Game Programming Genesis
Part VII : Developing the Game Structure
by Joseph "Ironblayde" Farrell

Introduction

I know, I know, this was supposed to be about writing a tile engine, but after I finished that last article and started thinking about how best to approach the series from here on, I decided that wasn't a good idea. Let me tell you why.

Several times I have either alluded to or actually talked about the fact that you can't structure your game code the same way in Windows as you would in DOS. Everything has to be more organized because the entire main loop must execute at least once per frame, so you don't lose track of the messages that Windows is sending to your application.

Now, you've seen a little bit of this, in the short demo programs I've provided with the previous articles. But you probably haven't already tried to extend this method of programming to something large, unless you've already been writing Windows games based on the previous articles in the series. So, I came to the conclusion that before starting to introduce you to the workings of all the parts of a game, it would be best to devote an article to examining the structure of a game, to see how all those components will eventually fit together.

This article is a bit different from the others in that we won't be looking at much code, if any. So if you haven't read up on DirectDraw, or if some things are a bit unclear, don't worry, because we're taking a break from it. :) All you need is a basic knowledge of how a Windows program works, so the first article or two will suffice.

One other thing that I'll mention here: for this article, and most of the subsequent articles, I'm going to be referring to Terran a lot as an example, so you may want to download the demo so you can see the working implementation of what I'm describing. As of the time of this writing, the demo may not run with all video cards, but once Demo 2 comes out, this will hopefully be resolved. All right then. Let's get to it!

Overview

Above all, before you start programming, you should have a detailed design planned out of exactly how the logic in your program is going to operate. There is no bigger mistake you can make than to just sit down and start coding right away. It will all seem clear and organized at first, but soon you'll start to realize how many things you didn't think of, and

as you start adding them to your code in whatever place you can find for them, your program will quickly become disorganized and inefficient. Trust me; that's why I started over on Terran. :)

You should start with the `WinMain()` function, as that's where the program begins. This may seem obvious to you, but a lot of people start off by writing fading routines, or collision detection functions, and all the rest of the specific, detailed stuff, before writing the main functions that bring them all together. This is a little backwards. In an ideal top-down design, you should write the main function, followed by every function called by that main function, and so on. This is a good way to do things for at least two reasons.

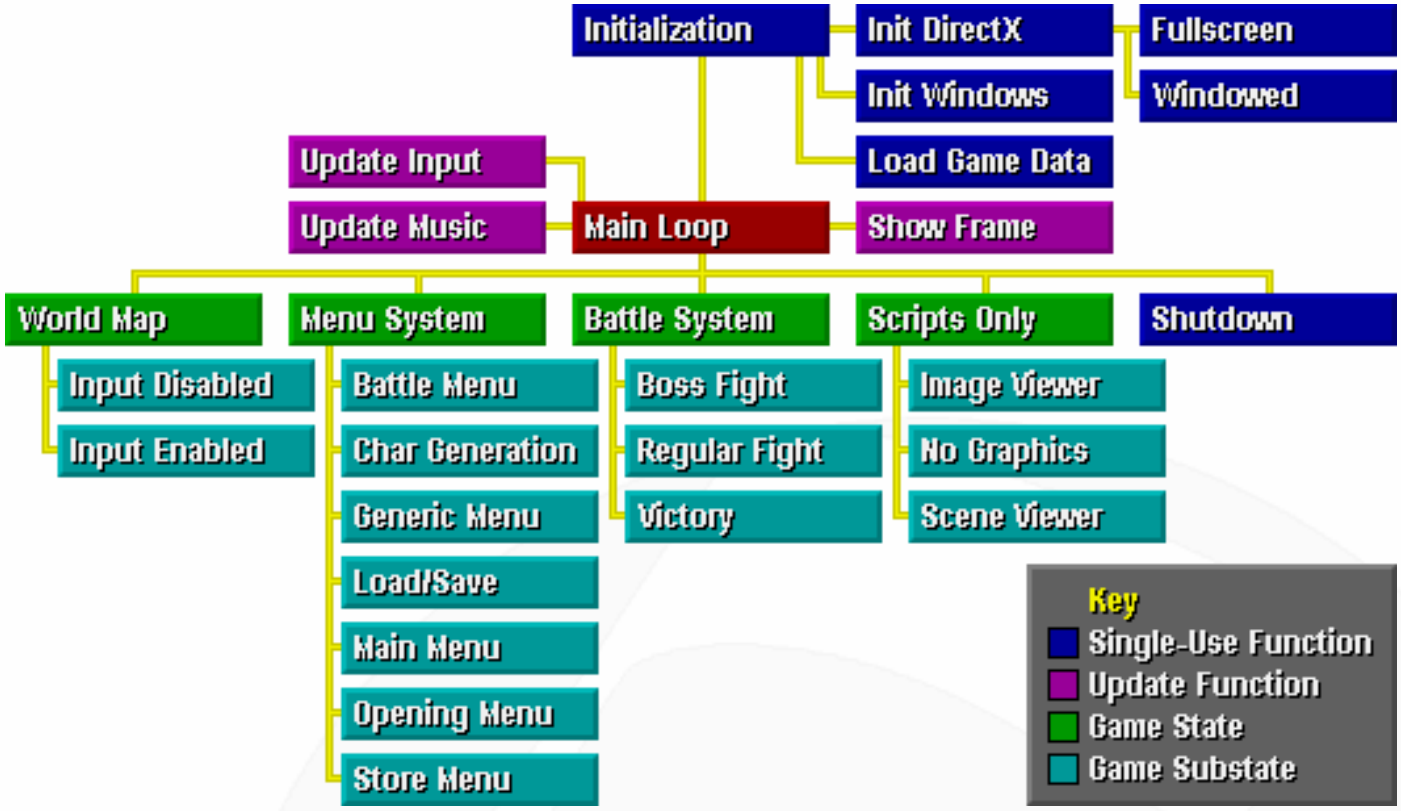
First, everything is functional right away. If you start your game project by writing all kinds of technical functions, you can't test and debug them without creating some temporary code that sets up a scenario in which they would be used. On the other hand, if you design your game from the top down, the idea is that you should be able to run it at any time, which is essential because you'll probably be wanting to test your game constantly as you go. At each stage of development, more and more of the details are added, and the game starts to take shape.

Second, the overall structure of the game is apparent from the beginning. When you write the details first, you may find later on that you're making your more general functions messy in order to call the details in the correct way. In essence, the details would be dictating the structure of your program, whereas the structure of your program should be defining how the details should be implemented. So the better approach is to go from general to specific. If you have your general program structure up and running by the time you start developing the details, you have several advantages. You know exactly how that last level of functions needs to be written in order to work with your design, and you don't have to write test cases for your functions, because they're already in place!

What I have done with Terran is to divide the main program into five sections, and then proceeded with a top-down design on each section. Four of the sections are the actual game content: the scrolling engine, the scripting engine, the battle engine, and the menu system. The fifth component encompasses all of the things that need to be done with Windows and DirectX to make things run. This includes initialization, shutdown, and things like changing from fullscreen to windowed mode. Each of the four game components has multiple substates which further break down the roles of each function. What I'm going to do with this article is to cover them all in detail, so you can see how everything is organized. Hopefully it will give you an idea as to what you'd like to do with your game, and make it a bit easier to understand how to implement specific game components when I start going through them in the next article.

Example Implementation: Terran

Before anything else, let me show you a little map of the program flow in Terran, to illustrate what I'm talking about:



This diagram by no means represents all of the functions in the game, but it shows the basic flow of things. Let's work through the whole thing, one step at a time.

Execution starts at the `Initialization` element of the map, which is basically the first call made from `WinMain()`. It's broken down into many more stages than are shown, but if I showed every function in Terran on this map, it would be impossible to read. :) You should be pretty familiar with the Windows and DirectX components of this function -- they create a window, and all the required DirectX interfaces. There are a lot of interfaces we haven't covered yet, but that's not important for this article. We'll get to them as we go. The `Load Game Data` element loads all sorts of information from external files. This includes data on the items available in the game, data the scripting engine needs to operate, magic system information, etc.

The last thing done in initialization is to load a script which tells the program where to go from there. Almost every aspect of Terran is driven by the scripting engine, as I'll talk about in detail later. In fact, there will be an article in the future devoted to creating a very basic, albeit very useful scripting engine for your own games.

After initialization is complete, the program enters the main loop, where it stays until the user's had enough and shuts the game down. Each iteration of the main loop takes four basic steps that are shown on the diagram.

First, the input devices are updated. There are two input devices in Terran, the keyboard and the joystick. This function takes the current state from each device and logically `OR`s

them together into one master input table which has two columns. Column 1 tells whether or not the button or key is currently up or down, and column 2 tells whether or not the button has been pressed just that instant. Column 1 is used for continuous input, like walking around on the map, whereas column 2 is used for discrete input, like making menu selections.

Step two of the main loop is to update the music state table. Basically all this does is checks through all the game songs to see if any of them are flagged as playing. If they are, the game checks to make sure they're really still going, or if they've stopped since last time the main loop ran. If they've stopped, the function sets their flags to reflect this. Don't worry about the details of this right now -- implementing music is a topic for much later.

The third step is the most important one: the main loop calls one of the game's five states. These are `World Map`, `Menu System`, `Battle System`, `Scripts Only`, and `Shutdown`. You get the basic idea of what each one does. Note from the diagram that `Shutdown` is a single-use function; once it gets called, that's it. Everything closes down. The other four require a bit of explanation.

Each of the game's states has a number of substates, which are shown in light blue on the diagram. Each state function (for example, `WorldMapMain()`) performs some tasks that are common to all the instances of that state, and then branches into actions specific to each substate. In some cases this is a simple `if` or `switch` statement, and in some cases it is a whole set of functions.

World Map: The world map state is very simple. First it runs any scripts that are currently loaded. Second, based on the substate, it either interprets the current user input or ignores it. Finally, all onscreen characters are updated, the map is drawn, and any effects currently loaded are performed. (More on implementing effects later.) That's it!

Menu System: The menu system is a bit more complex, because it has to handle everything from loading games to generating characters, from purchasing items to setting equipment. The main function sets the colors for the menu based on which options are active, then branches to the appropriate menu subfunction. This is done using an array of function pointers. The game substate is an integer that serves as an index into this array. The line looks like this:

```
(*lpfnMenuSubfunctions[substate.current])();
```

If that looks a little weird to you, you might want to read up a little on function pointers, as they can be rather useful. Each substate handles the menu-specific details like determining which options are valid and taking the final action if the user makes a choice in the final level of a menu. It also renders the current frame, since the method used to do this may vary depending on the menu. After that's taken care of, the main menu function runs the current scripts, applies any active effects, and plots any auxiliary text boxes, like one

showing the player's current gold, or a character's stats.

Battle System: Battle is a strange one because it's closely linked to the menu system; the battle system is semi-active, a la Final Fantasy, so whenever a menu pops up to control a character, the battle system transfers control over to the menu system. Thus, since both the menu system and the battle system must be able to handle the battle logic, nearly all of the logic is placed in another function, which can be called from either game state. The main battle function, then, is largely a placeholder. It does, however, handle the turn queue, which I'll get to in a second. The function that handles all the logic works something like this:

First, all active characters and enemies are checked to see if their turn has come up. If an enemy's turn comes up, the enemy AI takes over and makes a choice for that enemy. If a character's turn comes up, and the menu system is not currently active, then the menu system is activated and the player receives control of that character. If the menu system is already handling another character, then the character whose turn has come up is placed in a data structure called the turn queue. Once the main battle function is active rather than the menu system, it will check the turn queue. If it's not empty, the function dequeues the character who's been in there the longest and sends control back to the menu system.

In either case, once a character or enemy's action has been decided, the details of that action are placed into a second queue, called the action queue. Each time the battle logic comes up (from either the battle system or the menu system), one of two things happens. If there's an action currently underway, the script controlling that action is run. If there is no action currently happening, the game checks the action queue and sets up the next action, if there is one.

After all that is done, the background is copied, the characters and enemies are drawn, and any active effects are applied, at which point the battle logic is finished.

The difference between the substates for regular and boss fights is simply that you can't escape the fight -- you must finish it. This corresponds to a simple `if` statement. The third substate, `Victory`, is a bit different. It reports the gains of the battle. There's no additional logic needed for it, though, because it's all handled by a script, as usual.

Scripts Only: This last of the four main game states is dead simple: it runs any scripts that are loaded. That's it! This state is usually not encountered during play except duing initialization, but it can come up whenever the game isn't displaying anything and is doing some behind-the-scenes work.

Shutdown: This does exactly what you'd expect: it exits the main loop, releases all the DirectX objects that were created, deallocates any memory that's still being used, and shuts the program down.

The final step of the main loop is to display the current frame, which is simply copying the back buffer to the screen, using double-buffering if the game is in windowed mode, or page-flipping if the game is in fullscreen mode. I don't think we've covered page flipping yet, but it'll come up sooner or later. :)

A Word on Game States

The approach that I have just described, dividing your program up into several states, is a good one because what it does is to allow you to consider your program as a number of smaller programs. Instead of trying to develop the whole thing at once, you can just do one state at a time, which is simpler to write, and simpler to debug. All you have to do is change the game's initial script to set the active game state to whatever it is you're currently working on, and you can test it independently of the other aspects of the game!

My approach to Terran was obviously to write the scripting engine first, since it requires a script to initialize. Mainly I wrote up all the basic logical functions like decision statements, loops, and variables. After the scripting engine was on its feet, I went to the world map state. Once I had that all set up, and had tested it independently of the scripting engine, I wrote the menu system. After it was done, then I tested the game with all three elements integrated. This was not hard to do -- in fact, it required no C code on my part, because the scripting engine makes all the transfers of control. So basically all I did was write a simple script, and then sit back and watch it all happen. :)

Now I've come to the battle system. I actually haven't started coding it yet, but you wouldn't know that from the level of detail I already know the implementation in. I have all the logic diagrams and such drawn out, so I know exactly how everything will work. The only thing that remains is to actually write it, which takes awhile even if you know exactly what you're going for. But when I start doing this, I'll take out the game's current initialization script that sets up the whole game, and insert one that simply sets up a battle. In this way, I can develop the battle system as though the rest of the game wasn't even there, so I don't have to worry about it. Once the battle system is done, I'll just put the old initialization script back into place, and just like that, the new component is integrated.

My recommendation would definitely be to try setting up something like this for your game when you start it. As you can see, it makes the development process a lot nicer and results in fewer shouting matches between the programmer and Visual C++. Those are tiring, becausethe compiler always wins. It does against me, anyway. :)

Extended Events

By extended events, I mean anything that takes place over the span of multiple iterations of the main loop. This is a main difference between DOS and Windows programming. In DOS games, you probably wrote your fade so that the whole thing took place in one

iteration of the main loop. That is, you probably called a function called `FadeOut()` or something similar, which displayed 30 or so frames before returning. You know by now that Windows doesn't like you doing things like that, so you need to write effects that know how to update themselves each frame. Figuring out exactly how to do this can be a point of some confusion to new Windows programmers, so once again, I'm going to explain how I've done it in my game, to use it as an example. Hopefully it will give you some ideas as to how to pull off the same kinds of things.

First let's figure out exactly what's going to need to run over a number of frames. The first and most obvious category is graphical effects. This includes fades, animations, particle system displays, etc. Up next is text effects, which can include fading text onto or off of the screen, or simply displaying text over a number of frames. Third, we have things like waiting for a key before going any further. It's so simple you may not have thought of it, but you can't just sit in an empty loop until the user presses a key! You have to keep going! Finally, perhaps most importantly, are scripts. Most scripts will not execute in a single frame. You don't want them to, because they may be controlling things that show up on the screen. Terran's script for the opening scene, for instance, takes about six and a half minutes to run. At thirty frames per second, that's about 11,700 frames.

Scripts

All right, then... so how do we get a script to run over multiple frames? It's actually quite simple. When I wrote up the set of commands supported by my scripting engine, I separated them into two groups: terminating and non-terminating. The former contains commands which, once executed, cause the script to stop running until the next frame. The latter contains commands which can be executed all in the same frame; that is, they don't terminate the script for that frame. Thus, in any given iteration of the main loop, a script may execute any number of non-terminating commands, but only one terminating command.

Before going any further, let me show you the simple data structures I use to hold my scripts:

```
int nScripts[MAX_LINES][MAX_PARAMETERS + 1][MAX_SCRIPTS];
int nNextLine[MAX_SCRIPTS];
```

There are several constants used here that are `#define`d elsewhere in the program. `MAX_LINES` is the maximum number of lines that a script may contain. `MAX_PARAMETERS` is the maximum number of parameters that a script command can take. I've added one to it in the array declaration because there needs to be a column to hold the command itself, in addition to its parameters. `MAX_SCRIPTS` is the maximum number of scripts that can be executing simultaneously. There are implementations that use less memory than this, such as allocating memory as more scripts are needed at once, and using a variable-size memory location for each command, but it's really not necessary since this represents only

a small chunk of memory, unless you want to run fifty scripts at a time or something insane like that. In truth, you'll probably never need more than three. I have `MAX_SCRIPTS` set to five just in case.

The `nNextLine` array holds the index of the next line to be executed for each script. When a script is loaded, its corresponding entry in `nNextLine` is set to 0 to indicate the first line should be executed. If an entry in `nNextLine` is -1, this means no script is loaded in that slot. The entries in `nNextLine` are used to easily implement `if` statements, loops, and all sorts of other goodies.

Whenever the scripting engine is called to run anything that's currently loaded, it checks the script slots one at a time. If a script is loaded in that slot, and no effects are linked to it (more on this in a sec), then it executes commands from that script until a terminating command is encountered. That command is executed, and then the engine leaves the script and goes on to the next one. Pretty simple, hey?

So now you're probably wondering: if Terran's opening scene script takes 11,700 frames to execute, does that mean that there are 11,700 terminating commands in it!? Of course not! If you write a script that ridiculously long, I expect no less than a cure for cancer to come out of it. The idea behind the long execution time is that the script can be suspended. For example, take a look at this pseudo-script-code:

```
DoSomeAnimation
ShowText 12, 133
```

Suppose this corresponds to "Do some random animation, then display twelve lines of text, starting with line #133." Well, the animation is going to take time to finish! If you want to display the text after the animation is complete, you can't just execute the animation in one frame, and display the text in the next. This is where the idea of effects comes into play.

Effects

An effect as I define it for my game is anything you can see happening on the screen. Fades, particle systems, displaying text, etc. all fall into this category. I store an effect in a structure that looks like this:

```
typedef struct GAMEFX_type
{
  DWORD dwID;          // effect ID number
  DWORD dwFlags;       // control flags
  int nCurrent;        // for FOR loops
  int nTarget;         // ''
  int nStep;           // ''
```

```
    int nLinkedScript;  // ID of script this is linked to (or -1 for none)
    int x, y;           // a screen location
    void *buffer;       // optional data table
} GAMEFX, FAR* LPGAMEFX;
```

Let me run through the parameters and explain what I'm using all this stuff for.

DWORD dwID: This is the effect identifier. It takes a value from one of a number of FX_ constants I have defined, such as FX_FADE, FX_TEXTIN, or FX_PARTICLESYSTEM.

DWORD dwFlags: This can be used to store miscellaneous flags for each effect, and so its use varies from one effect to another.

int nCurrent, nTarget, nStep: These are used to define for-style loops, or incremental effects. For example, I have a script command called FadeDown. An example call would be:

```
FadeDown 34, 11
```

This means, "Fade down to 34% by increments of 11%." The way the GAMEFX structure is set up for this, nCurrent is set to the current fade percentage, whatever that happens to be, nTarget is set to 34, and nStep is set to -11. In each iteration of the main loop, nCurrent is decreased by 11, and that percentage of fade is applied to the frame. Once nCurrent is less than or equal to nTarget, the effect terminates.

int nLinkedScript: This is the important one. If this is -1, the effect executes side-by-side with any scripts that are running. But if this value is nonnegative, then the script with that index cannot continue execution until the effect finishes. This is important! Let's go back to our earlier example.

```
DoSomeAnimation
ShowText 12, 133
```

Suppose these commands were found in the script in slot #1. Then when the effect generated by DoSomeAnimation was set up, it would be linked to script #1, and thus the script would halt execution until the effect was done, at which point the text would be displayed, like we wanted! This adds one more step to our script handler: if the script is linked to any effects, don't execute it. Also note that any script command which will link the script to an effect must be terminal, so script execution stops right away.

int x, y: Many effects, like displaying text, require a location on the screen, so I provide one here. If you need multiple locations for an effect, well, that's what the final parameter is for...

`void *buffer`: Many effects can easily be specified by using only the above parameters. But what if you want to create a blizzard by sweeping 800 particles across the screen on increasing-amplitude sine waves? You don't want to be calculating that stuff as you go! The solution is to take this pointer, `malloc()` yourself some memory, and you've got a lookup table. My philosophy is to precalculate everything that can be precalculated. Just remember to `free()` the memory when the effect is done.

So is this all starting to fall into place? With the above specifications laid out, you can easily make your effects such that they can be executed one frame at a time. The only question that remains is how to store the effects in memory. Well, effects aren't something I like to have a limit on, and since you're going to be accessing them in sequential order all the time, this is a great candidate for a linked list. When you want a new effect, just stick a structure on the end of the list. When an effect is finished, just cut the links. It's as simple as that! If you're not familiar with linked lists, find a tutorial, because I use them a lot. :)

One other thing I'll do with effects is to give them an extra parameter in their corresponding script commands that flags whether or not they should link the script that calls them. For instance, if I wanted to fade the screen down, and only then display some text, I could put this in my script:

```
FadeDown 50, 5, TRUE
ShowText 4, 230
```

This says that the call to FadeDown should link the script, and thus the text should be shown only after the fade is complete. But change the `TRUE` to `FALSE`, and the text will show while the screen is fading. This is a very simple, yet very effective way to either run several effects one after the other or simultaneously, with very little effort on your part.

Closing

All right, that about wraps up this long-overdue article. We covered splitting a program into multiple components which can be developed independently of one another, and I think you've got all the basics now for designing a scheme that will let you run scripts and graphical effects that make Windows happy. In other words, they don't hog the processor for too long at once. You should be able to implement all these things and still maintain a good frame rate.

One thing to notice from this article is just how much of the game can be controlled with scripts. I use them for everything -- initialization, image loads, map loads, NPC setup, battle effects, fading, enemy AI, menu setup, etc. And as we'll see in a future article, probably #9, this can all be achieved with a scripting language that is very straightforward and easy to implement, with no need for any black magic on the programmer's part.

Now that I've gone over the basics of how you structure a Windows game program, the

next article will finally be the long-awaited tile engine. I promise this time. :) I'm not sure how long it will be until I get around to it, but it's coming. In the meantime, if you've got any questions, go ahead and hit me with them:

E-mail: ironblayde@aeon-software.com
ICQ: UIN #53210499

Happy coding, until we meet again...

Copyright © 2000 by Joseph D. Farrell. All rights reserved.

Discuss this article in the forums

© 1999-2001 Gamedev.net. All rights reserved. Terms of Use Privacy Policy
Comments? Questions? Feedback? Send us an e-mail!