Game Programming Genesis Part I : Beginning Windows Programming **GameDev.net**

Game Programming Genesis
Part I : Beginning Windows Programming

by Joseph "Ironblayde" Farrell

Introduction

The purpose of this article is to introduce the very basics of Windows programming. By its end, you should be able to put together a working, albeit simple Windows program. All that is required is a basic knowledge of C. I rarely use C++ extensions in my code. However, since Windows itself is object-oriented, a little knowledge about classes never hurt anyone. If you're not familiar with it, don't worry, you won't need anything complicated, so you should be able to pick up what you need as you go. All of the code examples included have been tested using the Microsoft Visual C++ 6.0 compiler. If you don't have a Win32 C/C++ compiler, this is the one to get. That said, let's get started!

Setting Up

The two header files that contain most of the Windows functions you'll need are `windows.h` and `windowsx.h`. Make sure you include both in your programs. Aside from that, you'll just be using the standard C headers, like `stdio.h`, `conio.h`, and so on. Aside from that, there's one line of code you'll see at the beginning of many Windows programs:

```
#define WIN32_LEAN_AND_MEAN
```

Besides having a cool sound to it, this line excludes some MFC stuff from the Windows header files, to speed up your build time a little bit. Since you're not likely to be using MFC for games programming, it's probably a good idea to use this most of the time. If you've never seen this type of statement before -- a `#define` directive followed by only a name -- it has to do with something called conditional compilation. Take a look at this example:

```
#ifdef DEBUG_MODE
    printf("Debug mode is active!");
#endif
```

If the program containing this code has a line in the beginning that reads `#define DEBUG_MODE`, then the `printf()` statement will be compiled. Otherwise, it will be left out. This is a useful way to enable or disable code within your program that helps you track down any logic errors you might have. In the case of `WIN32_LEAN_AND_MEAN`, its definition is used to remove rarely-used components of the Windows header files. Got it? Good. On to the code...

The WinMain() function

Just as all DOS-based C programs begin execution with the `main()` function, Windows programs begin with the `WinMain()` function. A basic, empty `WinMain()` function looks something like this:

```
int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
    return(0);
}
```

For a function that does nothing but return a value, it's sure got a lot of unfamiliar stuff in it! First things first; what's the deal with the `WINAPI` declarator? This is an example of what's known as a calling convention. It affects such things as the way parameters are passed to the function, which function performs stack cleanup, and a few other things that you never really see. A function with the `WINAPI` calling convention passes parameters left-to-right, as opposed to the default right-to-left order. Unless you're going to be using assembly language with your programs, you don't need to know all the details of how calling conventions work, only that `WinMain()` must have the `WINAPI` convention specified.

Next, let's take a look at the four parameters that the function receives:

`HINSTANCE hinstance`: This is a handle to the instance of your application. Basically, these are like pointers that are used to keep track of all the applications that are running at any given time. Many Windows functions take the instance of your application as a parameter, so it knows which application to apply the action to.

`HINSTANCE hPrevInstance`: You don't need to worry about this parameter, as it's now obsolete. In older versions of Windows, this would be a handle to the instance of the application that called your application. The only reason it's included anymore is for backwards compatibility. You'll see a few more things like that as you go on with Windows programming.

`LPSTR lpCmdLine`: This is a pointer to a string containing the command-line parameters used when the program was invoked. Note that there is no parameter specifying the number of command-line parameters, so you'll need to determine that yourself.

`int nCmdShow`: This integer indicates how the main window should be opened. You don't need to do anything with this if you don't want to. It takes values given by constants beginning with `SW_`. Some examples are `SW_SHOWNORMAL` for the default method, `SW_MAXIMIZE` or `SW_MINIMIZE` for maximizing or minimizing windows, etc.

That's about it for `WinMain()`'s parameters. Often, the only one that will be of any consequence is hinstance. Before we go on to actually displaying a window, something needs to be said about the way Microsoft names variables.

Hungarian Notation

Microsoft uses a standardized way of naming variables, functions, constants, and classes that is known as Hungarian notation. You've already seen an example of this in the `WinMain()` function. The Hungarian notation for variable names consists of several prefixes which reveal the variable's data type. These are the prefixes used:

b	`BOOL (int)`

by	`BYTE` or `UCHAR` (`unsigned char`)
c	`char`
cx, cy	`short` (usually lengths; c stands for "count")
dw	`DWORD` (`unsigned long`)
fn	Function pointer
h	Handle (like a pointer, used for Windows objects)
i	`int`
l	`LONG` (`long int`)
lp	Long pointer (32-bit)
msg	Message (we'll cover this later)
n	number (`short` or `int`)
s	String
sz, str	Null-terminated ("ASCIIZ") string
w	`WORD` or `UINT` (`unsigned int`)
x, y	`short` (usually coordinates)

In addition, variable names consisting of more than one word have each word capitalized, with no underscores. For instance, a pointer to an area of memory used for player data might be called `lpPlayerData`. This standard notation is often helpful in understanding code. For instance, in the `WinMain()` function discussed above, without seeing the function header, Hungarian notation tells you that `hinstance` and `hPrevInstance` are handles, `lpCmdLine` is a 32-bit pointer, and `nCmdShow` is an integer.

Function naming under Hungarian notation follows the same rules as variables, minus the prefixes. In other words, the first letter is capitalized, and the first letter of subsequent words in the function name are capitalized as well. An example might be `ShowCharacterStats()`.

The rule for naming constants is that all capital letters are used, and underscores often separate words within a constant's name, or separate a prefix from a constant's name. The constant `WIN32_LEAN_AND_MEAN` is an example. One thing you'll see often in Windows is that constants are often prefixed by an abbreviation of the function with which they are meant to be used. For example, the constants `SW_SHOWNORMAL`, `SW_MAXIMIZE`, and `SW_MINIMIZE`, which I mentioned briefly earlier, have the `SW_` prefix because they are meant to be used as arguments to a function called `ShowWindow()`.

Finally, classes are named in the same manner as functions, except with a capital C preceding the name, so an example class for a vehicle in a racing game might be `CVehicle`.

You don't have to use this naming convention in your programs, but you should be familiar with it, because all Microsoft products follow these guidelines. It is rather convenient if you can convince yourself to start using it. I'm still working on that. Anyway, moving on...

Messages

When you were programming in DOS, you didn't need to worry about other programs that might be running, because DOS is not a multitasking OS. When programming in Windows, however, you must take this into consideration. For this and other reasons, Windows uses what are called messages to communicate with applications and tell them what's going on. Messages serve a variety of purposes. They tell an application when its window is being resized, moved, or closed. They signal to a program that it is about to be closed. They inform a program when part of its window must be refreshed. They can be used to track mouse movement and button presses. The list goes on. In any case, your Windows program must be able to handle these messages.

The way this is done is through the use of a special type of function called a callback function. Callback functions are functions which you don't actually call from your code; rather, certain events cause them to be called. You create such a function by using the `CALLBACK` calling convention, much like the `WINAPI` convention is used for `WinMain()`. I'm going to leave this topic for a minute, though, because before you can process messages for your window, you have to be able to create the window in the first place.

Window Classes

Here's where it helps to know a little about C++, because the first thing you must do to create a window is to create a window class. The class contains all the information about the window such as what icon it uses, the menu attached to it (if any), etc. In just about every Windows program you create, you'll need to create a window class to meet your needs. In order to do this, you need to fill out a `WNDCLASSEX` structure. The "EX" part of the name stands for "extended," as there is an older version of this structure called `WNDCLASS`. We'll be using the extended version. Here's what it looks like:

```
typedef struct _WNDCLASSEX {
    UINT    cbSize;
    UINT    style;
    WNDPROC lpfnWndProc;
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;
    HICON   hIcon;
    HCURSOR hCursor;
    HBRUSH  hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
    HICON   hIconSm;
} WNDCLASSEX;
```

This structure has quite a few members to it, and you must set them all in order to create your window class. It's not so bad, though. Let's run through a brief description of all the fields.

`UINT cbSize`: This is the size of the structure, in bytes. You'll see this a lot, especially if you get into DirectX. It's included so that if a structure of this type (or rather, a pointer to that structure) is passed as a parameter to a function, the structure size can simply be looked up rather than having to be computed. Always set it to `sizeof(WNDCLASSEX)`.

`UINT style`: This is the window style, which takes constants prefixed by `CS_`. Furthermore, you can combine several of these constants by using the `|` (bitwise OR) operator. Most times there are only four you'll use. For the sake of keeping the length of this article down, I'll show you those four. You can always look up the rest on your MSDN Help. You did remember to get Visual C++, didn't you?

CS_HREDRAW	Specifies that the window should be redrawn if it is horizontally resized.
CS_VREDRAW	Specifies that the window should be redrawn if it is vertically resized.
CS_OWNDC	Allows each window to have a unique device context or DC (not covered in this article).
CS_DBLCLKS	Discerns between single- and double-clicks while the mouse is in this window.

WNDPROC lpfnWndProc: A pointer to the callback function that handles messages sent to this window. If you've never used function pointers, the address of a function is simply the function's name, without the parenthesis afterwards.

int cbClsExtra: This is reserved for extra class info, which most programs don't need. Certainly you won't find many uses for this when writing games, so simply set it to 0.

int cbWndExtra: Basically the same as cbClsExtra, except for extra window information. You'll almost always be setting this one to 0 as well.

HANDLE hInstance: This is the instance of the application using the window class, which is one of the parameters passed to WinMain(). This should be set to hinstance.

HICON hIcon: This is a handle to the icon that represents the program, and will usually be set using the LoadIcon() function. Until you learn how to use resources in your programs, you can set this to a generic system icon by using LoadIcon(NULL, IDI_WINLOGO). There are other IDI_ constants representing Windows icons; you can find the list in the Help files for your compiler.

HCURSOR hCursor: This is a handle to the cursor used for the mouse while it is in your window. This is usually set using the LoadCursor() function. Again, you can use resources to load your own custom cursors, but until you learn that, or if you just want the standard Windows cursor, use LoadCursor(NULL, IDC_ARROW).

HBRUSH hbrBackground: When your window receives a message that it needs to be refreshed (or "repainted"), the least that will happen is that Windows will repaint the area with a solid color or "brush." That brush is defined by this parameter. There are several kinds of stock brushes you can load using the GetStockObject() function. Some of these are BLACK_BRUSH, WHITE_BRUSH, GRAY_BRUSH, etc. For now, you're safe using GetStockObject(BLACK_BRUSH). Sorry I'm touching on all of these functions so briefly, but I'm trying to keep the length down. I'll revisit them in future articles, I promise!

LPCTSTR lpszMenuName: If you want to create a window with pull-down menus, this parameter gives the name of the menu to load and attach to the window. Since you don't know how to create menus yet, you can specify no menu by setting this to NULL.

LPCSTR lpszClassName: This is simply a name by which you refer to the class. You can call it anything you want, so use a descriptive name. You might call it "Game_Class" or something like that.

HICON hIconSm: This is a handle to the small icon used on the window's title bar and on the Start Menu bar. You set this the same way you set hIcon -- by using the LoadIcon() function. For now, we'll use LoadIcon(NULL, IDI_WINLOGO) for the standard Windows logo icon.

That's it! Now that you're familiar with all the fields of the WNDCLASSEX structure, you can fill it out and you're ready to create a window. A sample class might look like this:

```
WNDCLASSEX sampleClass;                                    // declare structure variable

sampleClass.cbSize =        sizeof(WNDCLASSEX);            // always use this!
sampleClass.style =         CS_DBLCLKS | CS_OWNDC |
                            CS_HREDRAW | CS_VREDRAW;       // standard settings
sampleClass.lpfnWndProc =   MsgHandler;                    // we need to write this!
sampleClass.cbClsExtra =    0;                             // extra class info, not used
sampleClass.cbWndExtra =    0;                             // extra window info, not used
sampleClass.hInstance =     hinstance;                     // parameter passed to WinMain()
sampleClass.hIcon =         LoadIcon(NULL, IDI_WINLOGO);   // Windows logo
sampleClass.hCursor =       LoadCursor(NULL, IDC_ARROW);   // standard cursor
sampleClass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);  // a simple black brush
sampleClass.lpszMenuName =  NULL;                          // no menu
sampleClass.lpszClassName = "Sample Class"                 // class name
sampleClass.hIconSm =       LoadIcon(NULL, IDI_WINLOGO);   // Windows logo again
```

And you're all set. There is one thing I should mention, though. Notice the typecast to `HBRUSH` on the result of the `GetStockObject()` function. This is because `GetStockObject()` can be used to load many objects, not just brushes, and so it returns a variable of type `HGDIOBJ`, which is a bit more general. In older versions of Visual C++, you wouldn't need the typecast, but VC++ 6.0 is more picky about typecasting, so you'll get an error if you try to compile without it.

The last thing you need to do is register the new class with Windows so you can use it to create new windows. This is accomplished with a simple function call to `RegisterClassEx()`. It only takes one parameter: the address of your structure. So in the example listed above, you would register the class like this:

```
RegisterClassEx(&sampleClass);
```

Now that Windows is familiar with the new class you've created, you can use it to create a window. It's about time, hey?

Creating Windows

The good news is that all you need to create a window is a call to `CreateWindowEx()`. The bad news is that this function takes a lot of parameters. You're probably getting sick of these long lists by now, but this one isn't too bad. Here's the function prototype:

```
HWND CreateWindowEx(
    DWORD dwExStyle,      // extended window style
    LPCTSTR lpClassName,  // pointer to registered class name
    LPCTSTR lpWindowName, // pointer to window name
    DWORD dwStyle,        // window style
    int x,                // horizontal position of window
    int y,                // vertical position of window
    int nWidth,           // window width
    int nHeight,           // window height
    HWND hWndParent,       // handle to parent or owner window
    HMENU hMenu,           // handle to menu, or child-window identifier
    HINSTANCE hInstance,   // handle to application instance
    LPVOID lpParam         // pointer to window-creation data
);
```

First things first: the return value. By now, all these crazy data types that Windows uses are probably starting to look familiar. If not, don't worry, you'll get used to it sooner than you think. The return type is `HWND`, which is a handle to a window. You'll want to store the value returned by `CreateWindowEx()`, as you'll need it as a parameter for several Windows functions. Now, let's tackle that parameter list. Many are self-explanatory.

`DWORD dwExStyle`: Extended window style is something you'll rarely use, so you can set this to `NULL` most of the time. If you're interested, the Help files for your compiler list a ton of constants beginning with `WS_EX_` that can be used here.

`LPCTSTR lpClassName`: Remember when you named the class you created? Just pass that name here.

`LPCTSTR lpWindowName`: This is simply the text that will appear on the window's title bar.

`DWORD dwStyle`: The window style parameter allows you to specify what type of window you want to create. There are a lot of constants that can be used here, beginning with `WS_`, and they can be combined with the `|` operator. I'll list just a few of the common ones:

`WS_POPUP`	A window that has no controls built into it.
`WS_OVERLAPPED`	A window with simply a title bar and a border.
`WS_OVERLAPPEDWINDOW`	A window with a title bar including all standard controls.
`WS_VISIBLE`	Specifies that the window is initially visible.

The `WS_OVERLAPPEDWINDOW` constant is actually a combination of several other constants in order to create a standard window. Basically, you can follow these guidelines. If you want a window that can be maximized, minimized, resized, etc., use `WS_OVERLAPPEDWINDOW`. If you want a window with a title bar but which has a fixed size, use `WS_OVERLAPPED`. If you want a window that has no controls on it whatsoever, use `WS_POPUP`. Such a window will just appear as a black rectangle originally. This is what you'll probably use for writing fullscreen games. Also, always specify the `WS_VISIBLE` flag, unless for some reason you don't want anyone to see your window, or if you want to take care of some other things first, and display the window later.

`int x, y`: These are the coordinates on the screen at which the upper-left corner of the newly created window will appear.

`int nWidth, nHeight`: These are, you guessed it, the width and height of the window, in pixels.

`HWND hWndParent`: This is a handle to the parent window of the window you're creating. This is mostly used with controls such as checkboxes and pushbuttons. For creating a main window, set this to `NULL`, which represents the Windows desktop.

HMENU hMenu: This is a handle to the menu which should be attached to the window. If you're loading a resource menu -- after you learn how to do that -- you would use the LoadMenu() function. For a window with no menu attached, simply set this to NULL.

HINSTANCE hInstance: This is the instance of the application; again, pass the parameter that was passed to WinMain().

LPVOID lpParam: This is something you're not likely to use, especially for games, where only simple windows are needed. It's used for creating things like multiple document interfaces. Just set it to NULL.

At last, we have everything we need to create a window. Here's a sample call that would get the job done:

```
HWND hwnd;
if (!(hwnd = CreateWindowEx(NULL,                  // extended style, not needed
                            "Sample Class",        // class identifier
                            "Sample Window",       // window title
                            WS_POPUP | WS_VISIBLE, // parameters
                            0, 0, 320, 240,        // initial position, size
                            NULL,                  // handle to parent (the desktop)
                            NULL,                  // handle to menu (none)
                            hinstance,             // application instance handle
                            NULL)))                // who needs it?
    return(0);
```

This might be something you'd use for a game, because it's a popup window. Notice that I've enclosed the CreateWindowEx() call inside an if statement. This is because if CreateWindowEx() fails, it returns NULL. The way this statement is set up, if the window can't be created for some reason, WinMain() simply returns and the program ends.

Now you've almost got enough to make a Windows program that creates a functional window. Almost. Remember when we created "Sample Class," and had to provide a pointer to a message handler function? We need to write that function before Windows will let us create anything.

Handling Messages

I've already explained some of the things messages are used for in Windows. Now I'll go over how to make use of them. The prototype for a message handling function looks like this:

```
LRESULT CALLBACK MsgHandler(
    HWND hwnd,     // window handle
    UINT msg,      // the message identifier
    WPARAM wparam, // message parameters
    LPARAM lparam  // more message parameters
};
```

The LRESULT type of the return value is used specfically for message processing functions like the one we're going to write. I talked about the CALLBACK convention a little bit earlier. The parameters are very simple:

HWND hwnd: This is the handle of the window that sent the message currently being processed.

UINT msg: This is a message identifier. The values for this parameter are constants beginning with WM_ (for "Windows message"). The number of different messages that can be sent is ridiculously high, but here are some important ones:

WM_ACTIVATE	A new window is receiving the focus.
WM_CLOSE	A window is being closed.
WM_COMMAND	A menu option has been selected.
WM_CREATE	A window has been created.
WM_LBUTTONDBLCLK	Left mouse button has been double-clicked.
WM_LBUTTONDOWN	Left mouse button has been pressed.
WM_MOUSEMOVE	The mouse has been moved.
WM_MOVE	A window has been moved.
WM_PAINT	Part of a window needs to be repainted.
WM_RBUTTONDBLCLK	Right mouse button has been double-clicked.
WM_RBUTTONDOWN	Right mouse button has been pressed.
WM_SIZE	A window has been resized.
WM_USER	Use this for whatever you want.

WPARAM wparam, LPARAM lparam: The exact use of these parameters depends on which message is being sent, but they are used to further specify the meaning of the message.

If you had to write code to handle every message that your window might receive, you'd probably go insane. I know I would! Thankfully, Windows provides a default message handler. If you don't have any special instructions for handling certain messages, you can always call DefWindowProc(). With that in mind, here is the simplest, fully functional message handler that you could possibly write:

```
LRESULT CALLBACK MsgHandler(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam)
{
    return(DefWindowProc(hwnd, msg, wparam, lparam));
}
```

Simple, hey? Usually you'll want to handle some of these messages yourself. In that case, you can write your own code, and return 0 to tell the program that you've dealt with the message. Here's an example of a message handler that calls an initialization function when the window is created, and calls the default handler for anything else.

```
LRESULT CALLBACK MsgHandler(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam)
{
    if (msg == WM_CREATE)
    {
        Initialize_Game();
        return(0);
    }

    return(DefWindowProc(hwnd, msg, wparam, lparam));
}
```

Your message handler will probably end up being a big `switch` statement to accommodate the messages you want to handle manually, followed by a call to the default handler for everything else. Now there's just one more thing I need to show you before everything is working smoothly, and that's how to make sure your message handler is getting called when it has work to do.

Reading the Message Queue

Near the beginning of your program's main loop, you need to see if the message queue -- where all pending messages are stored -- has anything waiting for you. If so, there are a few things you need to do in order for your handler to do its job correctly. The function you need here is `PeekMessage()`. Here is its prototype:

```
BOOL PeekMessage(
    LPMSG lpMsg,         // pointer to structure for message
    HWND hWnd,           // handle to window
    UINT wMsgFilterMin,  // first message
    UINT wMsgFilterMax,  // last message
    UINT wRemoveMsg      // removal flags
);
```

The return type, `BOOL`, is really just an `int`, but it takes only two values: `TRUE` or `FALSE`. If a message is waiting on the queue, the function returns `TRUE`. Otherwise, it returns `FALSE`. The parameters are pretty straightforward:

`LPMSG lpMsg`: This is a pointer to a variable of type `MSG`. If a message is waiting, this variable will be filled with the message information.

`HWND hWnd`: The handle of the window whose queue you want to check.

`UINT wMsgFilterMin, wMsgFilterMax`: The indices of the first and last messages in the queue to check. Most of the time, you'll only be interested in the first message on the queue, so you would set both of these parameters to 0.

`UINT wRemoveMsg`: Generally this takes only two values, `PM_REMOVE` or `PM_NOREMOVE`. Use the former if you want to remove the message from the queue after reading it, and the latter if you want to leave the message on the queue. Usually, if a message is waiting, you'll prepare it to be handled right away, in which case you should use `PM_REMOVE`.

If a message is waiting, you need to do a few things to get your handler to kick in. Don't worry, it's only two simple calls: one to `TranslateMessage()` and one to `DispatchMessage()`. Their prototypes are very similar:

```
BOOL TranslateMessage(CONST MSG *lpmsg);
LONG DispatchMessage(CONST MSG *lpmsg);
```

The first call performs a bit of translation on the message, as you may have guessed, and the second call invokes your message handler and sends it the appropriate information from the MSG structure. That's all you need to know! With every iteration of your main loop, if a message is waiting, you call these two functions and your MsgHandler() function takes care of the rest. Here's a code example:

```
if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

No problem! Now you can write a Windows program that creates and registers a window class, and creates a window with a valid message handler. That wasn't so bad, was it? There are just a few more things I'd like to mention before I wrap this article up. While we're on the topic of messages, there will come a time when you want to send messages manually. Here's how.

Sending Messages

There are actually two ways to go about doing this. You can either call PostMessage() or SendMessage(). Their prototypes are very similar:

```
BOOL PostMessage(
    HWND hWnd,      // handle of destination window
    UINT Msg,       // message to post
    WPARAM wParam,  // first message parameter
    LPARAM lParam   // second message parameter
);

LRESULT SendMessage(
    HWND hWnd,      // handle of destination window
    UINT Msg,       // message to post
    WPARAM wParam,  // first message parameter
    LPARAM lParam   // second message parameter
);
```

The parameters are the same as those taken by the MsgHandler() function we wrote, so I won't go over them again. The only thing you need to know is the difference between the two functions, so I'll go over each one briefly.

PostMessage() is used when you simply want to add a message to the queue and let your program logic take care of it. The function returns a nonzero value (TRUE) if it succeeds, or zero (FALSE) if it fails. It simply adds the message you specify to the queue, and returns immediately. In most cases, a call to PostMessage() will get the job done.

SendMessage() is a bit different. Notice that it returns an LRESULT, which is used for message processing functions. That's because SendMessage() doesn't post a message to the queue -- it translates the message and invokes the message handler immediately, and doesn't return until the message handler has finished processing the message. SendMessage() is used rather than PostMessage() for events of higher priority that need to occur quickly. Use it when you want something done immediately.

Now that you know that, the topic of messages leads into the last topic I need to mention for now, and that is a major difference between Windows programming and DOS programming.

Program Flow

In DOS, you don't need to worry about any of this message stuff. You don't need to concern yourself with multiple programs running simultaneously. But when you're programming in Windows, these are very important matters. As a result, your programs need to work a bit differently than they do in DOS. Consider this bit of pseudo-code:

```
// main game loop
do
{
    // handle messages here

    // ...

    // update screen if necessary
    if (new_screen)
    {
        FadeOut();
        LoadNewMap();
        FadeIn();
    }

    // perform game logic
    WaitForInput();
    UpdateCharacters();
    RenderMap();

} while (game_active);
```

Suppose FadeOut() works like this: when the function is called, it dims the image on the screen over a period of about a second. When the screen is totally black, the function returns. FadeIn() works in a similar fashion. WaitForInput() simply waits until a key is pressed. Perhaps it stores the input in a global variable somewhere. Now, in a DOS-based game, this is a perfectly acceptable way to do things. In a Windows game, it is certainly not!

Why not? Well, what happens when new_screen becomes true? It fades the screen out, loads a map, and fades back in. Altogether this takes about two seconds. That's two seconds during which no messages are being processed, so the user could do something such as minimize the window, but the program will keep running like it hasn't happened yet. This sort of thing can cause erroneous output, general protection faults, etc. Needless to say, this is unacceptable. The WaitForInput() function is even worse, because it suspends the flow of the program until a key is pressed during every frame. Whereas the previous example has the potential to cause trouble, this one is a near-certainty.

The bottom line is that if your game runs at 30 FPS, you need to make sure the entire main loop executes 30 times a second. Each iteration of the main loop should show only one frame, not many frames as in the theoretical `FadeOut()` function in the example. When first learning Windows programming, this can be a bit of an obstacle, because it's a different way of thinking. However, once you figure out how to set up your program to run this way, I think you'll find it makes for a much more organized and flexible program.

Closing

That's about it for basic Windows programming. While the example we developed over the course of the article doesn't do much except display a window, it contains the entire framework for a functional Windows application. Next time I'll get into handling resources, which allows you to incorporate custom icons, cursors, sounds, menus, and more -- right into your `.EXE`!

If you have any questions or comments about this article or anything else, feel free to contact me via E-mail at ironblayde@aeon-software.com, or via ICQ. My UIN is 53210499. Until next time, farewell!

Discuss this article in the forums