Game Programming Genesis
Part X : Tips and Tricks
by Joseph "Ironblayde" Farrell

Introduction

Here we are, with nine articles behind us, and only one left to go. It's unfortunate that I won't have time to take this any further, since there a lot more things I could go over, but we've gotten off to a good start in Windows game programming, and there are any number of places you can go from here: input devices, sound effects and music, scripting, the DirectX Graphics API, etc. There are lots of articles covering things like this to be found at GameDev.Net, or there's always the DirectX documentation.

In any case, to close the series out, I'm going to show you a few little things you can use while developing your games. A lot of these, like organizing large programs logically into multiple source files, are going to be necessities when you start building up a full game. For most of today's article you won't actually need knowledge of DirectX to follow along, so don't worry if you've missed previous articles in the series. All set? Let's start with the topic I just referred to: organizing programs.

Organizing Projects

All of the demo programs I've shown you along the way in this series have been relatively short. The source file for the most recent one, on adding characters to your game, was about a thousand lines long, but that includes comments and whitespace, and I use both quite a bit. Even so, when you start putting a full game together, you'll quickly find that putting all your code in a single source file just won't work. It's not very organized that way. Sure, you can use Visual C++ to search for functions pretty quickly, but it's still much better to have your program logically broken up so you can find things when you need them. Also, it's nice to be able to jump from one function to another just by switching documents, or to compare code from two documents side by side.

Organizing a program in C++ is easy: if each class has its own source and header files, you'll always be able to find what you're looking for. For C programs it's not so well-defined, so you should just do your best to group functions which serve similar purposes together. To give you an example, I'll show you what my source files for Terran look like. There are quite a few, but it makes it easy for me to know where everything is.

`audio.cpp`	This file contains all functions needed for loading and playing sound effects and music.
`battle.cpp`	These functions handle the details of the battle system, including the code that makes battle decisions for enemies.
`chars.cpp`	This contains everything pertaining to character handling, such as movement and animation, keeping NPCs sorted, and locating characters on the map.
`directx.cpp`	All the code that actually makes changes to DirectX interfaces is here, mostly for initialization and shutdown, but also for things like restoring and reloading lost surfaces.
`game.cpp`	The very general framework for each major game state is contained here.
`graphics.cpp`	Any function which renders graphics to a surface, be it fading or fireballs, maps or text boxes, is found here.
`input.cpp`	This file detects and reads input devices, and combines the relevant information into a small input structure used by the rest of the game.
`items.cpp`	Item functions such as buying, selling, using, transferring, or discarding items are here, along with anything else needed to manage character inventories.
`magic.cpp`	Spells are managed from here, both the learning and the casting of them. Graphical output is done in `graphics.cpp`, but this file is responsible for setting up those effects.
`maps.cpp`	This wraps up anything map-related: loading maps and collision data, performing collision detection, and launching scripts linked to the map.
`menus.cpp`	This file allows the user to navigate through any menu in the game, and sends commands to the rest of the game as necessary.
`scripts.cpp`	Here is where the scripting language extensions are set up, and specialized script functions are located. The general script loader and parser are in a library I wrote; I'll get to that later.
`stats.cpp`	These functions manage character statistics: computing those that are constantly changing, updating main stats at level increases, and so on.
`stdafx.cpp`	This is the precompiled header Visual C++ sets up for you when you create a simple project.
`terran.cpp`	Probably the simplest one, this one just has variable declarations, `WinMain()` and `WindowProc()`, plus some functions for loading game data which are called only once, at startup.
`terranrs.rc`	Terran's resource script, it contains a few icons, several audio files, and an enormous string table.

`text.cpp`	Anything text-related, such as generating lists or loading dialogue, is done here.

So hopefully that will give you an idea how to keep source files organized. I also have a large header file called `stdafx.h`, which contains declarations for a bunch of structure types, function prototypes, `extern` statements for global variables, and about five hundred million `#define` statements. If you haven't seen it before, the `extern` keyword allows other source files to access a global variable that's been declared in a different source file. It's just a qualifier that goes on the front of a variable declaration, like this:

```
extern int bImageLock;
extern STATE sGameState;
```

By including these extern statements in a header file, you can just include the header file at the top of every source file, and you'll have access to all your globals. The other thing to make sure you do with header files is to avoid including things like type declarations multiple times. For this, you use the `#if` and `#endif` directives. The way it's usually done is to check if a certain constant has been defined. If not, define it, then include all the stuff you need. That way, the next time that block of code is encountered, the constant has already been defined, so your declarations are only included once. For example:

```
#if !defined(_GAME_HEADER_001_INCLUDED)
#define _GAME_HEADER_001_INCLUDED

// ... constant declarations and such go here...

#endif
```

There are a lot of blocks like this inside the main Windows headers, which is why using `#define WIN32_LEAN_AND_MEAN` can keep a lot of unnecessary code out of your project builds. Most things you'll put in header files should only be included once, and so should be inside a `#if` block. But including external variables with `extern` should be included in every source file, so be sure to keep those statements outside of your `#if` block.

One last thing on this: if you're using Visual C++, you can also use a single line of code that reads `#pragma once` to accomplish the same thing. This line specifies that the header in which it is located should only be included one time in a build. Most `#pragma` directives are compiler-specific, though, so you'll need to use `#if`..`#endif` if you're not using Visual C++, or if you may work with your code on a different compiler one day.

Creating Libraries

When you've got a fairly lengthy bit of code that gets used pretty often, it's often convenient to build a library out of it rather than having to cut and paste code every time you want to use it. A good example of this is initialization code for DirectX. Do you really want to go through all that clip list crap every time you want to create a DirectDraw clipper? Me neither. So I wrote a large set of initialization functions, and dropped them all into a library that gets used in just about every DirectX program I write. Being able to set up a surface, a clipper, a game controller, a DirectMusic interface, etc. in a single function call is pretty convenient.

Building a static library is very easy. When you go to create a project in Visual C++, just select "Win32 Static Library." The only difference between creating a static library and creating any other Windows program is that a library is just a collection of functions. You don't execute it by itself, so it doesn't need a `WinMain()` or anything like that. Just write as many functions as you want in as many source files as you like, and they compile into a single `.lib` file you can include in your projects. The other thing you'll need to write is a header file for the library that contains things like constants or data types used in the library.

Terran uses two libraries of my own in addition to the standard DirectX libraries. The first one, `adxl.lib`, is my general-purpose library. It contains all kinds of nice functions for Win32 and DirectX, as well as loaders for different image and audio file formats. The other, `aeonscript.lib`, is my general scripting engine. It contains the code for loading and parsing scripts that contain general functionality like variable assignments and equations, `if` statements, loops, image loads, calling other scripts, and so on. Specialized functions for things like moving NPCs can be easily added to the scripting engine by any program that uses this library.

Re-using code will save you a lot of time when you get into large projects, so you might as well start now! Take a look at the programs you've been writing and see what kind of things you're using over and over. Those functions might serve you better in a library. Anyway, let's move away from organization now and take a look at some things you can add to the program code itself.

Runtime Log Files

How many times have you fired up your game program, anxious to see how your newest additions are working out, only to see a black screen or have the program crash on you? If you said "never," you're either a god or a liar. And since gods have no need to read these articles, that makes you a liar. :) Tracking down logic errors is no fun, so you need anything you can get that will help you locate problems. The Visual C++ debugger is a great tool for this, and log files are another. A log file is simply a text file generated by a program while it's running that tells the results of various function calls, the states of variables at certain points, or anything else you care to throw in there. Here's an example, a little snippet of Terran's log file:

```
DIRECTDRAW
DirectDraw interface created.
Fullscreen cooperation level set.
   --Message Received: WM_DISPLAYCHANGE
Resolution 640x480x16 set.
Surfaces created successfully in system memory.
Pixel format is 5.6.5 (16-bit).
DirectDraw clipper created.
Clipper attached to back buffer.
```

This, obviously, is one of the blocks of text generated during initialization of the game. Something like this can help you see exactly which function call is failing, or what setting is coming out not as you'd expect. When I first released a demo of Terran, some people said that the fading wasn't working properly. Thankfully they send back this log file, and I was able to see immediately that it would always fail when the pixel format was 5.5.5, so I knew right where to look for the bug. Another thing you can use log files for is outputting information about the user's computer. If things are running slowly for someone, it's possible that their machine doesn't accelerate some feature of DirectDraw that your machine does. Things like that are easy to spot when you can look at a log file for reference.

There are two basic ways to create a log file. One is to just open a file at the beginning of the program, write all your information with `sprintf()` statements during the game's run, and close the file at the end. The other way is to write a function that does that every time you want to log something. I prefer the latter, because if the program crashes, it's better not to have left the file open. That approach also makes it possible to easily enable or disable logging while the program is running, or add extra features instead of just a straight write. Here's an example of the logging function that I use:

```
BOOL ADXL_LogText(char *lpszText, ...)
{
  // only do this if logging is enabled
  if (bLogEnabled)
  {
    va_list argList;
    FILE *pFile;

    // initialize variable argument list
    va_start(argList, lpszText);

    // open the log file for append
    if ((pFile = fopen("log.txt", "a+")) == NULL)
    return(FALSE);

    // write the text and a newline
    vfprintf(pFile, lpszText, argList);
```

```
        putc('\n', pFile);

        // close the file
        fclose(pFile);
        va_end(argList);
      }

      // return success
      return(TRUE);
    }
```

Chances are you haven't written a function with an arbitrary number of parameters before, so I'll explain briefly. The elipsis (...) in the function header says that after lpszText, any number of additional arguments may follow, of any data type. To handle something like that, you need the va_list data type, and the va_start and va_end macros. The "va" presumably stands for "variable arguments." Anyway, the va_list type is just a pointer to a list of arguments. You need to create one in any function that receives a variable number of arguments. The va_start macro initializes a list of type va_list, and takes the list, and the previous argument as parameters. The va_end macro simply resets the argument list pointer to NULL.

Finally, the other bit of syntax to notice in the log function is the use of the vfprintf() function, which is the version of fprintf() that is designed for use with variable argument lists. There are also vsprintf() and vprintf() functions that function similarly if you ever need them.

The variable bLogEnabled that you see near the top is set by the other logging function I use, which simply enables the log file. It's a pretty straightforward one:

```
    int ADXL_EnableLog()
    {
      FILE* pFile;

      // enable log
      bLogEnabled = TRUE;

      // clear the file contents
      if ((pFile = fopen("log.txt", "wb")) == NULL)
        return(FALSE);

      // close it up and return success
      fclose(pFile);
      return(TRUE);
    }
```

The `bLogEnabled` variable is just a global variable within my ADXL library, where these functions are coming from. The variable initially is set to `FALSE`, so that nothing gets logged until after you call `ADXL_EnableLog()`. It's nice to have it set up this way, because then if you want to distribute a version of your program that doesn't generate a huge log file, you can just remove one line, instead of searching through your code to remove all the calls to the logging function.

Protecting Image Data

Suppose you've got a great new game project going, but it needs a ton of images -- say 15MB worth. How can you make it so that the end user can't change your images to whatever they want? You could include all your images as resources... but then your `.EXE` would be enormous, and wasting a lot of memory space. The only other option is to include them as external files, but then people can just open them up in any image editor and change them, right? Well, there are a few things you can do with this.

The first one is obvious: make up an image file format. This can be a good bit of work, though, since you have to come up with a fully descriptive header that won't be so easy for people to figure out at a glance, then decide on some method of filtering and compression to store the image data. There are plenty of compression libraries out there that you can use, like `zlib`, but there's one other way to go.

The other thing you can do is to either expand an existing file format, or assign meaning to some bytes in an image header that are either reserved or not currently used. As a couple examples, the `.BMP` file header has two reserved fields that must be set to zero for a standard bitmap, and the `.PNG` file structure is based on "chunks" and is expandable, so you can add your own fields. Now, what good is this? The idea is to use these extra storage locations to hold some sort of image "key" which is calculated in a manner known only to you, based on the image data. Ideally, you should have it set up so that if the image data changes, the key will also change. That way, when you load an image from your game, you can look at the image data and calculate the key value. If the calculated key is different from the one stored in the image file, you know the user's been tampering with your image, and you can spit out an angry error message.

The only question is, how do you decide on a method for generating a key? Let's try a few things. How about using the width plus the height of the image? That's no good, because if people were to change the image, chances are that they would not alter the dimensions, only the contents. So, how about the image file size in bytes? That's OK for some things, but for uncompressed file formats like `.BMP`, the file size won't change unless the color depth or image dimensions change, neither of which is likely. A better idea would be to do something like this. While you're displaying the image, you have to go through every pixel anyway... so while you're doing that, you might as well add up the total values for red, green, and blue as you go. Or just one of the three would be OK.

Let's say you do it for green. At the end, you'll have a very large number which is the combined intensity for green from every pixel in the image. Now just because you can, raise that value to the 10/9 power, multiply by 3, and add 1331. Why? Because nobody would guess to do that particular operation.

Key choices like that are much better because they will almost certainly be affected no matter what the user does to the image. There are a few things to watch out for, though. For instance, in my last example, a very large image might give you an overflow when you calculated the key. You can combat that by making sure the original key never gets too high. For example, as you're adding up the green values, you can do the addition modulo 2^{16} so that when you're done, you have a lot of room to work with. If you're working with an integer key, then you need to make sure you don't narrow the range too much. For example, if your key involves something like taking a fifth root of your initial calculation, then a very high key value can change quite a bit, and still have the fifth root evaluate the same when you're working with integers. Floating-point keys don't have that fault.

Other possible choices for keys would be things like how much each pixel differs from the one immediately to the right of it, in terms of any one of the color channels. Calculate all those values and add them up. For compressed image file formats like `.PNG`, you have even more choices. You can look at how each row compresses, and form a key based on the ratio of the compression of each row to the row beneath it. In any case, you can simply store the key within the image header, or make up a new section of the image file in which to store data. You can even generate multiple keys to make it that much harder to figure out. Users will probably be able to see your image files if you take this approach, since you're still using a standard image file format, but it's not easy to change them without the game program realizing it.

If you're interested in working with other image file formats, check out wotsit.org for as much information as you could ever want on more file formats than you ever knew existed. :) I'd highly recommend looking into using `.PNG` for games. It supports high-color modes, unlike `.GIF`, and uses a lossless compression algorithm, unlike `.JPG`. In fact, `.PNG`'s compression is so good that for artistic-type images (as opposed to photorealistic ones), `.PNG` will often compress as good as, if not a good deal better than `.JPG` will. The `.JPG` format wins hands-down for photorealistic images, but that's generally not what you'll be using for games, at least not at this point, so it's better to have something using lossless compression.

Introduction to Scripting

I've talked about scripting engines so much in this series and yet never got around to explaining how exactly it works. Plus I've gotten a lot of E-mails from people wanting to know this stuff, so I thought I'd give you a bit of an overview here. Obviously I can't come anywhere close to completely covering this since this is only a small part of a single article, but hopefully it will give you an idea for a framework you can build on.

To implement a scripting engine in your game, you'll need to do three things. First, you have to design the language. Second, you have to write a "compiler" that turns text scripts into some sort of code, so that users can't look at your scripts and know how to change them to do what they want. Third, you have to write the interpreter that loads and executes script files.

The first part isn't too difficult since you've already seen at least one programming language -- probably several -- so you have a good idea as to what you need in a scripting language. You need variables, decision statements, loops, the ability to read mathematical expressions, and functions. There are two ways to go as far as functions are concerned. You can build functions like MoveNPC into the scripting engine itself, or you can allow the scripting language to directly call functions which are in your game code. The first approach is easier, but not as flexible since you're adding functions to a scripting language that really only apply to one game. The latter approach is much better, because it allows you to use a general scripting engine in a variety of different programs. In addition to those, there are probably some general functions you want to add directly to the scripting engine that can be used from any program. Image loaders are a good example of that.

Once you've decided on what your language needs, and what the syntax is going to be, you're ready to write the compiler, which is easily the most difficult part of doing this. (The interpreter is actually quite straightforward.) What I do is to break all my script code down into fixed-length instructions, much like you would see in an assembly language. `If` statements and loops are replaced by branch instructions. Mathematical equations are broken down into single operations. For example, the equation `x = (a + b) * (c + d) / (e - f)` is broken into this:

```
    add t0, a, b
    add t1, c, d
    mul t0, t0, t1
    sub t1, e, f
    div x, t0, t1
```

Those simple instructions are then turned into bytecode. That seems simple at first, since all you basically need to do is pick a number that corresponds to the opcode (`add`, for example, is an opcode), and replace the word by the number, right? But there are other concerns. In the bytecode, how do you tell whether the arguments are variables or constants? How do you represent floating-point numbers? And just how do you translate a mathematical formula into fixed-length instructions in the first place? Things like this are all questions that need to be addressed before you can have a working compiler, and they all take a bit more time to explain than I have in this article.

The final part of the scripting engine, the interpreter, is relatively easy to write. After all, your compiler has already done the hard work of breaking down complex instructions into small, simple components. The interpreter's job is to open a script file, find out how many lines it contains, and then create some storage space for the script, either with an array or a linked list, and load the script into memory. Once that's done, each line in the script consists of an opcode number and a series of arguments. An easy way to execute the line is to use the opcode number as an index into a function table. You then have one function for executing each opcode. The interpreter just needs to be set up to use that function table to call the appropriate function, and then you can go ahead and build the rest of the interpreter, the part that actually acts on the commands, one little piece at a time. Since most of those functions will be things like assignments, arithmetic, and branches, it's really no problem. Things like moving NPCs are of course not trivial, but those type of actions are part of your game program, not the scripting engine. The scripting engine is only responsible for calling them.

For allowing your scripts to call functions already in your game program, you could give your scripting engine a function called `AddFunction()`, or something similar, which simply takes a function pointer and adds it to its opcode list. You'd also have to make a small data file with these assignments for the interpreter to read, so it knows how to translate those functions, but that's not a big deal either. I know this overview was very general and doesn't give you any code towards actually getting something like this on its feet, but hopefully it gives you some ideas.

The String Table

Remember way back in article 2, when we looked at resources, and I told you that you could use a string table to represent all kinds of data in a game? Well I thought I'd give you an idea as to what use you can make of a string table in an RPG. RPGs especially benefit from something like this, since most of them end up having lots of text.

The first and most obvious use for a string table is to store narration and/or character dialogue. Those two things together will make up a very large part of the text needed in an RPG, and so it's important to have an organized way to store it. You could put it in a file -- but then anyone playing your RPG would be able to change the dialogue however they wanted. Your beautiful ending scene could become a recreation of "All your base are belong to us," and we can't have that, can we? :) You could store the strings directly in your scripts somehow, but then you'd have to worry about how to encode them, and besides, it would break our plan of having short, fixed-length instructions. So what's left? Use the string table.

Storing dialogue this way helps your scripts in another way, too: it gives you a very easy format for creating a function that displays text on the screen. For example, look at the following script command:

```
    showText 10, 10, 253, 5
```

This could be translated as "At screen position (10, 10), show five lines of text, beginning with line 253 in the string table." The only problem here is that you're going to have a lot of dialogue, but you only have one string table, so it's easy to get lost in there. There are a couple things you can do to get around that. The first is pretty easy: just group your text into logical sections and use a few `#define` statements to define where the different sections are located, so you can get to them easily. You might have something like:

```
    #define ST_OPENINGSCENETEXT  35
    #define ST_FIRSTTOWNNPCS     253
    #define ST_SECONDTOWNNPCS    428
```

But if you have a really big project going, that's not going to be enough. In that case, what you might want to do is to write a utility that lets you add text to all sorts of little partitions that you can create, so it gives the effect of having many string tables, but then the program combines them all into one and writes the resource script for you. This can be very convenient, and it's quite easy to do. All you need is a way to create divisions of text, almost like a directory tree on your computer, and a way for the program to tell you what string table index it assigns to the text you write down, so that you don't even have to look at the string table to easily access any dialogue in the game.

Lists of items, magic spells, menu options, etc. are also good candidates for the string table, since you can just define a constant and then use offsets to locate the relevant data. For example, in Terran I have a constant called `ST_ITEMDESCRIPTIONS` which is an index into the string table. When I want to pull up a description for an item, I just load string number `(ST_ITEMDESCRIPTIONS + nItemNumber)`.

The last thing you might want to use a string table for is filenames, be they for image files, map files, scripts, or anything else. That way you can have all your filenames together, so they are easy to find and change if you need to do so. You can again use numeric constants to represent the filenames by acting as indices into the string table. When my game needs to load the script for initializing a game, it calls the script loading function with a value of `SCRID_GAMEINIT`, which of course just tells the loader where it can find the filename.

Closing

And so it ends. Congratulate yourself if you've been with me since the beginning of the series; we've come from displaying a blank window on a screen to being just about ready to take on most of the features of a simple, albeit complete DirectX-based RPG, or whatever other type of games you're interested in writing. I have to quit writing for awhile, but if you want to get more info on any part of game development instead of striking out on your own, GameDev.Net has all the resources you need.

As for the future, I'm thinking about getting a series on scripting going once I have more time (read: in May when classes are done with). Some people have even suggested writing a book based on this general outline. At first I didn't know about that, but it's starting to sound like an interesting idea. If I do something like that, it will of course be much more detailed, and cover a myriad things I didn't get to here, like scripting engines, responsive NPCs, tips for other game genres like basic shooters, and using DirectX 8 for graphics so you can get hardware acceleration for things like alpha-blending. If you'd be interested in seeing something like that, let me know, so I have an idea as to what people would think of it.

Thanks to the many, many people who sent in positive feedback on the series; this has gone over even better than I had hoped for! If you still have questions about anything in these articles, as always, feel free to E-mail me at ironblayde@aeon-software.com. I'm on ICQ less and less these days, so E-mail is your best bet. Well, happy coding, everyone, and I'll see you later.

Discuss this article in the forums