{1x})t$$
$$p_y(t) = p_{1y} + (p_{2y} - p_{1y})t$$
$$p_y(t) = p_{1z} + (p_{2z} - p_{1z})t$$

We want the point that satisfies both of these equations and the plane equation, which we can get by substituting the above three equations into the plane equation, as done below:

$$A(p_{1x} + (p_{2x} - p_{1x})t) + B(p_{1y} + (p_{2y} - p_{1y})t) + C(p_{1z} + (p_{2z} - p_{1z})t) + D = 0$$

After much math, solving this equation for t gives:

$$t = -\frac{D + Ap_{1x} + Bp_{1y} + Cp_{1z}}{A(p_{2x} - p_{1x}) + B(p_{2y} - p_{1y}) + C(p_{2z} - p_{1z})}$$

$$= -\frac{D + \langle A, B, C \rangle \cdot \mathbf{p_1}}{\langle A, B, C \rangle \cdot (\mathbf{p}_2 - \mathbf{p_1})}$$

Plug this value of t into the parametric equation for the line segment, and you get the intersection point.

The third step is the most challenging. We have the intersection point and need to determine if it lies within the boundaries of the polygon. The problem would be much simpler if the polygon and point were both two-dimensional. So what we do first is project both polygon and point onto an axis-aligned plane -- the *x*-*y* plane, the *x*-*z* plane, or the *y*-*z* plane. This essentially involves simply dropping the *x*, *y*, or *z* coordinate, depending on which plane we pick (see Figure 11.14).

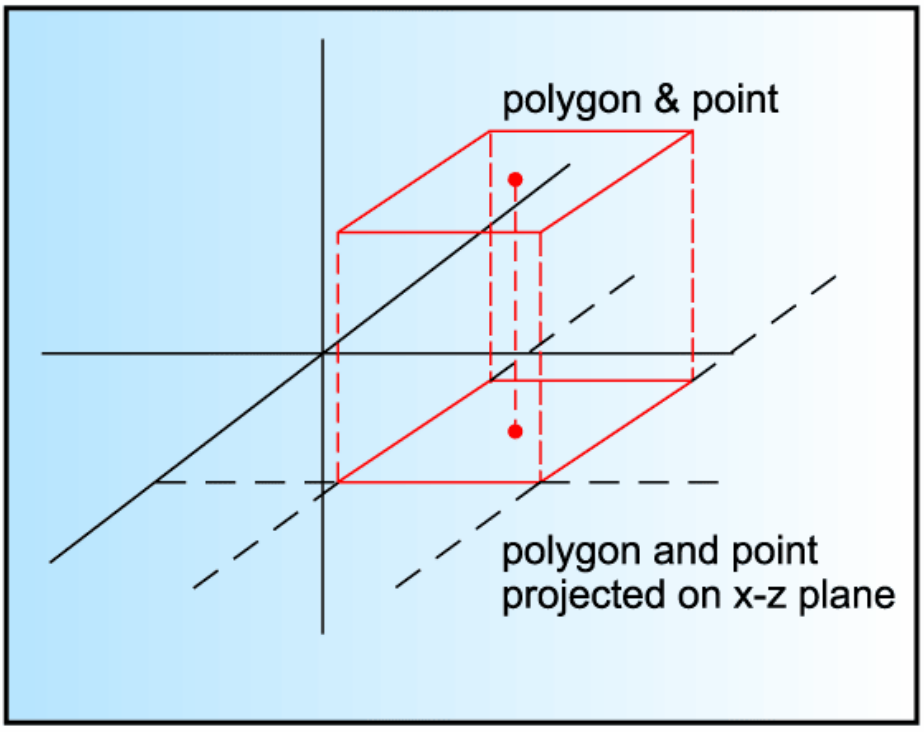


Figure 11.14: Projecting the polygon and the intersection point onto an axis-aligned plane.

In general, it does not matter what plane we use for the projection. But if the polygon is aligned with one of the planes, then projecting it onto that plane will produce a straight line, which makes it a bit difficult to determine whether or not the intersection point is inside the boundaries. For this reason, it is best to choose the plane whose surface normal (either in one direction or the other) most closely matches the normal of the polygon. This will ensure the broadest possible projection, as shown in Figure 11.15.

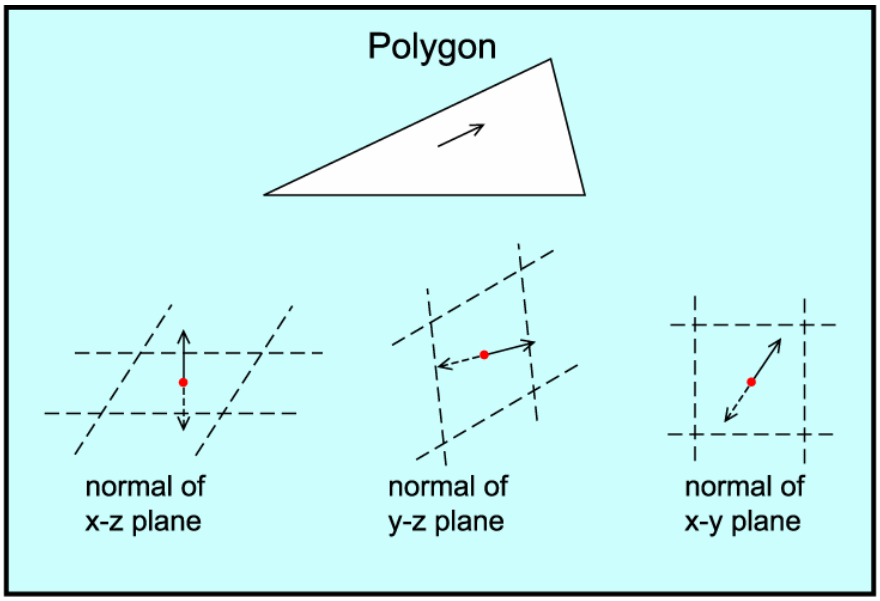


Figure 11.15: Choosing the projection plane based on the polygon's normal.

Once the problem is reduced to two-dimensions, it simplifies considerably. We only have to see if a two-dimensional point lies within the boundaries of a two-dimensional polygon.

To make the problem even simpler, we are going to assume that the polygon is convex. (That is, a line drawn from any two points in the polygon will never cross the boundaries of the polygon.) Since all polygons can be represented as a set of convex polygons, and since hardware generally only handles triangles (the simplest of all convex polygons), we do not really lose anything with this assumption.

This setup is shown in Figure 11.16.

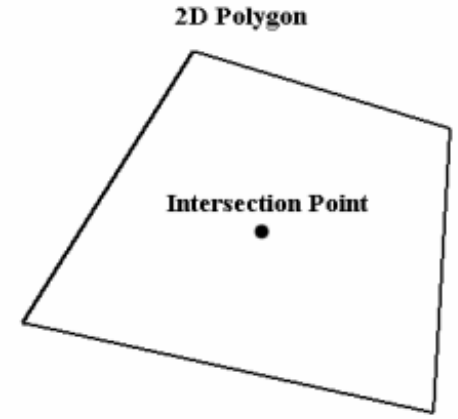


Figure 11.16: The simplified problem.

To determine whether a point lies within the boundaries of the polygon, we must first give those boundaries a mathematical representation. Suppose the points defining an edge of the polygon are given by the vectors $\mathbf{p_1} = \langle p_{1x}, p_{1y} \rangle$ and $\mathbf{p_2} = \langle p_{2x}, p_{2y} \rangle$. Then we can represent the edge parametrically by the following two-dimensional vector equation:

$$P(t) = \mathbf{p_1} + (\mathbf{p_2}-\mathbf{p_1})t$$

Notice the vector $\mathbf{p_2}-\mathbf{p_1}$ describes the orientation of the line. If we rotate this vector by $\pi/2$ radians, to get another that is perpendicular to the direction of the line, then we get the vector $\mathbf{p'} = \langle p_{2y} - p_{1y}, p_{1x} - p_{2x} \rangle$. (You can check this using the standard rotation formulas.) This vector gives us a way to check which side of the line a point is on. For checking the point **q**, all we have to do is form the vector $\mathbf{q}-\mathbf{p_1}$, and then dot it with $\mathbf{p'}$. Since the dot product is positive when the angle between the vectors is less than $\pi/2$, and negative otherwise, the sign of the dot product effectively tells us which side of the line the point **q** is on (see Figure 11.17).

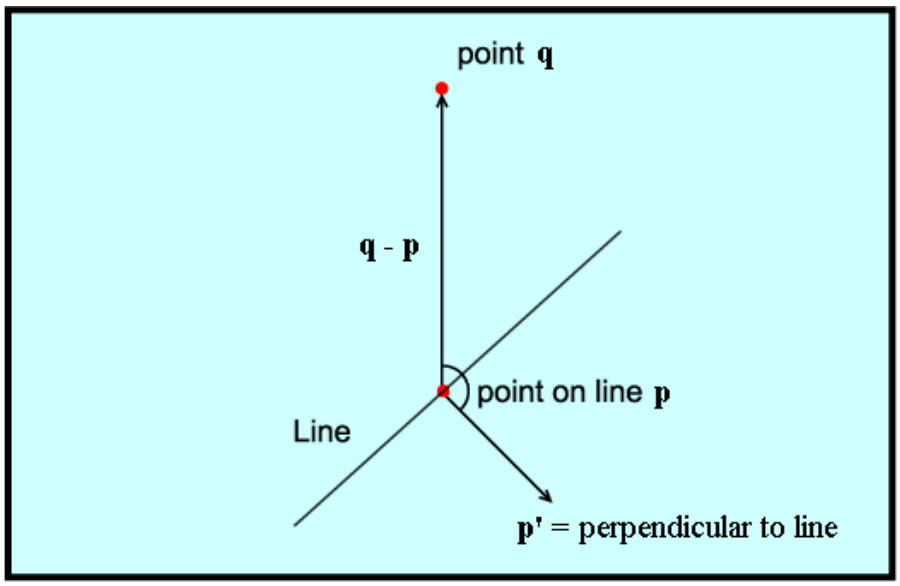


Figure 11.17: Determining which side of the line a point q lies on.

This helps us because all we have to do is test the intersection point against all the edges of the polygon. If the intersection point is on the interior side of each edge, then the intersection point lies within the boundaries of the polygon. Otherwise, it does not. This concept is illustrated in Figure 11.18.

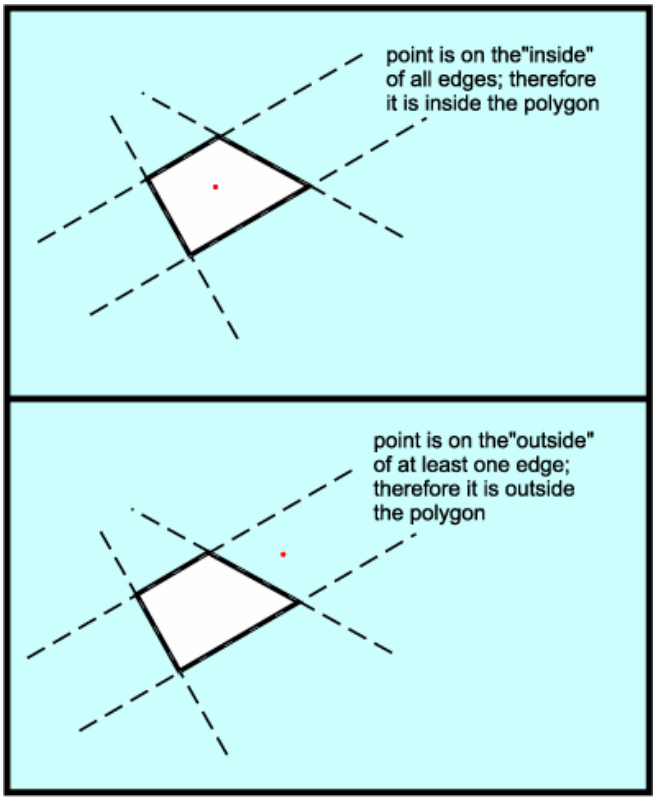


Figure 11.18: Checking to see if the intersection point lies within the boundaries of the polygon.

All we need to do now is determine which sign (positive or negative) goes with the inside of the polygon for each edge. This is easy. Just take a point known to be inside the polygon, send it through the above calculations, and check the sign of the dot product. If it is the same as the sign of the intersection point, then the intersection point lies on interior side. Otherwise, it lies on the exterior side.

You can find a point inside the polygon by averaging all the points defining the polygon. The resulting average is the geometric center of the polygon. (This trick works only with convex polygons.)

Let us summarize the procedure now. Suppose the points defining the polygon are given by the two-dimensional vectors $\mathbf{p_1}$, $\mathbf{p_2}$, ..., $\mathbf{p_n}$. Further suppose that $\mathbf{q}$ is the intersection point. Then to determine whether or not $\mathbf{q}$ lies within the boundaries of the polygon, follow these steps:

Compute $\mathbf{c} = \frac{\mathbf{p_1} + \mathbf{p_2} + \cdots + \mathbf{p_n}}{n}$, the geometric center of the polygon.

For each edge defined by $\mathbf{p_i}$, $\mathbf{p_j}$ (there are n+1 of them, the last defined by $\mathbf{p_n}$, $\mathbf{p_1}$), calculate the following quantity:

$$s = \left((\mathbf{c} - \mathbf{p_i}) \cdot < \mathbf{p_{jy}} - \mathbf{p_{iy}}, \mathbf{p_{ix}} - \mathbf{p_{jx}} >\right)\left((\mathbf{q} - \mathbf{p_i}) \cdot < \mathbf{p_{jy}} - \mathbf{p_{iy}}, \mathbf{p_{ix}} - \mathbf{p_{jx}} >\right)$$

If s is positive, then the signs match, and the intersection point lies on the same side of the edge that the geometric center does. If s is negative, however, they lie on opposite sides and you know the intersection point does not fall inside the polygon. If s is zero, then the intersection point falls exactly on the edge of the polygon, and how you handle this case is up to you.

If s is positive (or non-negative, depending on how you handle the zero case) for each of the n+1 lines, then the point lies inside the polygon, and otherwise it does not.

This is all rather straightforward, but unfortunately, it is also computationally expensive. To speed up the process, you should compute for each object a *bounding sphere*; that is, a sphere that completely encloses (bounds) all of the points of the object. Then if you want to see if two objects have collided, first check their bounding spheres to see if they penetrate each other. (This is very fast, since all you have to do is compute the distance between the centers of the spheres and compare it to the sum of the radii.) Only if the bounding spheres penetrate each other should you perform the polygon-to-polygon intersection test outlined above.

Computing a bounding sphere is not difficult at all. Just find the geometric center of the object, call that the center of the bounding sphere, and compute the distances from the geometric center to all the points defining the object. The maximum distance is the radius of the bounding sphere.

Once you know a collision has taken place, you need to handle it. In general, this requires some heavy-duty physics, but if the collision is between the viewer and, say, a polygon, then you probably just want to slide the viewer along the polygon. One of the exercises at the end of this chapter asks you to derive equations to handle this. (Hint: use vector projection and the normal of the polygon.)

In the next section, we will derive the math behind the shadows you see in most of today's games.

11.4 Shadow Casting

Realistic lighting models can produce stunning results (as demonstrated in the movie *Final Fantasy*), but unfortunately, the better the mathematical model, the more time it takes the computer to display the scene. Since games have to update the display very frequently (more than 30 times per second) in order to produce fluid animation, they usually have to compromise when it comes to lighting.

In the case of generating shadows, the compromises are significant. In general, only point light sources are permitted (sources that emanate light from a single point), light is not allowed to bend (it travels in a straight line, thus producing very sharp shadows), and only select polygons can cast shadows (it is typically too time-consuming to generate real-time shadows for all polygons in a game world).

Still, even with these simplifications, the results can be impressive, and are certainly far more realistic than having no shadows at all. One only has to look at a game like Doom III™ to see the amazing results of real-time shadows.

So how can we add shadows to a game? Well if you want an object to cast shadows on very simple surroundings, like a flat, level surface, then you can project the polygons comprising the object onto the surface by creating rays from the light source to the vertices of the polygons. The intersection of these rays with the surface will give you a number of polygons that you can display with a dark, slightly transparent texture. (You just have to be sure not to shade the same pixel twice.) This method of shadow casting is shown in Figure 11.19.

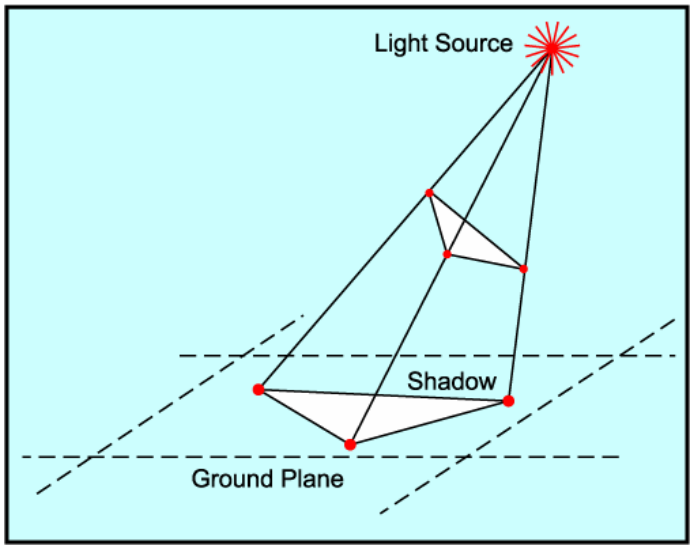


Figure 11.19: A simple method of shadow casting for planes.

If the geometry of the environment is more complex, however, then the above method will not work unless you add a lot of computational overhead (like clipping the projected polygons to the shape of the polygons they are projected on -- *really* not something you want to do in real-time). For complex environments, games often use a more general method based on the concept of *shadow volumes*.

Imagine rays extending from the light source to the vertices of a polygon, and then further outward to some finite (but faraway) distance, where they intersect the vertices of an enlarged copy of the original polygon. The volume defined by these rays and two polygons is called a *shadow volume*, and objects located in the volume are to be shadowed. Figure 11.20 shows an example shadow volume.

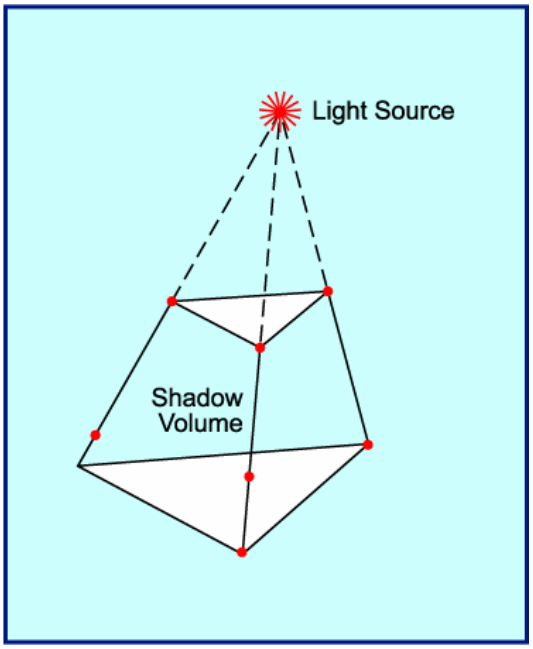


Figure 11.20: A shadow volume for a polygon.

The faces of the shadow volume, including the two polygons at the ends, are called *shadow polygons*. The *backside* of a shadow polygon is defined as the side that faces inward; the *frontside* is defined as the side that faces outward. These distinctions are shown in Figure 11.21.

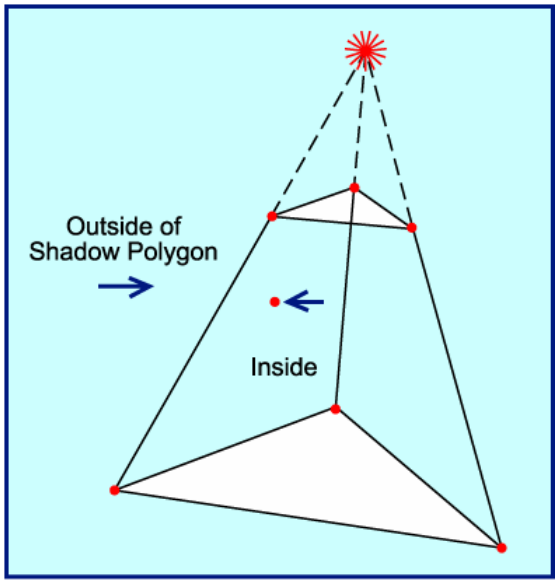


Figure 11.21: The two sides of a shadow polygon.

The basic procedure for determining if a point falls in the shadow is simple. Count the number of frontsides the viewer sees at that point, and call this *x*. Then count the number of backsides the viewer sees at the point, and call this *y*. If $x > y$, then the point falls in the shadow, and otherwise it does not. Figure 11.22 shows how this works for a couple of sample points.

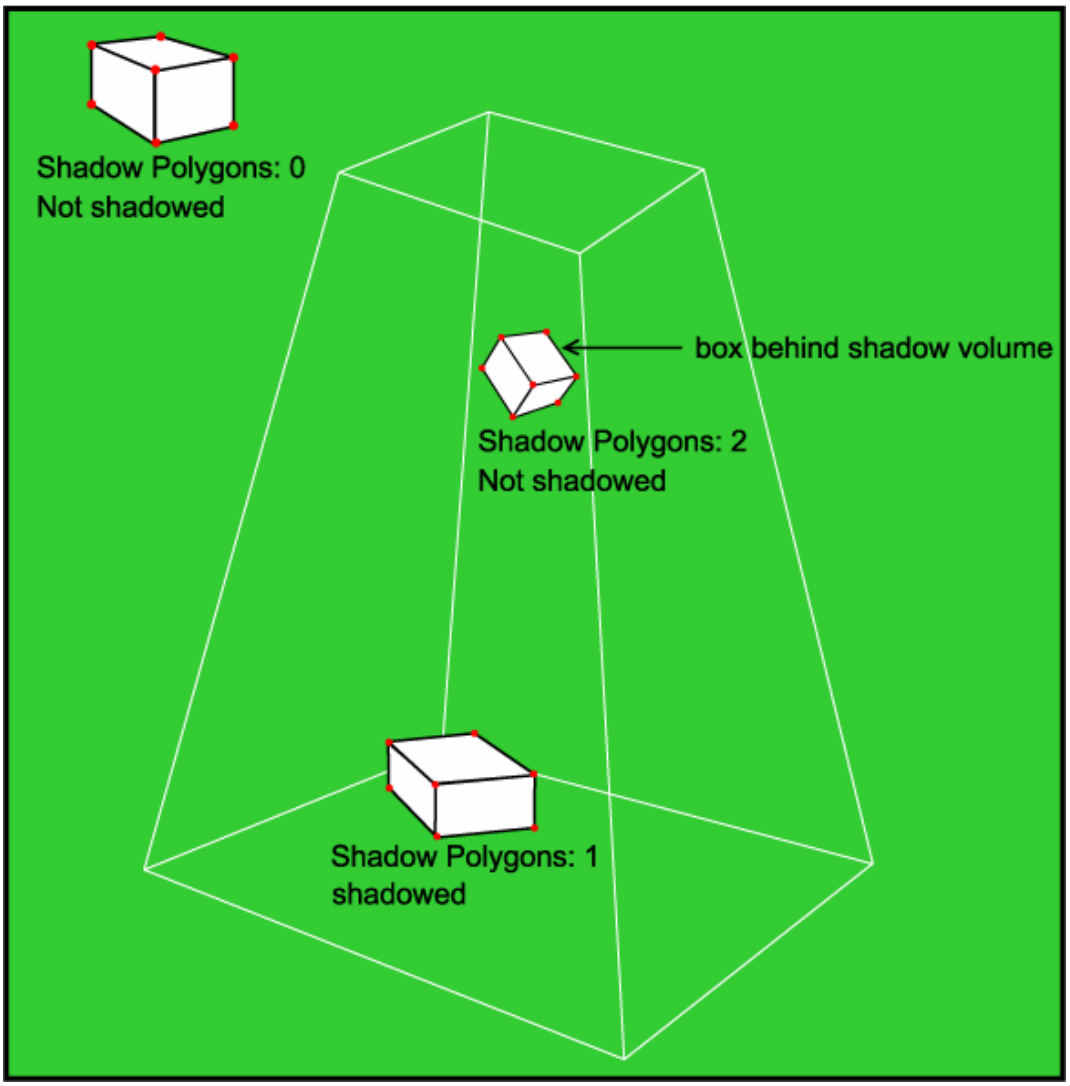


Figure 11.22: Determining whether or not points should be shadowed.

The only math that this method requires is the creation of the shadow polygons. You can do this by first giving each ray a parametric representation. If $\mathbf{p_i}$ is the *i*th vertex of the polygon, and **c** is the location of the light source, then the following equation describes the *i*th ray:

$$R_i(t) = \mathbf{c} + (\mathbf{p_i} - \mathbf{c})t$$

Each polygon edge creates a four-sided shadow polygon (a quadrilateral). You can create both of these polygons and the enlarged polygon at the end of the shadow volume by choosing a sufficiently large value of *t* in the above equation. (Keep in mind that points outside the shadow volume never get

shadowed.) Strictly speaking, you do not need to create the enlarged polygon, since the shadow volume should extend far enough out that the viewer is not likely to go beyond it.

If you are feeling ambitious, you might want to explore the idea of creating a shadow volume for an entire object, rather than just a single polygon. This can be a bit tricky, but if the object does not move relative to the light source, then this approach will be much faster, since you only have to compute the shape of the shadow volume once. One way of computing such a shadow volume is to project the entire object onto a plane perpendicular to the light source, and then generate shadow polygons for only those edges that are on the outside of the projected shape. You can safely ignore all the interior edges, which will result in an increase in performance.

Shadow volumes and other forms of real-time shadowing in games are explored in detail in the Graphics Programming course series (Parts II and III), so you will encounter these concepts and their implementation a little later in your studies.

In the next section, we will talk about the basic principles of lighting in games.

11.5 Lighting in 3D Games

The general problem of lighting is quite complicated. Each point in the world can receive light from light sources and also from all the objects around it. Further, each point reflects light to some degree depending on the material properties of the substance reflecting the light. There do exist techniques to model this complexity, most notably *radiosity*, but even supercomputers can take hours or days to compute lighting for scenes using these techniques. (You will learn more about implementing radiosity and other complex lighting models in the Graphics Programming course series later in your studies).

So for the time being, simpler models will have to suffice. Just how simple? Most games today look at two things when determining how to light a point; whether or not there exists a clear line-of-sight between the point and a light source and what the orientation of the surface is relative to the light source at that point.

Checking for a clear line-of-sight is straightforward. Create a line segment from the light source to the point, and then see if it intersects any polygons in between. (Bounding spheres improve performance, but are not necessary unless the objects or light sources are moving, since you can just generate the lighting information once and then reuse it.)

You can determine the orientation of a surface relative to the light source by computing the dot product of the surface's normal and a vector describing the orientation of the light source. Then you can either hit the dot product with the inverse cosine function to get the angle between the two vectors (this assumes the vectors are normalized, if not you will first have to divide by the product of their magnitudes) or use the dot product directly to compute the shade at that point. Surfaces directly facing the light source should be brightest, and surfaces facing away from the light source should not receive any direct light from that source (see Figure 11.23).

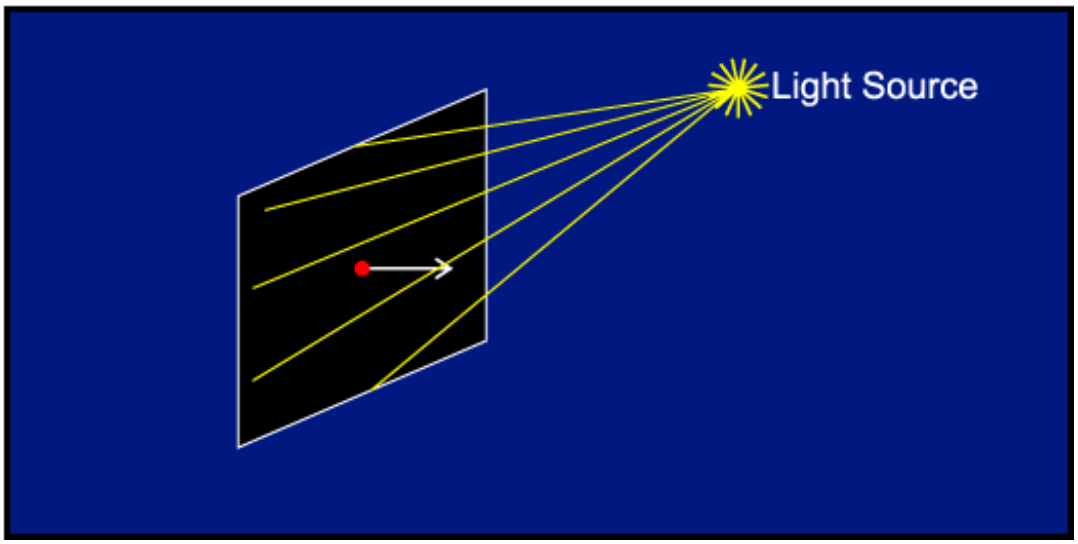


Figure 11.23: Shading polygons based on their orientation relative to a light source.

Of course, it would be a bit impractical to shade each and every point in the world. So what some games do instead is shade a single point on a polygon and use color that for the entire polygon. Other games will shade all of the vertices of the polygon and then smoothly interpolate the shading values across the face of the polygon when displaying it. This can produce crude shadowing, but nothing approaching the realism of the technique covered in the last section (not unless you use many extremely small polygons). There are other forms of lighting as well which we will discuss in the next chapter. Again, you will get into the real details on lighting in games when you begin your studies in the Graphics Programming course series. For now, we just want to nail down a theoretical foundation for understanding the techniques.

Conclusion

The applications we have looked at in this chapter are just a few of the countless ways to use math in computer games. In addition to applications in computer graphics, math is relied on exclusively in the realm of physics simulation -- a topic you will examine later when you take the Game Physics course. Even artificial intelligence makes heavy use of mathematical constructs like matrices and vectors and you will have a chance to look at some of these applications in the Introduction to Artificial Intelligence course a little later in your training.

Exercises

*1. Suppose the viewer is located at (*x*, *y*), with a conical field of view of θ, and that an object with bounding sphere *r* is located at (*s*, *t*). (See Figure E11.1.) What is the angle between the viewer and the side of the bounding sphere closest to the viewer's field of view?

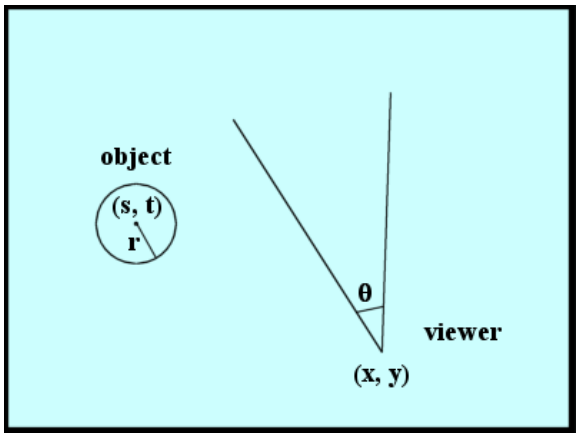


Figure E11.1: The setup for Exercise 1.

2. For this exercise, you will need the result of Exercise 1. Suppose the viewer can turn to the left or right at a maximum of ω radians per second. If the viewer is stationary (or moving strictly forward), what is the minimum number of seconds required before the viewer can see the object mentioned in Exercise 1? (You can use this information to avoid transforming and displaying the object for the calculated number of seconds. Optimizations such as this exploit *frame coherence* -- the tendency of one frame to resemble the next.)

*3. Devise a means of calculating the union, intersection, and difference of two objects composed entirely of triangles. (These operations are used extensively in 3D design programs like GILES™.)

*4. Invent an angle-based method for determining if a 2D point falls within a 2D triangle.

5. In previous exercises, you discovered how to determine if a computer character can see the player. But the technique used did not take into consideration the presence of objects that may be obstructing the character's line of sight. To fix this, you can send out rays from the character to a number of points on the player, and see how many of the rays intersect objects on their way from the character to the player. Describe how you can use the result of Exercise 4 to solve this problem.

6. Devise a fast method for determining if a polygon lies within an axis-aligned box.

!7. One method of shadow generation places a temporary viewer at the light source, and renders the view into an off-screen buffer. But instead of rendering colors into the buffer, the only information that is stored is the depth of each pixel. Describe how this information can be used to generate shadows, and derive any necessary equations.

Course Conclusion

After finishing your exercises in this lesson, you will want to leave some time to begin preparing for your final examination. However, you are encouraged to try writing some of your own software that uses either some of the equations we have derived in this course, or new ones you come up with yourself. One way or the other, as you progress through the rest of your training here at Game Institute, this is going to happen anyway.

Certainly, your training as a game developer is far from over. At this point you have gained some good insight into the mathematics that will be encountered as you work your way through the rest of the Game Institute curriculum, but there is much left to do. You will certainly have noticed that throughout this course we have referred to topics that will be studied in greater depth in other courses. Specifically, we mentioned the Graphics Programming course series a number of times. This was not done to duck the issue, but to allow you to tackle those concepts in a more appropriate setting where you can use them in a hands-on way.

As you might have guessed, the Graphics Programming series (a multi-part series of courses that focuses on 3D graphics and game engine design) is where the true core of your game engine development training will occur. As you progress through the GP series, you will certainly see all of the mathematics we have discussed (and even some that we have not) being used in many different ways. Every single algorithm and technique that we only briefly touched upon (lighting and shadows, collision detection, reflections, etc.) will be thoroughly analyzed and put to use in an actual 3D game engine that you will build yourself over the course of the series. By the time you are done, not only will you have a thorough understanding of game mathematics in the context of a real game application, but you will also have a very powerful game engine that you can use to build your own commercial quality games.

Keep in mind that your training will not stop there, nor will your use of mathematics. As mentioned a number of times, you will also try your hand at incorporating proper physics into your real-time game engine when you take the Game Physics course. You will even get to use a good deal of the math we discussed here in this course when you study artificial intelligence in the Artificial Intelligence course. Truth be told, there is really not a single course that you will encounter here at the Game Institute that does not include at least some of the math techniques we have discussed. So you are now in a much better position to be able to not only *use* the math, but to understand why you are doing so and how it all works.

Best of luck to you as you continue on with your game programming adventures!