

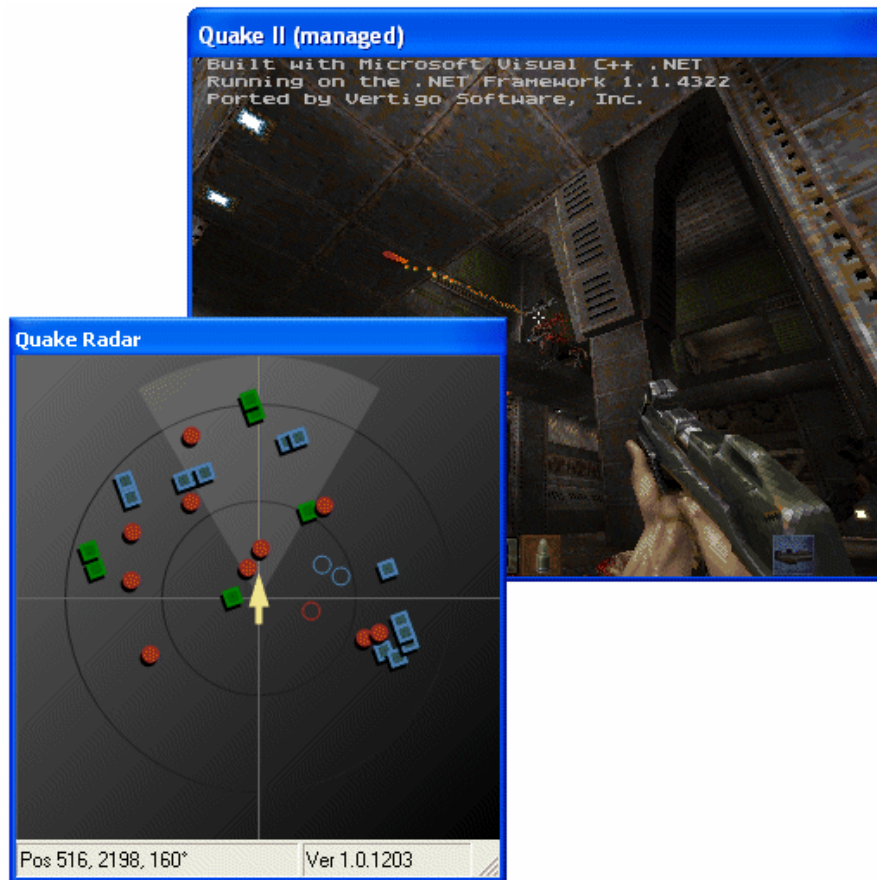
Quake II .NET

Ralph Arvesen, Vertigo Software, Inc.
July 2003

In 1997, the computer gaming company [id Software](#) released a watershed first-person shooter game called **QUAKE II**, which went on to sell over one million copies and earn industry accolades as Game of the Year. Later, in December 2001, id Software generously made the QUAKE II 3-D engine available to the public under the [GNU General Public License](#) ("GPL").

Now, in July 2003, Vertigo Software, Inc. is releasing **Quake II .NET**, a port of the C-language based engine to Visual C++ with a .NET managed heads-up display. We did this to illustrate a point: one can easily port a large amount of C code to C++, and then run the whole application as a managed .NET application using the Microsoft Common Language Runtime (CLR) without noticeable performance delays. Once running as a .NET managed application, adding new features is easy and fun.

This paper discusses what was involved in porting and extending the Quake II engine to Quake II .NET.



Source Code and Files

The complete source code for the Quake II engine is available from id Software at <ftp://ftp.idsoftware.com/idstuff/source/quake2.zip>. The source code is released under the terms of the GNU General Public License ("GPL"). You should read the accompanying **readme.txt** and **gnu.txt** files for more information on the GPL.

There are two parts to the Quake game: the engine and the data.

Game engine

The engine is the code that runs the game and is composed of the following files:

File	Description
quake2.exe	The main game executable.
ref_soft.dll	The software rendering engine.
ref_gl.dll	The OpenGL rendering engine.
gamex86.dll	The core game engine.

The managed version of Quake II .NET contains an additional file that implements the radar extension:

File	Description
Radar.dll	Managed C++ radar extension.

Game data

Maps, monsters, weapons, and other game essentials are contained in a data file (often packaged in a single PAK file) in the **baseq2** folder. Multiplayer data is stored in the **baseq2\players** folder.

How to Run Quake II .NET

Vertigo provides the five files above. However, you need just one more file, **pak0.pak** in order to run Quake II .NET. The PAK file contains id Software's copyrighted 3-D models and images and they'd prefer that you get that file from them:

All of the Q2 data files remain copyrighted and licensed under the original terms, so you cannot redistribute data from the original game... -John Carmack, id Software, from the **readme.txt** in the GPL'd source code

Therefore, you need to get the PAK file from the official Quake II demo. Here's what you need to do:

1. Install the **Quake II .NET.msi** from www.vertigosoftware.com/quake2. This installs the files for the native and managed versions.

2. Download and open the Quake II demo from <ftp://ftp.idsoftware.com/idstuff/quake2/q2-314-demo-x86.exe>. This unpacks files to your system (default folder is c:\windows\desktop\Quake2 Demo).
3. Copy the **pak0.pak** file from the **Quake2 Demo\Install\Data\baseq2** folder to the **%ProgramFiles%\Quake II .NET\managed\baseq2** and **%ProgramFiles%\Quake II.NET\native\baseq2** folders. This file is big—about 48MB and you’re making two copies. If you uninstall Quake II .NET, you have to remove these two copies by hand.
4. Run the managed or native version by clicking the shortcut in the Start menu.

How to Build the Code

Building the code is straightforward but you need to copy the generated EXE and DLLs to the runtime before running the app. The steps are outlined below:

1. Unzip the Quake II .NET source ZIP file. This contains the Quake engine code ported to Microsoft® Visual C++® .NET 2003.
2. Open the **quake2.sln** file.
3. Select the target configuration (release or debug, native or managed) and build the solution. Files are generated in the specified build configuration (**Release Managed** for example).
4. Copy the engine files from the source location to the Quake II .NET runtime installation (the default folder is **%ProgramFiles%\Quake II .NET**). The following table shows what files to copy:

File	Copy To
quake2.exe	\Program Files\Quake II .NET\
ref_soft.dll	\Program Files\Quake II .NET\
ref_gl.dll	\Program Files\Quake II .NET\
gamex86.dll	\Program Files\Quake II .NET\baseq2
Radar.dll	\Program Files\Quake II .NET\ (only required for the managed version)

Copying the files could be automated with a custom post build step.

How We Ported the Code

The source code was downloaded from id Software’s FTP site as discussed above. This code contains a Visual Studio 6 workspace file named **quake2.dsw**. When

opening this file, Visual Studio prompts you to update the project files and generates a solution file named **quake2.sln**. The following changes were made to the projects:

- Platform-specific code was removed by removing assembly files and disabling inline assembly routines.
- The project configurations were modified to include **Debug Managed**, **Debug Native**, **Release Managed** and **Release Native** builds.
- All of the files had the **Compile as C code (/TC)** switch specified. Instead of renaming all of the source files with a CPP extension, the **Compile as C++ code (/TP)** switch was specified.

Once the build environment was set up, it was time to port the code to C++.

Porting to Native C++

The following issues were encountered during the port from C to C++.

Keywords

The C++ language reserves keywords that are not reserved in the C language. For example, the Quake code uses a variable called **new** which was renamed to **new_cpp** for the C++ version.

```
// C
qboolean new;

// C++
qboolean new_cpp;
```

The Quake code defined its own boolean type with **true** and **false**. These are reserved keywords in C++ so this was typedefed as a **bool** as shown below.

```
// C
typedef enum {false, true} qboolean;

// C++
typedef bool qboolean;
```

Strong typing

The C++ language is strongly typed so assignment and function arguments require casting if the types don't match. This was the largest portion of the port. Though tedious, it was easy to port since the compiler identified the exact problem including source file, line number, and the required cast. An example is shown below.

```
// C
pmenuhnd_t* hnd = malloc(sizeof(*hnd));

// C++
pmenuhnd_t* hnd = (pmenuhnd_t*)malloc(sizeof(*hnd));
```

The Quake code uses **GetProcAddress** to dynamically retrieve the address of functions in other DLLs. All of the calls required casting and it was quickly noted that there were a lot of these calls. A script was created for the portion of the code that read the build log and modified the source code with the proper cast. An example of the required cast is shown below.

```
// C
qwglSwapBuffers = GetProcAddress (
    glw_state.hinstOpenGL, "wglSwapBuffers" );

// C++
qwglSwapBuffers = (BOOL (__stdcall *) (HDC)) GetProcAddress (
    glw_state.hinstOpenGL, "wglSwapBuffers" );
```

The C language does not require that declarations exactly match definitions or declarations in other source files and this caused compiler errors: **C2371** (redefinition; different basic types) and **C2556** (overloaded functions only differ by return type). Function declarations and definitions were modified to resolve any conflicts. For example, the function declaration below was changed to return an **rserr_t** instead of an **int**.

```
// C
int GLimp_SetMode( int *pwidth, int *pheight,
    int mode, qboolean fullscreen );

// C++
rserr_t GLimp_SetMode( int *pwidth, int *pheight,
    int mode, qboolean fullscreen );
```

If **extern** was used to declare the function that was defined in another file, this error was not caught until link time and appeared as an unresolved external.

Using COM objects

The calling convention for COM interfaces is different between C and C++ because vtables (virtual function table) are supported in the C++ language. For the C language, the vtable of the COM interface is explicitly accessed and a 'this pointer' is passed as the first argument. An example that calls the **Unlock** method of a COM object is shown below.

```
// C
sww_state.lpddsOffScreenBuffer->lpVtbl->Unlock(
    sww_state.lpddsOffScreenBuffer, vid.buffer );

// C++
sww_state.lpddsOffScreenBuffer->Unlock(
    vid.buffer );
```

Porting to Managed C++

Managed code runs within the context of the .NET run-time environment. It is not compulsory to use managed code, but there are many advantages to doing so. A program written with managed code using Managed Extensions for C++, for

example, can operate with the common language runtime to provide services such as memory management, cross-language integration, code access security, and automatic lifetime control of objects.

The first step required when porting native C++ to managed C++ is to set the **/CLR** compile switch by enabling Managed Extensions for C++. Depending on your project, this might be the only step required, but the following errors were also encountered when porting Quake to managed C++.

Incompatible switches

The **/clr** and **/YX** (Automatic Use of Precompiled Headers) switches are incompatible so the **/YX** switch was turned off for managed builds.

Mixed DLL loading problem

The code compiled but all of the projects that generated DLLs had the following link warning.

```
LINK : warning LNK4243: DLL containing objects compiled with /clr is
not linked with /NOENTRY; image may not run correctly
```

This warning occurs when a managed C++ DLL contains an entry point and the linker is letting you know a deadlock scenario could occur during the loading process. This was fixed by doing the following:

- Add the **/NOENTRY** link switch.
- Link with the **msvcrt.lib** library.
- Include the symbol reference **DllMainCRTStartup@12**.
- Call **__crt_dll_initialize** and **__crt_dll_terminate** in the DLL.

The following articles explain the details of this issue:

- PRB: Linker Warnings When You Build Managed Extensions for C++ DLL Projects - <http://support.microsoft.com/?id=814472>.
- Mixed DLL Loading Problem - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vcconMixedDLLLoadingProblem.asp.

Forward declaration problem

The Quake2 executable contains forward declarations to structures that are defined in other DLLs. The Visual C++ compiler fails to emit the necessary metadata for these structures and a **System.TypeLoadException** is thrown at runtime indicating that the structure could not be found in the assembly.

This occurs for the **image_s** and **model_s** structures and can be fixed by defining the structs in the main executable assembly. More information on this can be found at <http://www.winterdom.com/mcppfaq/archives/000262.html>.

```
// in cl_parse.c

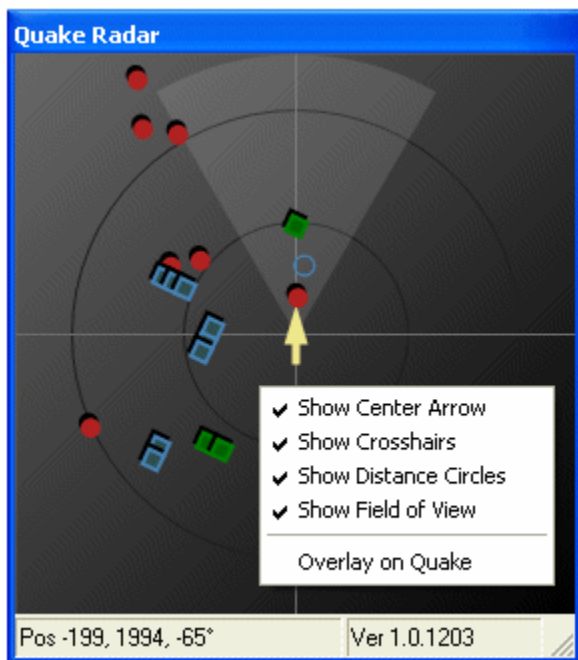
// empty definitions for structs that are forward declared
// this causes the compiler to emit the proper metadata
// and not throw a System.TypeLoadException exception
struct image_s {};
```

```
struct model_s {};
```

Extending Quake

Now that we had Quake II running inside in the .NET run-time environment, we wanted to add a significant new feature, written solely in .NET. After looking at the games we play today (namely Halo) we settled on a heads-up radar display that shows enemies, power-ups and other interesting objects in birds-eye view.

The radar extension was created in managed C++. Since this is a managed class, GDI features of the .NET Framework are used, including arrow caps, gradient brushes, antialiasing, and window transparency and opacity. Radar items are rotated around the center of the window using `Matrix.RotateAt` instead of calculating each item's position with trig functions. A context menu allows visual items, crosshair and field of view for example, to be shown or hidden.



The last menu item (Overlay on Quake) overlays the radar on top of the Quake window; the radar hides the window frame and status bar, sets window transparency and opacity, and resizes itself to fit over the Quake window.



Radar items are stored in an STL **vector** list. The code snippet below shows how an **iterator** is used to loop through the list and draw each item on the radar.

```
// draw each item in the list
ItemVector::iterator i;
for (i = m_items->begin(); i != m_items->end(); i++)
{
    // calculate location on radar
    rc.X = (int)center.X +
        ((*i).x/Const::Scale) - (Const::MonsterSize/2);

    rc.Y = (int)center.Y -
        ((*i).y/Const::Scale) - (Const::MonsterSize/2);

    switch ((*i).type)
    {
        case RadarTypeHealth:
            g->FillRectangle(Brushes::Green, rc);
            break;

        case RadarTypeMonster:
            g->FillEllipse(Brushes::Firebrick, rc);
            break;

        . . .
    }
}
```


A couple of build issues were encountered when using the STL **vector** class in the extension:

Compiler error C3633

The first issue was a compiler error when an **std::vector** class member was added to the managed class.

```
private __gc class RadarForm : public System::Windows::Forms::Form
{
    . . .
private:
    std::vector<RadarItem> m_items;
    . . .
};
```

```
\quake2-3.21\Radar\RadarForm.h(92): error C3633: cannot define
'm_items' as a member of managed 'Radar::RadarForm'
```

This error indicates that you cannot define **m_items** as a member of the managed class **RadarForm** because **std::vector** contains a copy constructor. This was solved by using a pointer for the vector list.

```
private __gc class RadarForm : public System::Windows::Forms::Form
{
    . . .
private:
    std::vector<RadarItem>* m_items;
    . . .
};
```

Compiler error C3377 and C3635

The next issue involved passing an unmanaged type to the extension. The Quake code passes a **std::vector** pointer to the extension to update the radar but this created the following compiler errors.

```
// update method in the radar extension class
static void Update(int x, int y, float angle,
    std::vector<RadarItem>* items)
{
    . . .
}

\quake2-3.21\client\cl_ents.c(1612): error C3377:
'Radar::Console::Update' : cannot import method - a parameter type or
the return type is inaccessible

\quake2-3.21\client\cl_main.c(305): error C3635:
'std::vector<RadarItem,std::allocator<RadarItem> >': undefined native
type used in 'Radar::Console'; imported native types must be defined in
the importing source code
```

This was fixed by defining an empty class that derives from **std::vector** as shown below.

```
_nogc class ItemVector : public std::vector<RadarItem>
{
};

// pass an ItemVector instead of std::vector
static void Update(int x, int y, float angle,
    ItemVector* items)
{
}
```

Integrating with Quake

There are three integration points with the Quake code: displaying the radar, updating the radar, and notifying the radar when the window position has changed.

Displaying the radar

A new command was added to the Quake vocabulary called **radar**. The following code toggles the visible state of the radar when the radar command is entered in the Quake command window.

```
// check for our new radar command
if (Q_stricmp(cmd, "radar") == 0)
{
    // toggle the visible state of the radar
    cl_radarvisible = !cl_radarvisible;
    Radar::Console::Display(cl_radarvisible, cl_hwnd);
    return;
}
```

Updating the radar

The radar is updated after a certain interval expires (500 ms). The code below constructs an STL **vector** list with radar items and passes the list to the extension.

```
void UpdateRadar(frame_t *frame)
{
    // see if enough time has elapsed to update the radar
    static int oldTime;
    int newTime = timeGetTime();
    if (newTime - oldTime < UPDATE_RADAR_MS)
        return;

    // update time so can detect next interval
    oldTime = newTime;

    // store radar items in an STL vector list
    ItemVector* items = new ItemVector();
    RadarItem item;

    // get the players info
```

```

int playernum = cl.playernum+1;
entity_state_t* player = &cl_entities[playernum].current;

// loop through list and add items to the radar list
entity_state_t* s;
int pnum, num;
for (pnum = 0 ; pnum<frame->num_entities ; pnum++)
{
    // get item entity_state
    num = (frame->parse_entities + pnum)&(MAX_PARSE_ENTITIES-1);
    s = &cl_parse_entities[num];

    // make sure this is not the player
    if (s->number != player->number)
    {
        // add item to the radar list
        item.x = s->origin[0] - player->origin[0];
        item.y = s->origin[1] - player->origin[1];
        item.type = GetRadarType(s);
        items->push_back(item);
    }
}

// pass to the radar extension so it can update the display
Radar::Console::Update(
    player->origin[0], player->origin[1],
    player->angles[1], items);

// clean up list
delete items;
}

```

Window position changed

The radar needs to know if the Quake window position or size has changed when it is displayed in overlay mode. The Quake code processes the WM_WINDOWPOSCHANGED message and passes the event to the radar extension.

```

case WM_WINDOWPOSCHANGED:
    // pass along to the radar
    Radar::Console::WindowPosChanged(hWnd);
    return DefWindowProc (hWnd, uMsg, wParam, lParam);

```

The **_MANAGED** macro

The Visual Studio C++ compiler contains the Microsoft-specific predefined macro **_MANAGED**. This macro is set to 1 when the **/clr** switch is specified and was used to wrap managed-specific code.

```

// setting the title of the window
#ifdef _MANAGED
    "Quake II (managed)",
#else
    "Quake II (native)",
#endif

```

Performance

Getting existing projects into managed code is useful since it offers a lot of design freedom, for example:

- Use garbage collection or manage memory yourself.
- Use .NET Framework methods or Window API calls directly.
- Use .NET Framework classes or existing libraries (STL for example).

However, usefulness only matters if the managed application has the performance you require. Running Quake II.NET in the timedemo test indicates the managed version performs about 85% as fast as the native version. The performance of the managed version was acceptable and testers did not notice a difference between the two versions. You can run the timedemo test by doing the following:

1. Display the command window by pressing the **tilde (~)** key.
2. Enter **disconnect** if currently playing a game. This is not necessary if Quake is in demo mode.
3. Enter **timedemo 1** and press enter.
4. Press the **tilde (~)** key again to close the command window. Quake runs through the demo measuring the frame rate.
5. Press the **tilde (~)** key to stop the test. The frame rate is displayed in the command window.
6. Enter **timedemo 0** to turn off the test.

Summary

Porting the C code to native C++ took about 4 days, and porting to managed C++ took another day. The extension took about two days to implement, as did poking around the Quake code to figure out the integration points. The experience overall was very good and we felt productive porting and extending the code. It's nice to mix native and managed code, to have control over memory management, and to use existing libraries as well as .NET Framework classes, all in the same application.