Game Programming 101 Part II

by Bruno Sousa

Introduction

I'm back and I have some good news and some bad news. The bad news is that we will probably not work on the DirectDraw wrapper as I said in the last article. The thing is, we need some more (not-so) important skeletons to help us work with DirectX before we can start on the wrapper. One of them is a Win32 skeleton and another is an error handler. I think we need to build these two things first so that we can debug our DirectX application more easily. The good news is that we will have a complete program at the end of this article. It won't do very much, but it will work (or so I hope).

Enough said, let the coding begin.

Error Handling

Before we start on our Win32 skeleton we should create some kind of error visualization system. Since using GDI (graphical device interface) to display text in DirectX isn't very fast, we're going to make have our error routine log errors in a text file instead. We will also add an option to exit the game when an error occurs. We'll create the class CError to handle all error routines. The prototype for our class will be the following:

```
class CError
{
public:
  FILE    *fErrorLog;
  bool    bQuit;
  LPSTR   lpzMessage;

  CError (LPSTR, bool);
  ~CError ();

  ProcessError (DWORD);

};
```

Now for a brief explanation of each variable. fErrorLog is a pointer to the file where we will log all our errors. bQbuit is just a Boolean variable holding true or false to indicate whether the program should quit or not when an error occurs. lpzMessage is the actual error message. The size of the message will be dynamically allocated when an error occurs for best performance in terms of memory and customizability (that sounded weird). CError is the constructor for the class; it takes a string and a boolean as arguments. The string is the name of the file where the errors will be logged and the boolean sets whether or not the program will quit when an error occurs. The destructor will clean things up when we're done.

For the C programmers, a class is just like a struct with functions; they have more advanced features as well, but we will not use them. Just in case you don't know, a constructor is called when an instance of the class is declared and the destructor is called when the instance is killed, either by the program termination, the variable going out of scope, etc.

We will now start the actual code for the error handling routines. First we must code the constructor and destructor.

```
CError::CError (LPSTR lpzFileName, bool bQuitOnError)
{
  fErrorLog = fopen (lpzFileName, "wt");
  bQuit = bQuitOnError;
}

CError::~CError ()
{

}
```

The constructor is very straightforward; it only opens a file for writing in text mode using the filename specified by the first argument, and sets the quit flag to the value of the second argument.

The destructor is empty for now.

Next is the main core of our error handling. It's very basic for now, but it will grow as we start adding the errors for DirectX.

```
CError::ProcessError (DWORD dwError)
{
  DWORD dwMsgSize;

  switch (dwError)
  {
  default :
    dwMsgSize = strlen ("Unkown error…\n");
    lpzMessage = (LPSTR) malloc (dwMsgSize + 1);
    strcpy (lpzMessage, "Unkown error…\n");
    break;
  }

  if (fErrorLog != NULL)
  {
    fprintf (fErrorLog, lpzMessage);
  }

  if (lpzMessage != NULL)
  {
    free (lpzMessage);
  }

  if (bQuit == true)
  {
    if (fErrorLog  != NULL)
    {
      fclose (fErrorLog);
    }

    PostQuitMessage (dwError);
  }
```

```
      return 0;
    }
```

Let's look at this code more closely. We first declare a variable to hold the length of the string; after that we use the switch statement to find out which error we're handling. Right now, only the `default` is used, and it does three things: it checks the length of the string, allocates sufficient memory for it, and copies it to the string member of our class, `lpzMessage`. We then check to see if the error should be logged to the file and write it as needed. We then free the memory allocated to the string, and finally we check the `bQuit` flag to see if we should abort the program and close the file (after checking to see if the file is open).

I just want to add two comments about this function. First, whenever we want to add another error we should put the code before the `default` case; we will do this later. Second, we don't do any checking to see if all went well with the error handling. This is your homework. Check to see if the memory is allocated correctly, if the file was opened successfully, etc. Try to do this on your own, and if you can't then e-mail me and I'll help you out. I'll also post the corrections in the next article.

We will now need an instance of the class.

```
    CError   ErrorHandling ("errors.log", true);
```

And that's about it for error handling J. Now let's move on to the world of Windows.

The Win32 Skeleton

Many game programmers I know don't really bother learning the basics of Windows programming. They just copy-paste an existing skeleton the got from someone else and write their game on top of it. Even though there is nothing really wrong with that, you will be limited by the skeleton. I'm not going to teach you much about the Win32 API, but I'll teach you enough to put you on the right track to create windows the way you want them to appear.

```
    char szClassName [] = "Chapter2";
    char szWinName [] = "Chapter2";

    LRESULT CALLBACK WndProc(HWND,UINT, WPARAM, LPARAM);
```

The first two variables are the class name and window name. We'll use these in a minute. The function prototype is for handling messages (more on this later).

For any DOS or Unix programmer, your C/C++ program always starts with `void main ()`, or if you want command line arguments `void main (int argc, char *argv[ ])`. In Windows it starts with `WinMain`, which has a few more parameters than you may like. You must also include `windows.h` in your files.

```
    int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
    nCmdShow)
    {
      MSG    msg;
```

```
HWND     hWnd;
WNDCLASSEX  wcl;

bool     bRunning;

ZeroMemory (&wcl, sizeof (WNDCLASSEX));
wcl.cbSize  = sizeof (WNDCLASSEX);
wcl.hInstance  = hInst;
wcl.lpszClassName = szClassName;
wcl.lpfnWndProc = WndProc;
wcl.style   = 0;

wcl.lpszMenuName = NULL;
wcl.cbClsExtra  = NULL;
wcl.cbWndExtra  = NULL;

wcl.hIcon  = LoadIcon (NULL, IDI_APPLICATION);
wcl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
wcl.hCursor = LoadCursor (NULL, IDC_ARROW);

wcl.hbrBackground = (HBRUSH) GetStockObject(BLACK_BRUSH);
```

The return type of WinMain is `int APIENTRY`. This is the way Windows handles specific Win32 API entries. We will also use `LRESULT CALLBACK` but that's a completely different story. We then have four parameters. The first one is the current instance of the program. The second one is the previous instance, but it's not used in Windows 95 or later, so you can just ignore it. The third parameter is an array of strings containing the command line arguments; all arguments are separated with a blank character and, as opposed to DOS/Unix, it doesn't include the executable name as the first parameter. The last parameter is how the Window will show in default mode.

We need another couple of variables. The first is the message to be processed, the second is the handle for the window, and the third one is where we will hold the window class information. The variable `bRunning` will tell us if the game is running or not.

We then set up our window class. The variable names are quite easy, so I'll just cover the not so obvious ones. `cbSize` is needed to let Windows know the size of the class when registering. `lpfnWndProc` is the message handler. `cbClsExtra` and `cbWndExtra` are extra properties of the class, which in this case are set to nothing.

```
if (!RegisterClassEx(&wcl))
{
  ErrorHandling.ProcessError (ERROR_REGISTER_CLASS);
  return (-1);
}
```

We then try to register the class and if an error ocurrs we call our error handling routine and log it. You need to add `#define ERROR_REGISTER_CLASS 1` to `Error.h` and the following piece of code to `ProcessError` just before `default:`.

```
case ERROR_REGISTER_CLASS :
  dwMsgSize = strlen ("Chapter 2 - Error log file\nCould'nt register class...\n");
```

```
        lpzMessage = (LPSTR) malloc (dwMsgSize + 1);
        strcpy (lpzMessage, "Chapter 2 - Error log file\nCould'nt register class...\n");
        break;
```

This will add the "Couldn’t register class" error.

Back to WinMain, we need to create our window and show it. We set the class name, window name, type of window (WS_OVERLAPPEDWINDOW is the standard window with a title bar, minimize/maximize box and close box), and position and size (0,0, 640,480). We set the parent as the desktop, supply no menu, use the current instance and use NULL as the last parameter (advanced functions).

```
      hWnd = CreateWindow (szClassName, szWinName, WS_OVERLAPPEDWINDOW,0 , 0, 640, 480,
          HWND_DESKTOP, NULL, hInst, NULL);

      ShowWindow(hWnd, nCmdShow);
```

And finally we get to the last part of WinMain. We create a loop normally referred to as the message loop. The important thing you need to know is that it receives input from you or the Windows system and sends it to your message handler.

```
      while (bRunning)
      {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
          if (msg.message == WM_QUIT)
          {
            bRunning = false;
          }
          TranslateMessage(&msg);
          DispatchMessage(&msg);
        }
        else
        {
        }
      }

      return 0;
    }
```

We finally return 0 to let the program know that we're done.

Is that it? No, we still need the message handler J.

The Message Handler

```
    LRESULT CALLBACK WndProc (HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
    {
      switch (msg)
      {
      case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
      default:
        return DefWindowProc (hWnd, msg, wParam, lParam);
        break;
      }

      return 0;
    }
```

We don’t need to worry too much about the parameters of the function because the Windows system calls this function automaticlly. You just need to have those four parameters declared.

We determine the type of message and how to handle it by using a `switch` statement. This simple program just uses `WM_DESTROY`, which is a message that is sent to the program when it's beilg closed. We handle the message by telling the program to quit and `return 0` to let Windows know we processed the message.

All messages that aren't processed by us are returned to Windows to use the default processing by calling `DefWindowProc`.

Conclusion

This was a long tiring article. I hope you were able to understand everything we covered. If you have any problems compiling the source, working through the material or any suggestions/corrections, feel free to mail me.

Ohh! Just one more thing, one month before school ended, I finally got a job in the industry and I got a new e-mail account. Feel free to use akura@crosswinds.net to contact me.

We will finally (I promise) dig into DirectDraw in the next article. Until then, stay well folks.

Get the source here!

Discuss this article in the forums