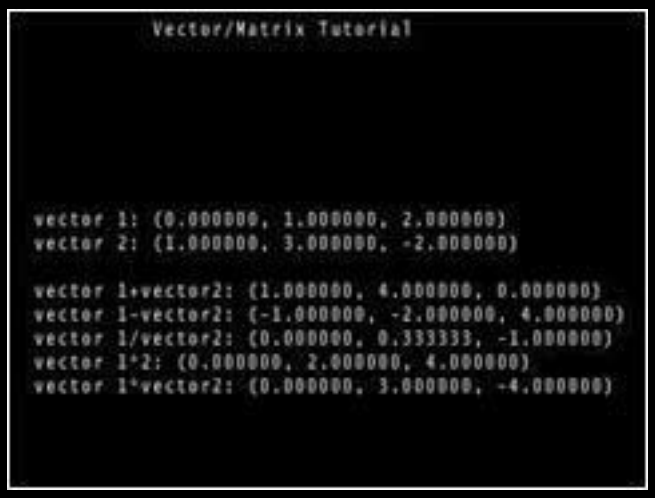
Game Programming Tutorial Index

These tutorials will contain comprehensive coverage on Game Programming in OpenGL. These articles are going to assume some familiarty with OpenGL, and C/C++, but thats about it. Not only will these tutorials cover games, but they will also be covering various effects and things that will be used in games (such as terrain, particles, player models, etc.). These tutorials require DirectX8 (for input, and sound), and the code is made for compatibility with Microsoft Visual C++ 6.0. I strongly recommend that you pick up the following books for your 3D knowledge: Real-Time Rendering (Moller/Haines), The OpenGL Programming Guide, 3D Games: Real-Time Rendering and Software Technology (Watt/Policarpo) and OpenGL Game Programming (Astle/Hawkins). These are all *great* references for general 3D information, and OpenGL information. I would also recommend that you check out the references that I will be listing at the end of each tutorial (when applicable), as they will contain papers/articles specific to the topic at hand, and will help you learn even more about it!

- **Trent Polack**

The Framework And Direct Input:

In this tutorial you will learn some of the extremely cool features of the OpenGL wrapper and code template I have made for your use. I will also discuss using DirectInput to read keyboard using DirectX8. If you don't have the DX8 SDK, you may want to get it now. It's important to note that these tutorials are NOT for new OpenGL programmers. You should have a good understanding of both OpenGL and Visual C++ before you attempt this tutorial or any future OpenGL Game Programming Tutorials!



Vectors and Matrices:

This tutorial is theory, math and more theory. In this tutorial you will create two very useful classes. These classes are for use with the wrapper from lesson 1. One class is for vectors, and the other is for 4x4 matrices. You will also learn about operator overloading to make your code a little bit easier to read when using the two classes! This tutorial is a must read for anyone interested in how matrices work!

Designing a Particle Engine:

Now for some really cool stuff (the first game tut with graphics). In this tutorial, you will learn how to design and code a flexible / powerful particle engine! Create almost any type of special effect possible with particles. Particles are affected by gravity and eachother. Colors can change through interpolation, etc. This is a long tutorial, so be prepared :)

Model Mania:

In this tutorial you will learn how to load two different formats of models: .MD2 (Quake 2 type models) and .MS3D (MilkShape's native format). The .MD2 format is perfect for First Person Shooters (being that it was used in Quake 2), and .MS3D format is good because MilkShape has support for the most popular model formats around. This tutorial is absolutely HUGE! 17 pages when printed! Be prepared for some wicked code and ALOT of reading :)

This code is not guaranteed to be bug free. If you find mistakes or problems with any of the tutorials, please contact Trent Polack.

Back To NeHe Productions!

Game Programming Lesson 1

OpenGL Game Programming Tutorial One:

The Framework, and DirectInput

By Trent "**ShiningKnight**" Polack

Introduction:

One day while scanning the Internet, I noticed about 10+ different series of OpenGL programming tutorials, but I also noticed the extreme lack of Game Programming with OpenGL tutorials. And we just can't have that. So, this is where this series comes into play. It will be a series of tutorials on 3D Games and Special Effects. Ranging from topics such as: Particle Engines, Terrain Engines, Model Loading, and of course, cool games to mess around with. :-)

You may be asking yourself "why does game programming require special tutorials?" Well, the answer is that special notice needs to be taken when programming certain things. Such as realism, coolness, playability, good control, and speed. Games are a completely different ballgame than simple apps, and 'business' apps. An average game needs to be fun, sound, control, and look good, while having as little amount of slowdown as possible. Whereas your average business app only needs to fulfill one specific purpose... Wow. ;)

I am going to make these tutorials as easy as possible for anyone to learn, but unfortunately they **will** require some previous knowledge. First of all, a good knowledge of C, and a basic knowledge of C++ is needed (and if you don't know C at all, but rock with C++, don't worry. You actually do know C :)), as well some decent OpenGL knowledge (all of which you can learn through NeHe's tuts). I will try to use as little math as possible, but when I do need it, I will explain everything.

This first tutorial is going to seem rather 'boring' compared to the rest of the series, but unfortunately it is necessary. In this tutorial you will learn some of the extreme coolness of the OpenGL wrapper and code template I have made for your use. I will also be talking about using DirectInput to receive your keyboard input (using DirectX8, so if you don't have the SDK, you may want to get it now). Sooooo, without further chit-chat, lets start.

The Wrapper:

The wrapper's files (as of right now) are: shining3d.h, shining3d.cpp, math.cpp, and directinput.cpp. shining3d.h is the header file that: declares the necessary headers we will want to include, code that searches for the necessary libraries at compile time, preprocessor constants, simple structures, classes, and some global functions. Let me tell you right now, I will **not** be going through all of the previous files line by line. I will be skipping the internal code of the font functions, the initiation/shut down functions, and a lot of the other windows information. I will assume that you are already familiar with these (and some other OpenGL information that you can go learn in NeHe's tutorials. I will tell you what specific NeHe tutorial to refer to when I do that, go check out his site now at

http://nehe.gamedev.net (though, since you are reading this tutorial now, you are already there). :)

Now, lets start at the top of the shining3d.h header file. It starts by including some necessary files that we will need at one time or another, and then it include the DirectInput8 header file, as well as some OpenGL headers. And in case, you aren't familiar with the following code, I will explain it briefly:

```
#pragma comment(lib, "Dxguid.lib")              // Search For the DXguid Library At
Compile Time
#pragma comment(lib, "Dinput8.lib")             // Search For the DirectInput8
Library At Compile Time
#pragma comment(lib, "OpenGL32.lib")            // Search For The OpenGL32 Library At
Compile Time
#pragma comment(lib, "Glu32.lib")               // Search For The Glu32 Library At
Compile Time
```

Those lines search for the correct library (provided in the quotation marks) at compile time, that way, you don't have to include those libraries every time you want to compile the program. Sorta helpful. ;)

Next, let's jump to the classes! The first class is the main class that you will **always** need in your programs.

```
class SHINING3D
      {
```

Now, lets start by going through the private variables of the class. And just for a little refresher, the private parts of a class can **only** be accessed by information IN that class. For example, say you wanted to access a private int "bob." You could access "bob" in absolutely any function within that class, but if the user wanted to access "bob" from within their program, it would not be possible. Get it? Hopefully, because I just drank the last Mountain Dew in my house. Anyway, here are the private variables:

```
GLuint base;
S3DTGA font_texture;
```

These are both used for the font functions that we have. Now here are the font functions (you'll notice that I am jumping around the class a little, so bare with me :)):

```
GLvoid glPrint(GLint x, GLint y, const char *string, ...);
GLvoid Font_Init(GLvoid);
GLvoid Font_Shutdown(GLvoid);
```

You will need to put Font_Init() in the initiation part of the console if you want to use the font printing function. Similarly, you will need to put Font_Shutdown() in the shutdown part of the console. glPrint(GLint x, GLint y, const char *string, ...) will print text that you provide for the third argument, at the location you provide for the first two arguments. You can go to the main console (main.cpp) to see an example of using the function right now. I will fully explain the main console (I call it a console, you can call it "the main file," "the main template," "the main framework," or whatever floats your pickle... Or is that "tickles your boat?" I always seem to forget...) later on in the tutorial. Anyway, now you understand what each font function does, if you want further information on fonts, you can check out the functions' definition in shining3d.cpp or go to NeHe's tutorials on fonts, which are Tutorials 13-15.

Next on the list, (back to the private functions) is the cosine and sine initiation function. Here is the declaration:

```
GLvoid SIN_COS_Init(GLvoid);
```

And, even though the function is already called when the program starts (this is accomplished by using the class's constructor. A quick explanation of this is that anything in the constructor is done when the program starts), I will be going through the function definition anyway. The definition is in math.cpp. And here it is:

```
GLvoid SHINING3D::SIN_COS_Init(GLvoid)
      {
      GLuint loop;                                  // Looping Index, For The
```

```
Angle
       GLfloat radian;

       for(loop=0; loop<720; loop++)                    // Loop Through The 720
Angles
        {
        //convert angle to radians
        radian=(float)DEG_TO_RAD(loop);         // Turn Degrees To Radians

        //fill in the tables
        SIN[loop]=(float)sin(radian);           // Fill In The Tables
        COS[loop]=(float)cos(radian);
        }
       }
```

Ok, lets go through this. First of all, in case you didn't notice, I created some global arrays in the header for the SIN and COS *look-up tables* (values that are pre-computed to save time in the middle of a game). The previous function fills in the contents of both of the tables. For instance, if you want to know the cosine of 27, the code to do it would be:

```
Answer=COS[28];
```

Yes, we are *still* in the Shining3D class. Next we are going to go through the public variables (which are open to absolutely everything, being that they are *public* :)). Here they are:

```
int SCREEN_WIDTH;
int SCREEN_HEIGHT;
int SCREEN_BPP;
bool fullscreen;
```

Those are more along the lines of 'reference' variables for your use. The SCREEN_WIDTH, SCREEN_HEIGHT, and SCREEN_BPP are used to hold the windows width, height, and bits per pixel respectively. Those variables are assigned a value when the window is created (using the function that will be explained very shortly... hehehe). Last, but not least, of the public variables is the fullscreen flag, which tells whether or not the window is in fullscreen or windowed mode. It also is assigned a value when we create the window. Soooo, guess what we are going to talk about now. :) Here is the window creation function's declaration:

```
bool OpenGL_Init(int width, int height, int bpp, int screenflag);
```

I am not going to go through the definition, as you should already know how to create a window, and if you don't, I would recommend that you go do NeHe's first tutorial, and going to get a Mountain Dew or two wouldn't be a bad idea either... Now, there are some things that you need to know to use the previous function. First of all, if anything went wrong, the function will return **false** (which you really don't need to know, because if anything went wrong, it should be pretty obvious. If something did go wrong, three things could potentially happen: the window doesn't even show up, the window shows up though funky looking, and worst of all, you could get a blue screen of death). Next, you need to provide the width, height, and bits per pixel of the window. Some common width by height resolutions are: 640x480, and 800x600. I will be using 16 *bits per pixel* (known as 'bits' for the rest of the tutorial) most of the time, but you can also use 24 or 32. The higher the bit count, the more memory is required for the program, and the slower it will run, but it *will* look better. And for the last argument, there are three possible flags that you can pass to indicate what kind of window you want, here they are:

```
* FULLSCREEN  - Will Automatically Start In Fullscreen Mode
* WINDOW      - Will Automatically Start In Windowed Mode
* GIVE_CHOICE - Will Give The User A Choice Between Fullscreen And Windowed Mode
```

You will want to put a call to that function as the very **first** function you call in the initiation part of your program. And a quick note on the shutdown function (declaration shown below):

```
void OpenGL_Shutdown(void);
```

That function completely shuts down OpenGL, and the window that you created. You will want to put a call to that as the **last** thing in your shutdown function (just because, when it comes to initiation and shut down, the first thing created will usually want to be shut down last (and vice versa).

The last of the windows functions is shown here:

```
bool HandleMessages(void);
```
This function handles all the messages that your window will receive throughout it's lifetime. I will show you where to put it when we talk about the main console.

```
SHINING3D()
       {        SIN_COS_Init(); }
~SHINING3D()
       {        }
```
Those are the class's *constructor*, and *deconstructor* (which is like the constructor, except that it handles all of the shut down measures. A deconstructor is known by the little tilde, '~,' in front of it). As I said, we put the SIN_COS_Init(...) in the constructor, so its automatically called.

Now, lets talk briefly about the error log. Here is it's class:

```
class LOG
       {
       private:
        FILE* logfile;

       bool Init(void);
       bool Shutdown(void);

       public:

       bool Output(char* text, ...);

       LOG()
        {       Init(); }
       ~LOG()
        {       Shutdown();     }
       };
```
The only thing you need to worry about is the text output function:

```
bool Output(char* text, ...);
```
This function outputs text to the error log. You will treat it exactly like printf(...). An error log's object is already created inside the wrapper (so that I can output how things went inside the window creation function, shutdown, etc.). So if you want to use the output function, use it like this:

```
S3Dlog.Output("Hi");
```
That's about it for the classes (except for the DirectInput class, which will be covered very soon, so just be patient ;)). Now, I am going to go into some very brief details on the macros of the wrapper:

```
* RANDOM_FLOAT           - Finds A Random Float Between -1.0 And 1.0.
* CHECK_RANGE(x,min,max) - Checks The Range Of A Number, To Make Sure That It
                           Is Within The Min And Max That You Provide
* DEG_TO_RAD(angle)      - Converts An Angle That You Provide Into A Radian Value
* RAD_TO_DEG(radians)    - Converts A Radian That You Provide Into An Angle
* SQUARE(number)         - Multiplies A Number That You Provide By Itself.
```
Now we are going to go through some of the global functions. You should have an understanding of **all** of these

functions (except for maybe Random(...)) already.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam);
```
This function is needed for the window, you don't have to worry about calling it anywhere, its all handled for you by windows.

```
void Resize_GLScene(GLsizei width, GLsizei height);
```
This function resizes the OpenGL scene so that if the user changes the width or height, the scene's perspective will not get all screwed up. Though I have designed the window so that the user can't alter the sizes. This may seem like an odd thing to do, but say that you have your game set up for a certain window resolution and then someone goes and alters the width. Now, you may not have planned for that, and now the user can see a lot more than you had wanted them to.

```
bool TGA_Load(S3DTGA_PTR image, char* filename, GLfloat minfilter, GLfloat
maxfilter);
```
This function loads a .tga image from a file, into the address of a S3DTGA structure that you provide. You also need to provide the image's minfilter (if the image is smaller than it's actual size), and the maxfilter (if the image is larger than it's actual size). The filters are the same as you would use in OpenGL (I usually use GL_LINEAR for all of my textures, as it is the happy medium between speed and quality.

```
GLint Random(GLint min, GLint max);
```
This function returns a random value between the range that you provide. So if you wanted to get a random number between -10, and 11461 you would call the function like this:

```
Mynumber= Random(-10, 11461);
```
Yes! Now its time to move on to DirectInput!!! Using DirectInput for the keyboard is **really** easy, and I will try to make it even easier than it actually is! :) Sooooo, lets get to it! :)

Pressing Buttons with DirectInput:

DirectInput, as I just said, is **really** easy. I have already encapsulated the DirectInput information into a class, and its all ready and waiting for you to use it. I will be going through all of the functions anyway though, because this may be something new. Just a little bit of information before we get into the code though. All DirectX components use something called the Component Object Model (COM), which is why some of the functions and conventions used in DirectX may seem a little odd at first. I could go into explaining what COM actually is, but it is really confusing, and you really don't need the extra details. Now lets get back to the code! First we are going to go through the private Dinput variables that are necessary:

```
LPDIRECTINPUT8       lpdi;                          //Main DirectInput Object
LPDIRECTINPUTDEVICE8 lpdi_keyboard;                 //Keyboard's DirectInput
Device
```
LPDIRECTINPUT8 (a long pointer to a directinput8 object) is the type that DirectInput sets aside for the main object. lpdi is the main object, which acts as a manager to all of its 'employees,' which are the LPDIRECTINPUTDEVICE8 types. We must first initialize lpdi before we initialize the 'employee' lpdi_keyboard. lpdi_keyboard is the DirectInput device that will represent our keyboard once we initialize it. But, before we initialize the keyboard, we must initialize the main object (to manage the creation of the keyboard device object). We will want to create a variable to see if any of the initiation stuff will fail, so this is why we do this:

```
HRESULT hr;                                         //DirectX Result Variable
```
That is a specific type of variable specifically for DirectX that contains all of the DX's error information. We will assign the result of a function to that variable, then test that variable to see if something failed, or if everything went well!

```
//Create The Main DirectInput Object
```

```
hr= (DirectInput8Create(hinstance, DIRECTINPUT_VERSION, IID_IDirectInput8,
(void**)&lpdi, NULL));
```
That line initiates the main DirectInput object. I will explain the arguments in order of appearance: the first argument it a handle to our window's instance, so that DInput is familiar with the parameters of the window. The second argument is a constant that Microsoft just wants us to put there. The third argument tells DirectX what version of DirectInput we are going to use. The next argument is a pointer to our main Dinput object, and we will set the last argument to NULL, as it contains extra COM information that we really don't need.

```
if(FAILED(hr))
	{
	MessageBox(hwnd, "Could not create main DInput object", TITLE, MB_OK);
	return false;
	}
```
This, as I said before, checks to see if the previous function that we assigned hr to, failed or not. FAILED(...) is a general macro that Microsoft gave us to make our programming life easier. If the hr value does not contain the correct information that we want it to, then we are going to create a message box, and return false to our function. I would create a message box, and output error text to our log (which will be there by our next tutorial), but I wanted you to practice using the log, so this would be a perfect spot to output an error message to it.

```
hr= (lpdi->CreateDevice(GUID_SysKeyboard, &lpdi_keyboard, NULL));
```
Now that our main Dinput object has been created, hopefully, we are going to create our keyboard's main object. We do this by using a *method* (a function of a class) of our main Dinput object. I will, yet again, go through the arguments one by one: the first argument passes a GUID constant that Microsoft provides to tell DirectX that we are using the keyboard. The second argument passes a pointer to our keyboard's object, and yet again, we are going to pass NULL as the last argument. And being that I just showed you how to detect for an error, I am not going to do it again.

```
hr= (lpdi_keyboard->SetDataFormat(&c_dfDIKeyboard));
```
This sets our keyboard's data format, and all we are going to do is pass yet another constant that tells DirectInput that we are using the keyboard, and that we need the correct data format for it.

```
hr= (lpdi_keyboard->SetCooperativeLevel(hwnd, DISCL_FOREGROUND |
DISCL_NONEXCLUSIVE));
```
This sets our keyboard's behavior (cooperation level) with your computer. We are passing our window's main handle to tell the keyboard a little bit about our window. For the next argument we logically (bitwise) 'or' the flags for our keyboard together, so we can create the custom performance that we want. We are setting the keyboard's focus to the window in the foreground, and we want to make it non-exclusive to the rest of the environment (so that our program isn't hogging the keyboard).

And finally...

```
lpdi_keyboard->Acquire();
```
This is the final step in the keyboard initiation process. We are just going to acquire the keyboard for use! That's all there is to it Bob. :)

Now we are going to go through the shutdown procedures (which are almost **exactly** the same for every part of DirectX). First we need to check to see that the keyboard's object actually has information in it. If it does, then we unacquire the keyboard from use, and release the information associated with it. This is how you do it:

```
if(lpdi_keyboard!=NULL)
	{
	// Unacquire The Keyboard
	lpdi_keyboard->Unacquire();
```

```
	lpdi_keyboard->Release();
	lpdi_keyboard=NULL;
	}
```

And, we do the same thing for the main Dinput object, except that we don't have to unacquire this object.

```
if(lpdi!=NULL)
	{
	lpdi->Release();
	lpdi=NULL;
	}
```

That's it! We now have completely set up, and shut down DirectInput!!! Now, we just have one more thing to do, we have to update it every frame to make sure that our program still has the keyboard. To check the keyboard object's state, we are going to do this:

```
hr= (lpdi_keyboard->GetDeviceState(sizeof(UCHAR[256]),(LPVOID)&key_buffer));
```

That checks to see if we still have the keyboard. You will understand why we passed the arguments we did in a few minutes. If the previous function failed, we have to try to reacquire the keyboard, and if still fails, we kill DirectInput.

```
if(FAILED(hr))
	{
		if(hr==DIERR_INPUTLOST)
	 {
	 //Try To Re-Acquire The Keyboard
	 hr= (lpdi_keyboard->Acquire());
	 if(FAILED(hr))
		{
		MessageBox(hwnd, "Keyboard has been lost", TITLE, MB_OK);
		DInput_Shutdown();
		}
	 }
	}
```

Now we just have to learn how to see if a key is being pressed down or not. Well, we have to create a buffer of 256 unsigned chars to hold all of the keyboard's possible keys. Here is how we do it:

```
UCHAR key_buffer[256];                                  // The Keyboard's Key Buffer
```

You don't have to really worry about that too much, but this is what you will really want to burn into your brain:

```
#define KEY_DOWN(key) (key_buffer[key] & 0x80)
```

That is probably **the** most useful macro I have ever seen. It takes a DirectX key constant (which all begin with DIK_, you can find a complete list in the DirectX8 SDK) and accesses the key's entry in our key buffer, then logically (bitwise) 'and' it with 0x80 (which is what Dinput requires). If the key is being pressed down, then the macro returns true, and if it is not being pressed down, it returns false. So, if you wanted to move a ship to the north when the user presses the up directional button, this is how you would do it:

```
If(KEY_DOWN(DIK_UP))
	{
	// Move Ship
	}
```

That's all there is to it! I have created a class to encapsulate all the information, and here it is:

```
class SHININGINPUT
	{
	private:
	 LPDIRECTINPUT8       lpdi;              // Main DirectInput Object
```

```
        LPDIRECTINPUTDEVICE8 lpdi_keyboard;     // Keyboard's DirectInput Device

      public:

      bool DInput_Init(void);
      void DInput_Shutdown(void);
      void DInput_Update(void);

      SHININGINPUT()
       {       }
      ~SHININGINPUT()
       {       }
      };
```

Nothing should seem too hard, all you need to do is put a call to the init and shut down functions at the proper times in your console, and then put a call to the update function at every frame in your program. That's all there is to it (besides doing all of the checking to see if a key is down or not, which you have to do on your own).

The Console:

Ok, we are almost done! Now we are just going to go through the console! First, we are going to include our wrapper's header file:

```
#include "Shining3D.h"
```

Now, we have to declare all of our wrapper's objects. We are going to create the main object, and our DirectInput object (remember the log has already been created).

```
SHINING3D s3d;
SHININGINPUT sinput;

RECT window;
```

RECT window; will be talked about a little later in the console, when we talk about Orthographic projection.

```
bool Game_Init(void)
      {
      s3d.OpenGL_Init(640,480,16, GIVE_CHOICE);
      sinput.DInput_Init();
      s3d.Font_Init();

      return true;
      }
```

This is your main initiation function. Everything that you want to happen when you initiate your program will go here. As you can see, we are creating a 640x480 window, with a 16 bit color depth, and we are giving the user a choice to see if they want full screen or not. Then we create the DirectInput and font stuff! That's all that is going in this function for now.

```
bool Game_Main(void)
      {
```

This is our main game function. This function is called **every** frame to do all of the drawing, updates, etc. First we are going to update DirectInput, and check to see if the user wants to quit the program or not (by checking to see if he/she is pressing escape).

```
sinput.DInput_Update();                         // Update DirectInput To Make
Sure It's Still Alive
if(KEY_DOWN(DIK_ESCAPE))                        // Check To See If The User
```

```
Wants To Quit
	PostQuitMessage(0);
```

Now we are going to clear our depth and color buffers, and set our current matrix to the identity matrix (in a way we are 'resetting' it).

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);	// Clear Screen And Depth
Buffer
glLoadIdentity();					// Reset The Current
Modelview Matrix
```

Now we are going to go into orthographic mode so that we can position our font's location on the screen **a lot** easier! Anything that we want to be on the screen at all times will be put after we initiate ortho view. This is how you do it:

```
// Get In Ortho View
GetClientRect(hwnd,&window);				// Get The Window's
Dimensions
glMatrixMode(GL_PROJECTION);				// Select The Projection
Matrix
glPushMatrix();					// Store The Projection
Matrix
glLoadIdentity();					// Reset The Projection
Matrix
glOrtho(0,window.right,0,window.bottom,-1,1);		// Set Up An Ortho Screen
glMatrixMode(GL_MODELVIEW);				// Select The Modelview
Matrix
```

If any of this looks unfamiliar, go check out NeHe's tutorial, lesson #33. Now that we are in ortho view, we can position and draw our font!

```
glColor4f(1,1,1,1);
// Game Title
s3d.glPrint(240,450,"ShiningKnight");
```

Now that we are done doing things relative to the screen, we can get out of ortho view. Once we are done with getting out of ortho view, since this is all we are going to be drawing in this tutorial, we are going to swap the buffers (NeHe lesson #1), and get out of the function.

```
glMatrixMode(GL_PROJECTION);				// Select The Projection
Matrix
glPopMatrix();					// Restore The Old Projection
Matrix
glMatrixMode(GL_MODELVIEW);				// Select The Modelview
Matrix

SwapBuffers(hdc);					// Finally, Swap The Buffers
return true;
}
```

Now its time for the shutdown function!

```
bool Game_Shutdown(void)
	{
	s3d.Font_Shutdown();

	sinput.DInput_Shutdown();
	s3d.OpenGL_Shutdown();
```

```
        return false;
        }
```

That's it for our shutdown function. We are shutting things down in the opposite order of which we created them. And that's about all there is to it for now.

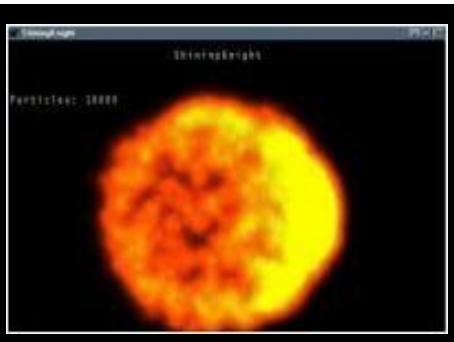
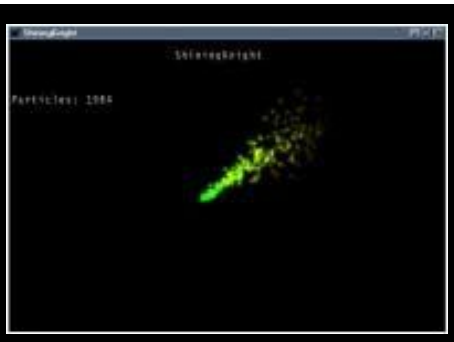
Last, and most necessary, we are going to define our WinMain function (which is equivalent to Main(...) in DOS. First we are going to make sure that our initiation function didn't fail, and if it did, we quit the function. Next we go into our main game loop, and handle our window's messages, and call Game_Main(). If that returns a false value, we quit the main game loop, call our shutdown function, and quit the program!

```
int WINAPI WinMain(     HINSTANCE hInstance,
                HINSTANCE hPrevInstance,
                LPSTR lpCmdLine,
                int nCmdShow)
        {
        // Do All Of The Initiation Stuff
        if(Game_Init()==false)
         return 0;

        while(TRUE)
         {
         if(s3d.HandleMessages()==false)
                 break;
         // Go To The Game_Main Function To Do Everything
         Game_Main();
         }

        // Call Game_Shutdown, And Do All The Shutdown Procedures
        Game_Shutdown();

        return 0;
        }
```

That's all there is to the console!

Conclusion:

Well, that was a fairly long tutorial. Sorry that we couldn't do any cool graphics this time, but just wait until the next one!!! I am putting the finishing touches on the Particle Engine for the next tutorial right now, and I am also working on the terrain engine a little too. The model tutorial is also going to **rock**. I also have some really cool ideas for the first game that we are going to be creating! If you have any questions, comments, ideas for tutorials that you would like to see, or just want to talk to me, e-mail me at ShiningKnight7@hotmail.com. Annnddddd, until next time: "Happy coding!" Oh yeah, and check out the some of the sneak preview of some of the stuff that our Particle Engine will be able to generate with about 10 lines of code. The first is an explosion, the second is a little moving trail that goes around the screen, and the third is a **big** ball that bounces around the screen, and the particle's attraction to each other gets weaker every time, until they fade away.


DOWNLOADS:

* DOWNLOAD the Visual C++ Code For This Lesson.

Back To NeHe Productions!

Game Programming Lesson 2

OpenGL Game Programming Tutorial Two:

Vectors and Matrices

By Trent "**ShiningKnight**" Polack

Introduction:

Now, before we can get into all of the cool graphics, sounds, and all out games, we must learn a little bit of math that is necessary for 3D. In this tutorial we are going to add two **very** useful classes to our wrapper: one for vectors, and one for 4x4 matrices. We are also going to do a bit of operator overloading to make our code a little bit easier to read when using these two classes!

Vectors:

Vectors are used to represent *direction* and *length* (sometimes referred to as *magnitude*). Vectors do not have a set location in 3D space, but they do have a set length. A vector is usually an ordered triple like this:

```
V = (x, y, z)
```

Vectors are usually represented as a line, so you can see that each line has its own direction, and length. You can see this in figure 1.1.

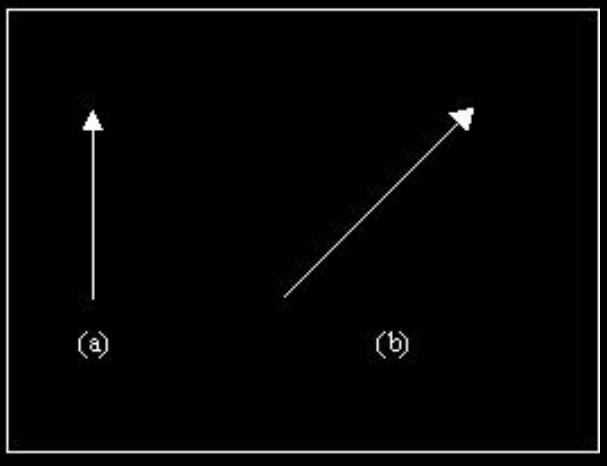


Figure 1.1 - Two examples of vectors

As you can see, each of the above two vectors has it's own direction, and length. Now, lets get on to the math. First of all, let's start with some simple vector addition:

```
R= V1 + V2
```

```
R= (V1x + V2x, V1y + V2y, V1z + V2z)
```

As you can most likely see, all you have to do is add the first vector's component by the second vector's corresponding component. Now, we are going to learn how to multiply a vector by a scalar (single value). This is used mainly to alter a vector's length. For example, if you multiply a vector by 2, you will have a resulting vector that is twice as large as the original one. To multiply a vector by a scalar, all you have to do is multiply each component of the vector by the scalar. Not too hard, see:

```
V * s = (Vx*s, Vy*s, Vz*s)
```

Next on our list of 'vector-math-mania' is how to get the magnitude/length (from now on, I'll be using those two interchangeably). A vector's magnitude is represented as '**|V|**.' And the magnitude of a vector is also a scalar (a single variable, instead of the usual ordered triple). Heres how you do it:

$$|V| = \sqrt{(V_x^2 + V_y^2 + V_z^2)}$$

Now, once that has been done, you can now *normalize* a vector! Vector normals are used **a lot** when it comes to lighting, so you better be familiar with them. Basically what a normal does is turn a vector into something called a "unit vector", which is what you need to have the vector in when you send the normal to OpenGL. You can probably get by without knowing them, but if you understand what a normal is, and why they are used, it will help you out in the long run. So, here is how you get a vector's normal:

```
U  = V / |V|
```

There you go your vector has now been normalized! Rejoice and be glad, because things are only going to get more complicated. ;) Now, lets get dotty, and learn about how to get two vector's dot product. The dot product is also referred to as the inner product. Here is how you get it:

```
V . D = (Vx * Dx) + (Vy * Dy) + (Vz * Dz)
```

See, not too hard! :) Now, the next thing about vectors is the hardest part of vector math that you will have to learn, the cross product of two vectors. We will need to know a vector's cross product when we do things such as collision detection, lighting (you hear that mentioned a lot don't you), and when doing some physics! And here is the last equation you will need to know for vectors (in this tutorial at least... hehe):

```
C = VxVX
```

```
C = (x, y, z)
```

```
C = (((Vy * VXz) - (Vz * VXy)), ((Vz * VXx) - (Vx * VXz)), ((Vx * VXy) - (Vy * VXx)))
```

The cross product of two vectors returns a normalized vector. And that's it, we are now done with the vector math theory!!! Next, we are going to make a little vector class for our use.

Coding a Vector Class:

Now, being that we just went over all of the theory, I am not going to talk about **every** section of the new code, as most of it is pretty self-explanatory. I have coded a vector class, which you will see in a second, but before that, lets talk about something that could be useful for increasing your codes clarity! Since your average vertex requires the same information, I have made a little define to make our vector class, our vertex class also:

```
#define VERTEX3D VECTOR3D
```

Now, if you are going to create a vertex, you can use VERTEX3D and always remember that that object is going to be used for a vertex, not a vector. Now, here is the vector/vertex class:

```
class VECTOR3D
{
	public:
	 GLfloat vertex[3];

	float ComputeLength(void);

	friend VECTOR3D operator+ (VECTOR3D v1, VECTOR3D v2);
	friend VECTOR3D operator- (VECTOR3D v1, VECTOR3D v2);
	friend VECTOR3D operator* (VECTOR3D v1, GLfloat scalar);
	friend VECTOR3D operator* (VECTOR3D v1, VECTOR3D v2);
	friend VECTOR3D operator/ (VECTOR3D v1, VECTOR3D v2);

	VECTOR3D()
	 {	memset(vertex, 0, sizeof(GLfloat[3]));  }
	VECTOR3D(GLfloat x, GLfloat y, GLfloat z);
};
```

That's our little vector/vertex class! vertex[3] is our main vector array. And then we have one member function for calculating the length of an object's vector array, and it returns the result as a float value. Now, the next parts may seem unfamiliar to you, what these are doing is a process called "operator overloading." Normally, if you had two objects of the VECTOR3D class, and you tried to add them together like this:

```
Result= v1+v2;
```

You would get an error, but if you overload the '+' operator, then you can do the previous line without any errors. I am not going to show the definition of all of the overloaded operators, but here is the code to overload the '+' operator:

```
VECTOR3D operator+ (const VECTOR3D &v1, const VECTOR3D &v2)
{
	VECTOR3D temp_v(0.0f, 0.0f, 0.0f);

	temp_v.vertex[0]= v1.vertex[0] + v2.vertex[0];
	temp_v.vertex[1]= v1.vertex[1] + v2.vertex[1];
	temp_v.vertex[2]= v1.vertex[2] + v2.vertex[2];

	return temp_v;
}
```

Notice that when you overload operators, you declare them as 'friends' to a class, instead of making them be an actual part of the class. Make sure that you remember that. Also, overloading operators may be cool and easy, but don't overdo it. I have included code for overloading the following operations:

- Adding two vectors together
- Subtracting two vectors
- Multiplying a vector by a scalar
- Multiplying a vector by another vector
- Dividing two vectors

We also have two different constructors for the vector class, one takes three arguments to initialize a vector's information the moment it is created, and the other is the default constructor, in case the other is not used. For instance, these two lines would have the same effect:

```
VECTOR3D vector(0.0f, 1.0f, 0.0f);
```

And the equivalent line:

```
VECTOR3D vector;
vector.vertex[0]=0.0f;
vector.vertex[1]=1.0f;
vector.vertex[2]=0.0f;
```

Also here are some of the other functions I have created:

```
void ComputeNormal(VECTOR3D* v1, VECTOR3D* v2, VECTOR3D* v3);
float ComputeDotProduct(VECTOR3D* v1, VECTOR3D* v2);
VECTOR3D ComputeCrossProduct(VECTOR3D* v1, VECTOR3D* v2);
```

None of the above functions are members of the VECTOR3D class, so remember that. They are pretty easy to figure out, as we just went through all of the theory! Also, if you want to see some operation overloading in action, check out Game_Main() in main.cpp. It has some samples of it, and the resulting vectors are shown on this tutorial's exe.

Matrices:

Now its time to talk about the matrix (no references to the movie, please)! A matrix is basically a two dimensional array containing various data. Here is a quick example:

```
    | 1    2    3   1 |
M = | 2    5    8   8 |
    | 1    1    2   7 |
```

That is a quick example of a 4x4 matrix. Let me say one thing before you start to think the wrong thing. Declaring a matrix in C/C++ isn't going to be like you would think it is. You may think that to declare a 4x4 matrix array, you would do it like this:

```
GLfloat matrix[4][4];
```

Well, you would be half right, and half wrong with that. First of all, in mathematical terms, the dimensions of a matrix are like so: ROWSxCOLUMNS. So a 2x3 matrix would look like this:

```
M =  | 2   5   7 |
     | 3   6   9 |
```

Now, back to declaring it in C/C++. OpenGL accesses matrices in something that is called column-major form, in which you access matrices with the 0 component of the matrix being the upper left hand element, and the 1 component being the element below that, like you can see below:

$$M = \begin{vmatrix} M_0 & M_4 & M_8 & M_{12} \\ M_1 & M_5 & M_9 & M_{13} \\ M_2 & M_6 & M_{10} & M_{14} \\ M_3 & M_7 & M_{11} & M_{15} \end{vmatrix}$$

Now, had you declared the 2D array that you did before (Glfloat matrix[4][4]), then you would be accessing the matrix in row-major form, and access would occur like this:

$$M = \begin{vmatrix} M_0 & M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 & M_7 \\ M_8 & M_9 & M_{10} & M_{11} \\ M_{12} & M_{13} & M_{14} & M_{15} \end{vmatrix}$$

Which would be bad, and then OpenGL would get confused, you would get confused, and it would just be one big mess. So, when you want to declare a 4x4 matrix in OpenGL (and the most prominent matrix that you will be

using is a 4x4) you can just do it like this:

```
GLfloat matrix[16];
```

And access would occur in column-major form, as discussed above. Now, we will discuss various types of matrices, and various operations that you can perform with them. The first type of matrix that we will talk about is the *Zero Matrix*, as shown below:

$$Z = \begin{vmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{vmatrix}$$

A *really* complicated matrix to remember isn't it! ;) You will not find much practical use for the zero matrix though, but its always good to know about it! Now, the identity matrix is very useful in 3D graphics programming however. Its basically setting a matrix to it's default, and allows you to perform transformations with a "clean slate." Here is the identity matrix:

$$I = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

This is what you set the current matrix in OpenGL to when you use glLoadIdentity(). And, as I said, gives you a clean slate to work with. Setting the current matrix to an identity matrix is like setting your point of focus back to (0,0,0). Now, onto some operations that you can perform with matrices! First of course, is matrix addition and subtraction. You treat both addition, and subtraction the same with matrices. You add/subtract one component of one matrix, by the corresponding component in the other matrix. And one quick note, to add/subtract two matrices, they need to have the exact same dimensions. See the example below.

$$M = \begin{vmatrix} M_0 & M_4 & M_8 & M_{12} \\ M_1 & M_5 & M_9 & M_{13} \\ M_2 & M_6 & M_{10} & M_{14} \\ M_3 & M_7 & M_{11} & M_{15} \end{vmatrix} \qquad X = \begin{vmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{vmatrix}$$

$$\begin{vmatrix} M_0 & M_4 & M_8 & M_{12} \\ M_1 & M_5 & M_9 & M_{13} \\ M_2 & M_6 & M_{10} & M_{14} \\ M_3 & M_7 & M_{11} & M_{15} \end{vmatrix} + \begin{vmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{vmatrix} = \begin{vmatrix} X_0+M_0 & X_4+M_4 & X_8+M_8 & X_{12}+M_{12} \\ X_1+M_1 & X_5+M_5 & X_9+M_9 & X_{13}+M_{13} \\ X_2+M_2 & X_6+M_6 & X_{10}+M_{10} & X_{14}+M_{14} \\ X_3+M_3 & X_7+M_7 & X_{11}+M_{11} & X_{15}+M_{15} \end{vmatrix}$$

Then, to do subtraction, you would add everything instead of subtracting. Now it is on to 'fake' matrix multiplication, which is multiplying a matrix by a scalar. All you have to do is multiply every part of the matrix by the scalar. Here is the example, as always:

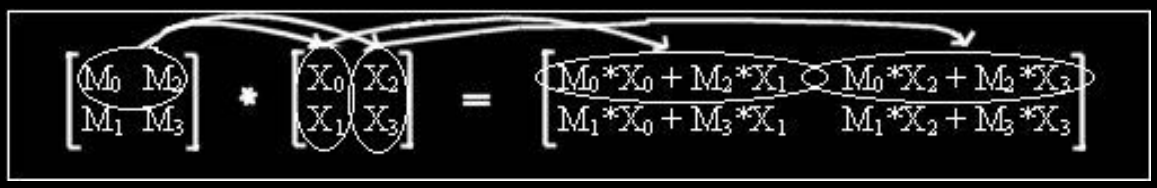
$$M = \begin{vmatrix} M_0 & M_4 & M_8 & M_{12} \\ M_1 & M_5 & M_9 & M_{13} \\ M_2 & M_6 & M_{10} & M_{14} \\ M_3 & M_7 & M_{11} & M_{15} \end{vmatrix} \qquad S \text{ (scalar)}$$

$$M*S = \begin{vmatrix} M_0*S & M_4*S & M_8*S & M_{12}*S \\ M_1*S & M_5*S & M_9*S & M_{13}*S \\ M_2*S & M_6*S & M_{10}*S & M_{14}*S \\ M_3*S & M_7*S & M_{11}*S & M_{15}*S \end{vmatrix}$$

Nothing too hard! But, that's not the 'normal' matrix multiplication. Multiplying one matrix by another matrix is

completely different than you make think it would be. First of all, when you multiply two matrices, the inner dimensions must be the same. For instance, you **can** multiply a 2x3 and a 3x5 matrix, but you **can not** multiply a 3x5 and a 2x3 matrix. The dimensions of the resulting matrix will be equal to the outer dimensions of the matrix that you are multiplying. For example, if I multiplied a 1x2 matrix and a 2x7 matrix, the resulting matrix would be a 1x7 matrix.

Now, lets get on with doing the actual matrix multiplication! To do this, you multiply each row of the first matrix by each column of the second matrix, like the example you see below:



Lets do a quick 2x2 and 2x3 example:

```
    |  1  3 |           | 1  3  5 |
M = |  2  4 |       X = | 2  4  6 |

M*X= | 7   15    23 |
     | 10  22    34 |
```

Hopefully you can figure out how I got that answer, because if you could, then you are done learning math for this tutorial!!! Now, all that's left to do is create a matrix class!

Coding a Matrix Class:

And now, lets code the matrix class! It's **a lot** like the vector class in some senses, except for the array size. Soooo, lets check it out!

```
class MATRIX4X4
{
	public:
	 GLfloat matrix[16];

	void LoadZero(void);
	void LoadIdentity(void);

	friend MATRIX4X4 operator+ (MATRIX4X4 m1, MATRIX4X4 m2);
	friend MATRIX4X4 operator- (MATRIX4X4 m1, MATRIX4X4 m2);
	friend MATRIX4X4 operator* (MATRIX4X4 m1, GLfloat scalar);
	friend MATRIX4X4 operator* (MATRIX4X4 m1, MATRIX4X4 m2);

	MATRIX4X4()
	 {	memset(matrix, 0, sizeof(GLfloat[16])); }
};
```

See, we even have the same operators overloaded in the matrix class (though the definitions are completely different), with the exception that there is no division. You may want to check out the matrix by matrix multiplication operator definition, as it is by far the most complicated one. Only one constructor this time, so if you need to initialize an entire matrix, just do it the long way ;) Also, I have made two member functions to set a matrix's information, one to load the identity matrix, and one to load the zero matrix. That's about it!

Conclusion:

Well, yet again, another basically graphic-less tutorial. That will all change with the next tutorial though! As I said, I am sorry about the delay with getting these tutorials up, my computers been broken, but I am getting it back next week! So expect some really cool tutorials in the next few weeks!

Ok, I have to give a **huge** thanks to NeHe for giving me the opportunity to do this! I also would like to thank Voxel, ZealousElixir, Lord Dragonus, and cmx for helping me out.

DOWNLOADS:

* DOWNLOAD the Visual C++ Code For This Lesson.

Back To NeHe Productions!

Game Programming Lesson 3

OpenGL Game Programming Tutorial Three:

Designing a Particle Engine

By Trent "**ShiningKnight**" Polack

Introduction:

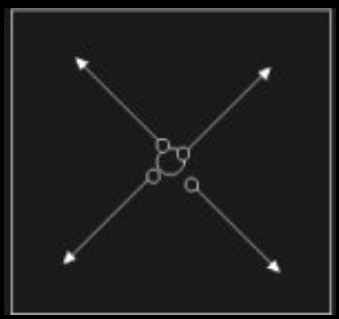
Ok, now that we have the formalities out of the way, we can get to talking about the **really** cool stuff (neat graphics). In this tutorial, we are going to be talking about how to design and code a flexible/powerful particle engine! If you need a little refresher on them, check out NeHe's code for his particle engine (http://nehe.gamedev.net/tutorials/lesson19.asp). And since this is going to be a rather long tutorial, lets get to it.

Particle Theory and Background:

The whole idea behind particle engines started "back in the day" of 1982 (even before I was born... wow). The person we have to thank for all of our particle goodness is a man that goes by the name of William T. Reeves [1]. He wanted to come up with an approach, that he called 'fuzzy,' to be able to render such things as explosions, and fire in a dynamic process. Reeves described the steps to implement a particle engine as the following:

- New particles are generated and placed into the current particle system
- Each new particle is assigned it's own unique attributes>
- Any particles that have outlasted their life span are declared 'dead'
- The current particles are moved according to their scripts (or in our case, according to their pre-assigned velocity vectors)
- The current particles are rendered

In our engine, each particle system (a group of particle's 'master') will have it's own emitter, and some set values to pass on to each particle. In case you need a quick refresher, the figure below should catch you up on some of the various ideas behind a particle engine.



As you can see from that picture, there is one big middle circle; this represents the particle system's *emitter* (a

place in 3D space where all of the particle will start). We also have several smaller circles surrounding it; these represent individual particles, and the arrows show their directional vector (which describes the direction that the particle will continually follow unless acted upon by gravity or such). That's it for your crash course in particle theory. Lets get to the code!

Designing the Particle Class:

I have designed so many different particle engines, and they all were very similar, and although I wasn't very pleased with any of them, I had a very hard time finding another **good** engine to check out. Then, one day, Richard 'Keebler' Benson sent me his particle demo entitled "Particle Chamber"[2]. This demo made me rethink the way that I was designing my engines, and is a very good demo that I recommend **everyone** to check out (the link is at the bottom of this tutorial).

Anyway, I have designed my engine with simplicity, flexibility, and power in mind. I have created a **TON** of different functions that can be used to form almost any type of special effect possible with particles. Before we check out the larger, Particle System class, let's check out the Particle class (which is used for EVERY particle). Here it is:

```
class PARTICLE
{
	private:
	 PARTICLE_SYSTEM* Parent;

	public:
	 VERTEX3D prev_location;		// The Particle's Last Position
	 VERTEX3D location;			// The Particle's Current Position
	 VECTOR3D velocity;			// The Particle's Current Velocity

	 float color[4];			// The Particle's Color
	 float color_counter[4];		// The Color Counter!

	 float alpha;				// The Particle's Current Transparency
	 float alpha_counter;			// Adds/Subtracts Transparency Over Time

	 float size;				// The Particle's Current Size
	 float size_counter;			// Adds/Subtracts Transparency Over Time

	 float age;				// The Particle's Current Age
	 float dying_age;			// The Age At Which The Particle DIES!

void Set_ParentSystem(PARTICLE_SYSTEM* parent);
	void Create(PARTICLE_SYSTEM* parent, float time_counter);
	bool Update(float time_counter);

	PARTICLE();
	~PARTICLE();
};
```

It may look intimidating, but the class is actually quite simple, and plus, I always like to show the simple class first, so if you want to see intimidating, check the next one out. ;) We only have one private variable, and this is used to set the particle's 'parent' system. We need to retrieve some information from the parent when we create a new particle, so that's why that is there.

You may be wondering why we are keeping track of two different vertices (one for the current, and one for the last). Well, this engine takes into account some time aspects and incorporates them into the movement of the particles. That way, if you are on a P75 (ancient) computer, and you are comparing your results to a P4 1.7ghz, the particles

will always be in the same position (though, I'm guessing that the P4 comp's particles will get to their final position a little more smoothly ;)).

Now, we also have two different color variables, one for the current color, and for the color 'counter.' The current color is used for, obviously, the particle's current color. The color counter is used to make the particle's current color transform to the system's specified final color throughout the particle's lifespan. For instance, if I told the particle system to have each particle color start out as orange, and end up being red (sort of like fire), what the color counter would do is take the end color (red), subtract it by the starting color (orange), and divide that by the age at which the particle dies. This comes up with a color away that we are going to use to interpolate the colors through out the particle's life. We also do this for the particle's size, and alpha value also.

To calculate the dying age of the particle, all we do is take the parent's specified life, add a random float (by the way, our macro RANDOM_FLOAT gets a value from [0.0f, 1.0f], or from 0 to 1), and multiply that by the parent's life counter.

Basically, all of the following is used for a particle's transformation throughout its life span. For example, say that you wanted to use do a smoke trail with particles (assuming that you have the correct texture), you would want to have the smoke's particle start out small, and pretty dense, but as time goes on, have the particle enlarge, and get less dense.

Everything that we have just been talking about appears in the particle creation function, shown here:

```
GLvoid PARTICLE::
       Create(PARTICLE_SYSTEM* parent, float time_counter)
       {
       VECTOR3D temp_velocity;
       float random_yaw;
       float random_pitch;
       float new_speed;

       // This Particle Is Dead, So It's Free To Mess Around With.
       age=0.0;
       dying_age= (parent->life)+(RANDOM_FLOAT*(parent->life_counter));
       CHECK_RANGE(dying_age, MIN_LIFETIME, MAX_LIFETIME);

       // Now, We Are Going To Set The Particle's Color.  The Color
       // Is Going To Be The System's Start Color*Color Counter
       color[0]= (parent->start_color.vertex[0])+RANDOM_FLOAT*(parent-
>color_counter.vertex[0]);
       color[1]= (parent->start_color.vertex[1])+RANDOM_FLOAT*(parent-
>color_counter.vertex[1]);
       color[2]= (parent->start_color.vertex[2])+RANDOM_FLOAT*(parent-
>color_counter.vertex[2]);
       color[3]= 1.0f;

       //Lets Make Sure That The Color Is Legal
       CHECK_RANGE(color[0], MIN_COLOR, MAX_COLOR);
       CHECK_RANGE(color[1], MIN_COLOR, MAX_COLOR);
       CHECK_RANGE(color[2], MIN_COLOR, MAX_COLOR);

       // Now, Lets Calculate The Color's Counter, So That By The
       // Time The Particle Is Ready To Die, It Will
       // Have Reached The System's End Color.
       color_counter[0]= ((parent->end_color.vertex[0])-color[0])/dying_age;
```

```
		color_counter[1]= ((parent->end_color.vertex[1])-color[1])/dying_age;
		color_counter[2]= ((parent->end_color.vertex[2])-color[2])/dying_age;

		// Calculate The Particle's Alpha From The System's.
		alpha= (parent->start_alpha)+(RANDOM_FLOAT*(parent->alpha_counter));
		// Make Sure The Result Of The Above Line Is Legal
		CHECK_RANGE(alpha, MIN_ALPHA, MAX_ALPHA);
		// Calculate The Particle's Alpha Counter So That By The
		// Time The Particle Is Ready To Die, It Will Have Reached
		// The System's End Alpha
		alpha_counter=((parent->end_alpha)-alpha)/dying_age;

		// Now, Same Routine As Above, Except With Size
		size= (parent->start_size)+(RANDOM_FLOAT*(parent->size_counter));
		CHECK_RANGE(size, MIN_SIZE, MAX_SIZE);
		size_counter= ((parent->end_size)-size)/dying_age;

		// Now, We Calculate The Velocity That The Particle Would
		// Have To Move To From prev_location To current_location
		// In time_counter Seconds.
		temp_velocity.vertex[0]= ((parent->location.vertex[0])-(parent-
>prev_location.vertex[0]))/ time_counter;
		temp_velocity.vertex[1]= ((parent->location.vertex[1])-(parent-
>prev_location.vertex[1]))/ time_counter;
		temp_velocity.vertex[2]= ((parent->location.vertex[2])-(parent-
>prev_location.vertex[2]))/ time_counter;

		// Now Emit The Particle From A Location Between The Last
		// Known Location, And The Current Location.  And Don't
		// Worry, This Function Is Almost Done.  Its Mostly Comments
		// If You Look At It. Nice To Know I Have Wasted A Lot Of
		// Time Typing These Comments For You.  Blah. :)
		location.vertex[0]= (parent->prev_location.vertex[0])+
temp_velocity.vertex[0]* RANDOM_FLOAT*time_counter;
		location.vertex[1]= (parent->prev_location.vertex[1])+
temp_velocity.vertex[1]* RANDOM_FLOAT*time_counter;
		location.vertex[2]= (parent->prev_location.vertex[2])+
temp_velocity.vertex[2]* RANDOM_FLOAT*time_counter;

		// Now A Simple Randomization Of The Point That The Particle
		// Is Emitted From.
		location.vertex[0]+=(Random((parent->spread_min), (parent-
>spread_max)))/(parent-> spread_factor);
		location.vertex[1]+=(Random((parent->spread_min), (parent-
>spread_max)))/(parent-> spread_factor);
		location.vertex[2]+=(Random((parent->spread_min), (parent-
>spread_max)))/(parent-> spread_factor);

		// Update The Previous Location So The Next Update Can
		// Remember Where We Were
		(parent->prev_location.vertex[0])=(parent->location.vertex[0]);
		(parent->prev_location.vertex[1])=(parent->location.vertex[1]);
		(parent->prev_location.vertex[2])=(parent->location.vertex[2]);
```

```
	// The Emitter Has A Direction.  This Is Where We Find It:
	random_yaw  = (float)(RANDOM_FLOAT*PI*2.0f);
	random_pitch= (float)(DEG_TO_RAD(RANDOM_FLOAT*((parent->angle))));

	// The Following Code Uses Spherical Coordinates To Randomize
	// The Velocity Vector Of The Particle
	velocity.vertex[0]=(cosf(random_pitch))*(parent->velocity.vertex[0]);
	velocity.vertex[1]=(sinf(random_pitch)*cosf(random_yaw))*(parent-
>velocity.vertex[1]);
	velocity.vertex[2]=(sinf(random_pitch)*sinf(random_yaw))*(parent-
>velocity.vertex[2]);

	// Velocity At This Point Is Just A Direction (Normalized
	// Vector) And Needs To Be Multiplied By The Speed
	// Component To Be Legit.
	new_speed= ((parent->speed)+(RANDOM_FLOAT*(parent->speed_counter)));
	CHECK_RANGE(new_speed, MIN_SPEED, MAX_SPEED);
	velocity.vertex[0]*= new_speed;
	velocity.vertex[1]*= new_speed;
	velocity.vertex[2]*= new_speed;

	// Set The Particle's Parent System
	Set_ParentSystem(parent);
	}
```

Phew, I am glad that function is over with! For a quick note, CHECK_RANGE is a macro that checks to make sure that a variable (the first argument) is within a minimum (second argument) and the maximum (last argument) range. Also, look at these 3 lines:

```
velocity.vertex[0]=(cosf(random_pitch))*(parent->velocity.vertex[0]);
velocity.vertex[1]=(sinf(random_pitch)*cosf(random_yaw))*(parent-
>velocity.vertex[1]);
velocity.vertex[2]=(sinf(random_pitch)*sinf(random_yaw))*(parent-
>velocity.vertex[2]);
```

This may be a speed caution, but it also creates the exact effect that we want. I tried using our SIN and COS tables, but they created explosions that were **TOO** perfect (it basically created a perfect sphere, instead of a sphere with 'bumps'). So, sometimes it is necessary to do dynamic calculations.

Next on our list is to show the particle's update function. This function updates the particle that the class currently represents ONLY (this is all handled within the particle system's update function). Lets go through this function bit by bit (or, since this is a pretty large function, byte by byte):

```
bool PARTICLE::
	Update(float time_counter)
	{
	static VERTEX3D attract_location;
	static VECTOR3D attract_normal;
```

This is the start of our update function, we are going to be using the two variables shown here for our particle attraction calculations (this is one of the FEW special effects that I have included within the engine, and it is for the particle's attraction to the emitter).

```
// Age The Particle By The Time Counter
age+= time_counter;

if(age>=dying_age)
```

```
	{
	//Kill the particle
	age=-1.0f;
	return false;
	}
```

The first line of this snippet increments our particle's age by how much time has gone by since the last time the function was executed. The next part will check to see if our particle is dead or not, and if it, we set its age to a negative number, which informs the system that this particle is ready to be re-created!

```
// Set The Particle's Previous Location With The Location That
// Will Be The Old One By The Time We Get Through This Function
prev_location.vertex[0]=location.vertex[0];
prev_location.vertex[1]=location.vertex[1];
prev_location.vertex[2]=location.vertex[2];

// Move The Particle's Current Location
location.vertex[0]+= velocity.vertex[0]*time_counter;
location.vertex[1]+= velocity.vertex[1]*time_counter;
location.vertex[2]+= velocity.vertex[2]*time_counter;

// Update The Particle's Velocity By The Gravity Vector By Time
velocity.vertex[0]+= (Parent->gravity.vertex[0]*time_counter);
velocity.vertex[1]+= (Parent->gravity.vertex[1]*time_counter);
velocity.vertex[2]+= (Parent->gravity.vertex[2]*time_counter);
```

This snippet increases all of the particle's vertices, and it's directional vector by the amount of time that has gone by since the last time our function has executed (déjà vu, eh?). All of this makes sure that our particle will be in the same position on slower computers, as the same particle on a faster system.

```
if(Parent->IsAttracting())
	{
	// Find Out Where Our Parent Is Located So We Can Track It
	attract_location.vertex[0]=Parent->Get_Location(GET_X);
	attract_location.vertex[1]=Parent->Get_Location(GET_Y);
	attract_location.vertex[2]=Parent->Get_Location(GET_Z);

	// Calculate The Vector Between The Particle And The Attractor
	attract_normal.vertex[0]= attract_location.vertex[0]-location.vertex[0];
	attract_normal.vertex[1]= attract_location.vertex[1]-location.vertex[1];
	attract_normal.vertex[2]= attract_location.vertex[2]-location.vertex[2];

	// We Can Turn Off Attraction For Certain Axes To Create Some Kewl Effects
(Such As A Tornado!)

	// Note: This Is NOT Accurate Gravitation.   We Don't Even Look At The
Distance
	// Between The 2 Locations!!!  But What Can I Say, It Looks Good.

	glNormal3fv(attract_normal.vertex);

	// If You Decide To Use This Simple Method You Really Should Use A Variable
Multiplier
	// Instead Of A Hardcoded Value Like 25.0f
	velocity.vertex[0]+= attract_normal.vertex[0]*5.0f*time_counter;
	velocity.vertex[1]+= attract_normal.vertex[1]*5.0f*time_counter;
```

```
        velocity.vertex[2]+= attract_normal.vertex[2]*5.0f*time_counter;
        }
```

All of the previous code calculates our particle's new directional vector, if we have attraction to the emitter on. What all of this does is alters the particle's vector so that it slowly curves back to the emitter over time, for that attraction look. If you would like to see this principle in action, execute the included demo, and press 'a' for a particle explosion, that has attraction turned on.

```
// Adjust The Current Color
color[0]+= color_counter[0] *time_counter;
color[1]+= color_counter[1] *time_counter;
color[2]+= color_counter[2] *time_counter;
```

This snippet is the part I was talking about earlier, that interpolates the current color towards the destined end color.

```
// Adjust The Alpha Values (For Transparency)
alpha+= alpha_counter*time_counter;

// Adjust Current Size
size+= size_counter*time_counter;

// Fill In Our Color Vector With The Final Alpha Component
color[3]=alpha;

return true;    // Yeee-aahhhhh Buddy!
}
```

This just finishes the update function. We are now done with the particle class!!! Oh yeah, and if you'd like some humor, check out some of the comments to the above functions... :) I always like to hide the humor from the actual tutorial, and leave it open for people to discover one day while looking through the code.

Designing the Particle System Class:

This is the turkey and mashed potatoes of the engine (ok... excuse me for not conforming to the classic phrase "meat and potatoes..." Conformity is a bad thing). This is the class that you will be using the most when you want to edit the look and feel of the system. I have created **several** functions for your editing purposes, and I am not going to describe most of them here, as they are very simple to understand, and if you really do need an explanation, I fully comment every one in the source, so check it out. Here is the particle system class:

```
class PARTICLE_SYSTEM
        {
        private:
         bool attracting;                       // Is The System Attracting Particle
Towards Itself?
         bool stopped;                          // Have The Particles Stopped
Emitting?

         unsigned int texture;                  // The Particle's Texture

         unsigned int particles_per_sec;        // Particles Emitted Per Second
         unsigned int particles_numb_alive;     // The Number Of Particles Currently
Alive

         float age;                             // The System's Current Age (In
Seconds)
```

```
        float last_update;                      // The Last Time The System Was
Updated

        float emission_residue;                 // Helps Emit Very Precise Amounts Of
Particles


       public:
        PARTICLE particle[MAX_PARTICLES];       // All Of Our Particles

        VERTEX3D prev_location;                 // The Last Known Location Of The
System
        VERTEX3D location;                      // The Current Known Position Of The
System
        VECTOR3D velocity;                      // The Current Known Velocity Of The
System

        float start_size;                       // The Starting Size Of The Particles
        float size_counter;                     // Adds/Subtracts Particle Size Over
Time
        float end_size;                         // The Particle's End Size (Used For
A MAX Boundry)

        float start_alpha;                      // The Starting Transparency Of The
Particle
        float alpha_counter;                    // Adds/Subtracts Particle's
Transparency Over Time
        float end_alpha;                        // The End Transparency (Used For A
MAX Boundry)

        VECTOR3D start_color;                   // The Starting Color
        VECTOR3D color_counter;                 // The Color That We Interpolate Over
Time
        VECTOR3D end_color;                     // The Ending Color

        float speed;                            // The System's Speed
        float speed_counter;                    // The System's Speed Counter

        float life;                             // The System's Life (In Seconds)
        float life_counter;                     // The System's Life Counter

        float angle;                            // System's Angle (90==1/2 Sphere,
180==Full Sphere)

        int spread_min;                         // Used For Random Positioning Around
The Emitter
        int spread_max;
        float spread_factor;                    // Used To Divide Spread

        VECTOR3D gravity;                       // Gravity For The X, Y, And Z Axis
        float attraction_percent;

       bool Update(float time, int flag, float num_to_create);
       void Render(GLvoid);
       unsigned int Active_Particles(void);
```

```
        float Get_Location(int coordinate);

        void Set_Location(float x, float y, float z);
        void Set_Texture(S3DTGA_PTR texture1);
        void Set_ParticlesPerSec(unsigned int number);
        void Set_Velocity(float xv, float yv, float zv);
        void Set_Size(float startsize, float endsize);
        void Set_Alpha(float startalpha, float endalpha);
        void Set_Speed(float Speed);
        void Set_Angle(float half_angle);
        void Set_SystemFlag(int flag, bool state);
        void Set_Color(float start_red, float start_green, float start_blue, float
end_red, float end_green, float end_blue);
        void Set_Life(float seconds);
        void Set_Spread(int Spread_Min, int Spread_Max, float Spread_factor);
        void Set_Attraction(unsigned int Attraction_Percent);
        void Set_Gravity(float xpull, float ypull, float zpull);

        bool IsAttracting(void)
         {       return attracting;      }
        bool IsStopped(void)
         {       return stopped; }

        PARTICLE_SYSTEM();
        ~PARTICLE_SYSTEM();
        };
```

A little on the big size isn't it. ;) Now, if you'll notice, I made the array of particle public, 'why' you ask? That way you can check a particle's position in space, and see if it is colliding with anything. A particle demo with collision ranks up there in the REALLY cool category. Now, instead of going through the variables one by one, I will just go through the functions, and describe them as I go (because there are quite a bit of them, and to understand their true meaning, you may have to see them in action to fully understand them). First, lets go through the particle system's version of the Update function:

```
bool PARTICLE_SYSTEM::
        Update(float time, int flag, float num_to_create)
        {
        int loop;
        unsigned int particles_created;
        float time_counter;
        float particles_needed=num_to_create;
```

This is the beginning of the function. Notice the three arguments, I'll talk about the first argument later when we see the particle engine in action for our demo, but lets talk about the others now. The flag variable can be one of three things:

- ONLY_CREATE You will be using this one when you want your current system to generate an effect when a 'cause' is triggered. For our demo, for instance, we use it when we generate an explosion, so when the 'e' key is pressed, this is what we call:
  `ps1.Update(time, ONLY_CREATE, 500);`
  This way, we can generate an explosion of 500 particles when the 'e' key is pressed, and we don't have to update the whole particle system if we don't want to (that would make the user hold down the 'e' key to see the actual animation).

- ONLY_UPDATE This is what we are going to use for systems that have effects generated dynamically (such as the explosion effect discussed above), instead of effects that need to be generated constantly (such as

fire).

- UPDATE_AND_CREATE This is the flag that we want to use when we want to generate dynamic effects such as fire, and keep the effect constant, refer to this example (which was taken from our demo source):
  `ps3.Update(time, UPDATE_AND_CREATE, 5);`
  This will create five particles every time it is called, and add it to the current effect (in this case fire), and also update the system.

Now, back to the function.

```
// We Need To Calculate The Elapsed Time
time_counter= (time-last_update);
```

This calculates the time that has elapsed since the last time the function is called.

```
if(flag== ONLY_UPDATE || flag== UPDATE_AND_CREATE)
	{
	// Set The Time Of The Last Update To Now
	last_update=time;

	// Clear The Particle Counter Variable Before Counting
	particles_numb_alive=0;

	// Update All Particles
	for(loop=0; loop<MAX_PARTICLES; loop++)
	 {
	 if(particle[loop].age>=DEATH_AGE)
		{
		if(particle[loop].Update(time_counter))
			particles_numb_alive++;
		}
	 }
	}
```

This calls the particle's update function (if the correct flag was included), and checks to see if the particle is currently alive or not, and if it, then we add a number to the total particles that are currently alive.

```
if(flag== ONLY_CREATE || flag== UPDATE_AND_CREATE)
	{
	// Now Calculate How Many Particles We Should Create Based On
	// Time And Taking The Previous Frame's Emission Residue Into
	// Account.
	particles_needed+= particles_per_sec*time_counter+emission_residue;

	// Now, Taking The Previous Line Into Account, We Now Cast
	// particles_needed Into An Unsigned Int, So That We Aren't
	// Going To Try To Create Half Of A Particle Or Something.
	particles_created= (unsigned int) particles_needed;

	if(!stopped)
	 {
	 // This Will Remember The Difference Between How Many We Wanted
	 // To Create, And How Many We Actually Did Create.  Doing It
	 // This Way, We Aren't Going To Lose Any Accuracy.
	 emission_residue= particles_needed-particles_created;
	 }
```

```
        else
         {
         emission_residue= particles_needed;
         particles_created=0;
         }

        // Lets Make Sure That We Actually Have A Particle To Create
        if(particles_created<1)
         {
         prev_location.vertex[0]=location.vertex[0];
         prev_location.vertex[1]=location.vertex[1];
         prev_location.vertex[2]=location.vertex[2];
         return true;
         }

        for(loop=0; loop<MAX_PARTICLES; loop++)
         {
         // If We Have Created Enough Particles To Satisfy The Needed
         // Amount, Then This Value Will Be Zero, And Will Quit This
         // Loop.
         if(!particles_created)
                 break;

         // If The Age Of This Particles Is -1.0, Then This Particle
         if(particle[loop].age<DEATH_AGE)
                 {
                 particle[loop].Create(this, time_counter);

                 // Now We Decrease The Amount Of Particles We Need To Create
                 // By One.
                 particles_created--;
                 }
         }
        }
return true;
}
```

This creates the necessary amount of particles that the system wants to create at certain times. We also check to see if the particle's state is set to stopped or not. When the system is stopped, the system builds up the number of particles that need to be created the next time around. For an example of this, when the fire is going in the included demo, hold down the 'X' button for a while, and when you're ready, move the mouse, and let go (so you don't quit the program, just freeze it), you will see a large ball of particles rise up, this is from the system building up the number of particles that need to be created. Then, once it is done seeing how many particles should be created, we go through the system's number of particles, and search for a particle that is open to be "messed with." We then call that particle's creation function. Check this line out:

```
particle[loop].Create(this, time_counter);
```

In case you aren't familiar with much C++, this sends the current class as an argument to something (in this case we assigning this class as the particle's parent).

And, our last function to discuss for the day is the particle system's render function, shown here:

```
void PARTICLE_SYSTEM::
        Render(void)
        {
```

```
        int loop;
        float size;
        VERTEX3D vert;

        glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_PIXEL_MODE_BIT);
        glDisable(GL_DEPTH_TEST);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE);
        glEnable(GL_BLEND);
```

The previous lines push the current attributes set (that are related to the depth buffer, and pixel mode **only**) onto the attribute stack for later retrieval, then we turn off the depth buffer, and turn on color keying.

```
        glBindTexture(GL_TEXTURE_2D, texture);

        for(loop=0; loop<MAX_PARTICLES; loop++)
         {
         size=particle[loop].size/2;
         particle[loop].color[4]=particle[loop].alpha;

         if(particle[loop].age>=DEATH_AGE)
                 {
                 glColor4fv(particle[loop].color);
```

Now we are going to create the particle's four vertices. You may be asking why we need to have a vertex object to store things. First of all, the call to glVertex3f with all three calculations for the coordinates would look pretty bloated. Second, the vector version of glVertex3f (glVertex3fv in case you didn't know) is a little bit faster than the non-vector version.

```
                 glBegin(GL_TRIANGLE_STRIP);

                         vert.vertex[0]=particle[loop].location.vertex[0]-size;
                         vert.vertex[1]=particle[loop].location.vertex[1]+size;
                         vert.vertex[2]=particle[loop].location.vertex[2];

                     glTexCoord2d(1,1); glVertex3fv(vert.vertex);

                         vert.vertex[0]=particle[loop].location.vertex[0]+size;
                         vert.vertex[1]=particle[loop].location.vertex[1]+size;
                         vert.vertex[2]=particle[loop].location.vertex[2];

                         glTexCoord2d(0,1); glVertex3fv(vert.vertex);

                         vert.vertex[0]=particle[loop].location.vertex[0]-size;
                         vert.vertex[1]=particle[loop].location.vertex[1]-size;
                         vert.vertex[2]=particle[loop].location.vertex[2];

                         glTexCoord2d(1,0); glVertex3fv(vert.vertex);

                         vert.vertex[0]=particle[loop].location.vertex[0]+size;
                         vert.vertex[1]=particle[loop].location.vertex[1]-size;
                         vert.vertex[2]=particle[loop].location.vertex[2];

                         glTexCoord2d(0,0); glVertex3fv(vert.vertex);

                 glEnd();
                 }
         }
```

```
        glPopAttrib();
        }
```

Lastly, we retrieve the attributes that we pushed onto the stack, and that's it for the render function!

Using the Particle Engine:

As I said before, there are a lot of functions that we didn't go over. Most of them are pretty self-explanatory, and if you are confused about one, just check out the source in Particle_System.cpp. Now, I am going to show you how to create three effects with those functions. First of all, lets create an explosion!

```
ps1.Set_Alpha(1.0f, 0.0f);
ps1.Set_Velocity(1.0f, 1.0f, 1.0f);
ps1.Set_Texture(&flare);
ps1.Set_Angle(360);
ps1.Set_Attraction(0);
ps1.Set_Color(1.0f, 0.0f, 0.0f,   1.0f, 0.55f, 0.1f);
ps1.Set_Gravity(0.0f, 0.0f, 0.0f);
ps1.Set_Life(4);
ps1.Set_Location(0.0f, 0.0f, 0.0f);
ps1.Set_ParticlesPerSec(0);
ps1.Set_Size(3, 2);
ps1.Set_Speed(10);
ps1.Set_Spread(-100, 100, 200);
ps1.Set_SystemFlag(STOP, SK_OFF);
ps1.Set_SystemFlag(ATTRACTION, SK_OFF);
```

Ok, first we set the texture that we want (when we get into mesh loading, we will make some mesh loading functions for the particle engine too). We then set the angle that we want the particles to be able to travel, which in this case is a full 360° (on all axes). We also don't want attraction at all, so we turn it off, and set the attraction percentage to 0 (you don't have to do this, but its better for clarity reasons). We also don't need any "particles per second" since we specify the number that we want to create when we trigger the effect. I think the explosion looked better if I increased the size, and it slowly dropped a little, so I set the size to start at 3, and end at 2. I also wanted the explosion to be halfway fast, so I set the speed a little high. And, I set the spread (the allowed distance for particles to spread from the emitter) to AT MOST -.5f, and .5f away. I also turned off 'stoppage' and the attraction effects.

That's all there is to using the engine, I also made a few more effects in the demo, so check them out. One last note, in particle.h, there is a define for MAX_PARTICLES, I set it to be 2000 particles per system. You can change this to whatever you want, but remember, speed is key in games, and the more particles you use, the better things will look, but particles are a slight speed hog (I went overboard on the particles I used in the demo, you should keep the max **at most** to about 1000).

The Demo Controls:

- 'f' to toggle fire on and off
- 'e' for a nice explosion
- 'a' for an explosion where the particles are attracted to the emitter

Conclusion:

There was some nice graphics for you, instead of the boring math theory, and wrapper explanations. In case you didn't notice, we are slowly turning our wrapper into an engine, and this is the engine that we will be using when we start to make our games. We can't make any games with it until we get the wrapper/engine into nice working order though (sound, collision detection, mesh loading/rendering, etc). So, that is what we are building up to.

I was mowing one day this week, and the inspiration for the first game we will make came into my head (and this

isn't going to be a simple game, its going to cover about 4-6 tutorials), that will incorporate **everything** that we will be talking about... So it will be some very cool stuff.

For this tutorial, I would like once again to thank NeHe, and you should too. :) Also, I would like to think Richard 'Keebler' Benson for being a particle expert, and Dave and Kevin of Gamedev.net for helping me out in their own unique ways (though they may not have realized it ;)). Thanks guys!

Bibliography:

1. Reeves, William T., "Particle Systems-A Technique for Modeling a Class of Fuzzy Objects," ACM Transactions on Graphics, vol. 2, no. 2, pp. 91-108, April 1983.

2. Benson, Richard. "Particle Chamber." (http://www.dxcplusplus.co.uk/DemoVault/ParticleChamber.zip)

DOWNLOADS:

* DOWNLOAD the Visual C++ Code For This Lesson.

Back To NeHe Productions!

Game Programming Lesson 4

OpenGL Game Programming Tutorial Four:
Model Mania

By Trent "**ShiningKnight**" Polack

Introduction:

In this tutorial we are going to be loading two different formats of models: .md2 (Quake 2 model type) and .ms3d (MilkShape's native format). Each model has it's own pros and cons, but each has a nice purpose in our engine. First, the .md2 format is perfect for First Person Shooters (being that it was used in Quake 2), and that is what our engine is going to be optimized for (though, the engine is going to be designed in a way that can be used for any game). The .ms3d format is good because MilkShape has support for the most popular model formats around, so it is the perfect medium for model conversions. The .md2 format is the more complicated of the two models formats, so lets get that over with first.

The .MD2 Model Format:

Ok, we are basically going to need four different structures to read it in: one for the file's header, one for vertex information, frame, and another vertex structure. Since the header comes first in absolutely any file, lets discuss that first, so heres the structure:

```
typedef struct MD2_HEADER_TYP
{
	int magic;
	int version;
	int skinWidth;
	int skinHeight;
	int frameSize;
	int numSkins;
	int numVertices;
	int numTexcoords;
	int numTriangles;
	int numGlCommands;
	int numFrames;
	int offsetSkins;
	int offsetTexcoords;
	int offsetTriangles;
	int offsetFrames;
```

```
        int offsetGlCommands;
        int offsetEnd;
} MD2_HEADER, *MD2_HEADER_PTR;
```

Now, lets go through each component one by one, to see what it is, and what purpose it serves us.

- magic - The 'magic' number that tells us that the file really is an .md2, this number must be 844121161 if it really is a valid .md2.
- version - The version number of the file, this should always be 8.
- skinWidth - Tells us the width of the model's skin (not used in our code).
- skinHeight - Tells us the height of the model's skin (not used in our code).
- frameSize - Tells us the size of each frame of the model (more about this, and other frame information later).
- numSkins - This tells us the number of skins the model has (not used in our code).
- numVertices - This tells us how many vertices the model has... This is really only useful if you want to display the number of vertices to the screen.
- numTexcoords - This tells us how many texture coordinates the model has, the weird part about this is that it isn't necessarily the same as the number of vertices.
- numTriangles - Tells us how many triangles the model has (we use this quite a bit).
- numGlCommands - This tells us if the model was optimized for triangle strips, or fans. Not many .md2 loaders utilize this (but of course, ours does), but it's a HUGE performance gain if used correctly.
- numFrames - The number of animation keyframes that the model has.

The following parts of the header are all offsets to the spot in the file where the information you may be looking for is:

- offsetSkins
- offsetTexcoords
- offsetTriangles
- offsetFrames
- offsetGlCommands
- offsetEnd

Ok, and what we are going to need to do is read the header in first when we load our model (you will see code to this later), as it tells us how much memory we may need for certain variables, and the way we are going to render the information. Now we are going to talk about a 'unique' structure to say the least... Unique, in that half of the variables in this structure are only used to hold information that we don't need. The light normal index is an index to a table of normals used in Quake 2... In short, don't worry about it, we are only using it as a 'decoy' variable (my word for variables that we use to stuff information into just so we can continue on in the file).

```
typedef struct MD2_VERTEX_TYP
{
        unsigned char vertex[3];
        unsigned char lightNormalIndex;
} MD2_VERTEX, *MD2_VERTEX_PTR;
```

We will be using the vertex information for each frame though, so keep in mind that half of this structure is useful. Now, lets talk about frames that each .md2 holds internally. Each .md2 has 198 frames of animation stored inside it, for animations such as running, crouching, dying, and idling. These are all very useful for first person shooters (which is why they were used in Quake 2... Ok, I won't say that line again, I promise). Here is the structure:

```
typedef struct MD2_FRAME_TYP
{
        float       scale[3];
        float       translate[3];
        char        name[16];
        MD2_VERTEX vertices[1];
} MD2_FRAME, *MD2_FRAME_PTR;
```

As you may be able to see, unless you're blind (then I don't understand how or why you are 'reading' this at all), this structure contains information so that you can scale your models down (as, they are quite large if you don't), and so you can translate them. Finally, the last structure that we need to talk about is the REAL vertex structure that you will be using... uhhhhh, nearly constantly. ;) Here it is:

```
typedef struct MD2_MODELVERTEX_TYP
{
	float x,y,z;
	float u,v;
} MD2_MODELVERTEX, *MD2_MODELVERTEX_PTR;
```

As you can see, this structure stores a vertex for OpenGL (the (x,y,z) ordered triple), and the texture information (the (u,v) ordered double). We will be using this structure quite a bit, being that each model has quite a bit of vertices (which is why you don't want to hard code the entire thing, one screw up, and well, your wife and kids won't see you for a week).

Now, let me show you the class that we will be using for our .md2 models (you cannot understand my growing love for classes... C programmers take note: classes are the best):

```
class MD2
{
	private:
	 MD2_MODELVERTEX vertList[100];

	 int numGlCommands;
	 long* glCommands;

	 int numTriangles;

	public:
	 int stateStart;
	 int stateEnd;

	 int frameSize;
	 int numFrames;
	 char* frames;

	 int currentFrame;
	 int nextFrame;
	 int endFrame;
	 float interpolation;

	bool Load(char* filename);
	void Render(int numFrame);
	void Animate(int startFrame, int EndFrame, float Interpolation);
	void SetState(int state);

	MD2() : stateStart(IDLE1_START), stateEnd(IDLE1_END),
		numGlCommands(0), frameSize(0), numFrames(0),
		currentFrame(IDLE1_START), nextFrame(currentFrame+1),
		endFrame(IDLE1_END), interpolation(0.0f)
	{	}

	~MD2()
	{
```

```
        if(glCommands)
                delete [] glCommands;
        if(frames)
                delete [] frames;
       }
};
```

That's it, for now, just concentrate on the loading, and rendering functions, we will get to the animation/state functions a little later. You may be wondering why nearly every variable is a pointer, well, using dynamic allocations are a nice memory saver as opposed to creating the maximum amount of memory from the start, and since I will be using the C++ way of allocating memory (its just cleaner that way), I'll show you a quick example. Say we want to load the glCommands variable, well, here is how we'd go about doing it:

```
glCommands= new long [header.numGlCommands*sizeof(long)];
```

That's all there is to it. Lets go through it one by one, 'new' is the C++ way of allocating memory, the form you need to use it in is this: variablePtr= new variableType [sizeYouWantToCreate]; Easy eh? And to delete glCommands you just have to do this:

```
delete [] glCommands;
```

Tough one eh? If you still don't get it, then go pick up a C++ book slacker. :)

Now, on to the loading code:

```
bool MD2::
       Load(char* filename)
       {
       FILE* file;
       MD2_HEADER header;
```

We create a file pointer to store the information while we read it in dynamically, and we are also creating a temporary MD2_HEADER to store our header information in (so we don't have to waste the space for a structure that we won't use inside our class).

```
if((file= fopen(filename, "rb"))==NULL)
       {
       S3Dlog.Output("Could not load %s correctly", filename);
       return false;
       }
```

This loads the file that was specified by filename. And if there was a problem opening it, then we get out of the functions (oh, and in case you were wondering, if we didn't get out of the function after an error, you would be facing a nice little "illegal operations" message box in front of your eyes when you ran the program.

```
fread(&header, sizeof(MD2_HEADER), 1, file);
```

This loads the header into our previously created header structure, we will be using this header **a lot** in this function, but we only need it for this function, so once again, that's why we made it specific to the function. We will need to make a few of the header's variables global (but only 2-3), but that's still less space than storing the whole thing.

```
if(header.magic!= 844121161)
{
       S3Dlog.Output("%s is not a valid .md2", filename);
       return false;
}

if(header.version!=8)
```

```
{
	S3Dlog.Output("%s is not a valid .md2", filename);
	return false;
}
```
This time, we are checking two of the header's variables that are specific to the type of file that we are loading. If either of those two if statements returns false, then we have a problem because **ALL** .md2's have should have the same magic and version number.

```
frames= new char[header.frameSize*header.numFrames];
```
As I was saying earlier about dynamic memory allocations, this is one of the cases where we apply it. While programming, we don't know the exact number of frames we are going to need, so we need to do it while the program is running. This fills frames with the correct number of frames (and we multiply that by header.frameSize so that the variable has adequate amounts of memory).

```
if(frames==NULL)
	return false;
```
This just checks to make sure that the memory was allocated successfully (and its rare that it isn't allocated right). Now that we have enough room in frames, we can fill it with the model-specific information:

```
fseek(file, header.offsetFrames, SEEK_SET);
fread(frames, header.frameSize*header.numFrames, 1, file);
```
First we advance to the spot in the file where the frame information is, then after that, we fill our frames variable with the information.

```
glCommands= new long [header.numGlCommands*sizeof(long)];

if(glCommands==NULL)
	return false;

fseek(file, header.offsetGlCommands, SEEK_SET);
fread(glCommands, header.numGlCommands*sizeof(long), 1, file);
```
This is the same case as before, except this time we are loading the glCommands information (to tell us whether or not we need triangle strips or fans to render our model).

```
numFrames	= header.numFrames;
numGlCommands	= header.numGlCommands;
frameSize	= header.frameSize;
numTriangles	= header.numTriangles;
```
This stores the global header information that we need to know for rendering inside our class.

```
fclose(file);

S3Dlog.Output("Loaded %s correctly", filename);
return true;
}
```
Now, lets close the file that we had open, output the success information to the log, and get out of the function!!! Phew, that was a pretty complicated function, especially if you've never had any experience with loading in files, and allocating memory dynamically. Now, on to the rendering function:

```
void MD2::
	Render(int numFrame)
	{
static MD2_MODELVERTEX vertList[100];
```

```
MD2_FRAME_PTR   currentFrame;
       VERTEX   v1;
       long*    command;
       float    texcoord[2];
       int      loop;
       int      vertIndex;
       int      type;
       int      numVertex;
       int      index;
```

This defines the variables that we will need throughout the rendering function. We have one frame variable, to tell us the current frame (since each model can only render one frame at a time), and then we have our four temporary vertices for outputting a vertex as a vector rather than an ordered triple (a very, very slight speed increase). Also, just a little tidbit, I had vertList as a temporary variable that was created every time the function was called, instead of being static... Then as I was writing this, I realized that creating a 100 structures every call may be a little taxing, I made it static (in case you are rusty, that means that once its created, it lasts the rest of the file), and I was getting about 10-15 frames more per second, not a huge increase, but its nice to have available.

```
currentFrame= (MD2_FRAME*) ((char*)frames+frameSize*numFrame);
command      = glCommands;
```

This sets the current frame information, and the glCommand that we are going to need when we output our vertices to OpenGL.

```
while((*command)!=0)
{
       if(*command>0)           // This Is A Triangle Strip
       {
        numVertex= *command;
        command++;
        type= 0;
       }
       else                     // This Is A Triangle Fan
       {
        numVertex= - *command;
        command++;
        type= 1;
       }
```

This sets up our vertices for rendering, and also the command variable for, once again, rendering our vertices as a tri strip or fan.

```
if(numVertex<0)
       numVertex= -numVertex;
```

This ensures that the number of vertices didn't get 'negatized' (new word) somewhere, since you can never ever have a negative number of vertices in a model, we make sure that its not going to be negative.

```
for(loop=0; loop<numVertex; loop++)
{
       vertList[index].u= *((float*)command);
       command++;
       vertList[index].v= *((float*)command);
       command++;

       vertIndex= *command;
```

```
        command++;

        vertList[index].x= ( (currentFrame->vertices[vertIndex].vertex[0]*
                           currentFrame->scale[0])+
                           currentFrame->translate[0]);
        vertList[index].z= -((currentFrame-> vertices[vertIndex].vertex[1]*
                           currentFrame->scale[1])+
                           currentFrame->translate[1]);
        vertList[index].y= ( (currentFrame->vertices[vertIndex].vertex[2]*
                           currentFrame->scale[2])+
                           currentFrame->translate[2]);
        index++;
}
```

A decently sized snippet of code, but don't let it intimidate you, because all we are doing is filling our vertex list with the correct vertex positions after scaling and translating them. See, this is one time when size really doesn't matter, but does a lot to screw with your head.

```
        if(type==0)
        {
                glBegin(GL_TRIANGLE_STRIP);
                for(loop=0; loop<index; loop++)
                {
                        v1.vertex[0]=(vertList[loop].x);
                        v1.vertex[1]=(vertList[loop].y);
                        v1.vertex[2]=(vertList[loop].z);

                        texcoord[0]= vertList[loop].u;
                        texcoord[1]= vertList[loop].v;

                        glTexCoord2fv(texcoord);
                        glVertex3fv(v1.vertex);
                }
                glEnd();
        }
        else
        {
                glBegin(GL_TRIANGLE_FAN);
                for(loop=0; loop<index; loop++)
                {
                        v1.vertex[0]=(vertList[loop].x);
                        v1.vertex[1]=(vertList[loop].y);
                        v1.vertex[2]=(vertList[loop].z);

                        texcoord[0]= vertList[loop].u;
                        texcoord[1]= vertList[loop].v;

                        glTexCoord2fv(texcoord);
                        v1.SendToOGL();
                }
                glEnd();
        }
       }
}
```

Now, that renders the vertices according to the command that they were made for. Nothing too hard! As you may see, I included a function in the VERTEX class for sending the array that the VERTEX class represents to OpenGL as a 3D vertex. I entitled it "SendToOGL" makes sense, doesn't it?

Ok, now, about the state and animation functions that you saw earlier. I have made a list of predefined constants for your use with .md2's... You can see them all in the table below:

IDLE1	The first of four idle animations
RUN	Animation for the model running. RUN FORREST RUN!
SHOT_STAND	Animation for when the model gets shot, but stays standing
SHOT_SHOULDER	Animation for when the model gets shot in the shoulder (still standing though)
JUMP	Animation for the model jumping
IDLE2	The second of four idle animations
SHOT_FALLDOWN	Animation for the model getting shot, and falling to the ground (used for getting shot by big weapons)
IDLE3	The third of four idle animations
IDLE4	The fourth of four idle animations
CROUCH	Animation for making the model crouch
CROUCH_CRAWL	Having the model crawl while crouching
CROUCH_IDLE	An idle animation while in a crouching position
CROUCH_DEATH	The model dying while in a crouching position
DEATH_FALLBACK	The model dying while falling backwards (death shot from the front)
DEATH_FALLFORWARD	The model dying while falling forwards (death shot from the back)
DEATH_FALLBACKSLOW	The model dying while falling backwards slowly

If you think that was a long list, think about how long it took me to analyze the model to see what it was doing, and what frame it was in... Can we say 'boring?' Ok, you know the states, "whoop dee frickin' doo bob, now what can I do with them?" Well, I made a nice little function (I won't show the code, its very redundant, so check it out for yourself in Loaders.cpp. But, say you have a model class object named, none other than, 'model.' And you want to put it into a running state, here is how you would do it:

```
model.SetState(RUN);
```

Well, while that line doesn't actually produce any visible results, it sets two variables inside the 'model' class to what they need to be to produce a running animation. And for you to be actually able to see the animation that it just performed, you need to use the Animate(...) function that I created (the idea came from Kevin Hawkins/Dave Astle [1], but not the code itself). Here is the line that would allow you to actually see the animation (once again, you can see the actual function code for yourself in Loaders.cpp, as the code is pretty self-explanatory, and space is precious):

```
model.Animate(model.stateStart, model.stateEnd, 0.02f);
```

That function takes the class's stateStart (the frame number of the starting state), and stateEnd and it will interpolate the model's animation at a speed of once every 50 frames (this is a great spot for you to use movement based on timing). Here is another example of using those functions, with a model's included weapon (use the actual model's information to animate the weapon).

```
glBindTexture(GL_TEXTURE_2D, hobgoblinSkin.ID);
glScalef(0.05f, 0.05f, 0.05f);
hobgoblin.Animate(hobgoblin.stateStart, hobgoblin.stateEnd, 0.02f);
glBindTexture(GL_TEXTURE_2D, hobgoblinWeaponSkin.ID);
hobgoblinWeapon.Animate(hobgoblin.stateStart, hobgoblin.stateEnd, 0.02f);
```

You need to load the model's skins (textures) in for that code though, you can check out a lot of it in main.cpp. Here is what you have done in the last 10 pages:

Amazing isn't it!? And we're not even done yet (which could be viewed as a good or a bad thing).

The .MS3D Model Format:

About a week ago, I didn't even plan on having this model format in my engine, I was just going to use .md2's, and be happy with it. But one night when I was trying to export an .md2, I found out that I needed to make the model have joints, and animations. And that's all good and well, except the model I needed didn't need to have any animations, it just needed to be a static (non-animated) object. Well, I just figured that I would write a simple .MS3D loading class, and that 'simple' class took me about 6 hours. :) But here it is in all of it's simplistic glory (if by chance, you are reading this, and think the class needs some more features, drop me a line (references at bottom), and we'll talk about it). First, the necessary file structures:

```
typedef struct MS3D_HEADER_TYP
{
        char id[10];
       int  version;
} MS3D_HEADER, *MS3D_HEADER_PTR;
```

Just like last time, we will be reading the header in first, except this time, the file is structured differently, and things to need to be read in perfect order for things to work! This header only has two variables for checking to make sure that the file is a .MS3D format:

- id - Should always be MS3D000000
- version - Should always be 3 or 4

Next is the vertex structure:

```
typedef struct MS3D_VERTEX_TYP
{
	unsigned char flags;
	unsigned char refCount;
	char boneID;
	float vertex[3];
} MS3D_VERTEX, *MS3D_VERTEX_PTR;
```

This is another class that has some 'decoy' variables. The boneID variable is used for skeletal animation, which is like the .MD2's animation frames, except the model has joints (just like us) that you can adjust dynamically... A great feature to say the least, but requires a large amount of hard work and research. I only know of two people who have researched it, one of which being Brett Porter (http://rsn.gamedev.net).

```
typedef struct MS3D_TRIANGLE_TYP
{
	unsigned short flags;
	unsigned short vertexIndices[3];
	float vertexNormals[3][3];
	float u[3];
	float v[3];
	unsigned char smoothingGroup;
	unsigned char groupIndex;
} MS3D_TRIANGLE, *MS3D_TRIANGLE_PTR;
```

That is our triangle class, all .MS3D (along with .MD2) files are ONLY composed of triangles, as they are the best primitive to use speed-wise in games. This is the main structure that we will be using for rendering.

```
typedef struct MS3D_GROUP_TYP
{
	 unsigned char   flags;
	 char name[32];
	 unsigned short  numTriangles;
	unsigned short*  triangleIndices;
	char materialIndex;
} MS3D_GROUP, *MS3D_GROUP_PTR;
```

Also, in MilkShape, you can choose to split the model into 'groups' of polygons. The model I am using for this demo has about 11 groups. Check it out!

Now, lets check out the .MS3D class:

```
class MS3D
{
	public:
	 unsigned short  numVertices;
	 MS3D_VERTEX*    vertices;
	 unsigned short  numTriangles;
	 MS3D_TRIANGLE*  triangles;
	 unsigned short  numGroups;
	 MS3D_GROUP*     groups;

	bool Load(char* filename);
	void Render(void);

	MS3D()
	{	}
```

```
	~MS3D()
	{
	 if(vertices)
		delete vertices;
	 if(triangles)
		delete triangles;
	 if(groups)
		delete groups;
	}
};
```

And that's our incredibly simple .MS3D class! As for variables, we are storing 3 variables to hold the amount of structures, and then we have a pointer to each structure that we will dynamically allocate memory for while loading. Speaking of loading, lets check that function out!

```
bool MS3D::
	Load(char* filename)
	{
	FILE* file;
	MS3D_HEADER header;
	int loop;
```

Ok, nothing should look unfamiliar to you, we just went through most of this with the .MD2 loading code! So it should seem partly like déjà vu!

```
if((file= fopen(filename, "rb"))==NULL)
{
	S3Dlog.Output("Could not load %s correctly", filename);
	return false;
}
```

We open the file for reading, and check to make sure that it actually loaded.

```
fread(&header.id, sizeof(char), 10, file);
fread(&header.version, 1, sizeof(int), file);
```

We read both variables of the header in (loaded separately, mainly because it didn't seem to want to work the normal way).

```
if(strncmp(header.id, "MS3D000000", 10)!=0)
{
	S3Dlog.Output("%s if not a valid .ms3d", filename);
	return false;
}

if(header.version!=3 && header.version!=4)
{
	S3Dlog.Output("%s if not a valid .ms3d", filename);
	return false;
}
```

This checks to make sure that the file we loaded in is **really** a valid .MS3D file.

```
fread(&numVertices, sizeof(unsigned short), 1, file);
vertices= new MS3D_VERTEX [numVertices];
for(loop=0; loop<numVertices; loop++)
{
```

```
        fread(&vertices[loop].flags,     sizeof(BYTE),  1, file);
        fread( vertices[loop].vertex,    sizeof(float), 3, file);
        fread(&vertices[loop].boneID,    sizeof(char),  1, file);
        fread(&vertices[loop].refCount, sizeof(BYTE),  1, file);
}
```

First we read in the number of vertices for this model, then we allocate the memory for the exact number of vertices. Next we fill the vertices with the information from the model file.

```
fread(&numTriangles, sizeof(unsigned short), 1, file);
triangles= new MS3D_TRIANGLE [numTriangles];
for(loop=0; loop<numTriangles; loop++)
{
        fread(&triangles[loop].flags,            sizeof(unsigned short), 1, file);
        fread( triangles[loop].vertexIndices,    sizeof(unsigned short), 3, file);
        fread( triangles[loop].vertexNormals[0],sizeof(float),           3, file);
        fread( triangles[loop].vertexNormals[1],sizeof(float),           3, file);
        fread( triangles[loop].vertexNormals[2],sizeof(float),           3, file);
        fread( triangles[loop].u,                sizeof(float),          3, file);
        fread( triangles[loop].v,                sizeof(float),          3, file);
        fread(&triangles[loop].smoothingGroup,   sizeof(unsigned char),  1, file);
        fread(&triangles[loop].groupIndex,       sizeof(unsigned char),  1, file);
}
```

Same thing as before, except this time we get the number of triangles the model has, allocate the memory, then fill it with information. Now, we read the groups in the same exact way, and get out of the function:

```
        fread(&numGroups, sizeof(unsigned short), 1, file);
        groups= new MS3D_GROUP [numGroups];
        for(loop=0; loop<numGroups; loop++)
        {
         fread(&groups[loop].flags,       sizeof(unsigned char), 1,  file);
         fread( groups[loop].name,        sizeof(char),          32, file);
         fread(&groups[loop].numTriangles,sizeof(unsigned short),1,  file);

         groups[loop].triangleIndices=new unsigned short [groups[loop].numTriangles];

         fread( groups[loop].triangleIndices, sizeof(unsigned short),
groups[loop].numTriangles,file);
         fread(&groups[loop].materialIndex,   sizeof(char), 1, file);
        }

        S3Dlog.Output("Loaded %s correctly", filename);
        return true;
}
```

Well kiddos, looks like its time to get rendery (for you uninitiated people, that's SK talk for "lets talk about rendering")! What I am about to show you is the entire function that will render your .ms3d model. It's pretty simple, so as an exercise in learning, I am going to make you figure it out. Here it is, learn it, love it, remember it:

```
void MS3D::
        Render(void)
        {
        int loop1;
        int loop2;
        int loop3;
```

```
	// Draw By Group
	for(loop1=0; loop1<numGroups; loop1++ )
	 {
		// Draw As Regular Triangles, Since .MS3D's Aren't Optimized Like
.MD2's
		glBegin(GL_TRIANGLES);
		for(loop2=0; loop2<groups[loop1].numTriangles; loop2++)
		{
			int triangleIndex	=
groups[loop1].triangleIndices[loop2];
			const MS3D_TRIANGLE* tri= &triangles[triangleIndex];

			// Loop Through The Triangle's Vertices, And Output Them!
			for(loop3=0; loop3<3; loop3++)
			{
				int index= tri->vertexIndices[loop3];

				glNormal3fv( tri->vertexNormals[loop3]);
				glTexCoord2f(tri->u[loop3], tri->v[loop3]);
				glVertex3fv(vertices[index].vertex);
			}
		}
		glEnd();
	 }
	}
```

Right now, you may be wondering why I didn't explain every part of it like I normally do... Well, there is a simple, and well thought-out, reason for why I do it. The point of this is to be a TUTORIAL, and I find that if everything is explained to a reader, then that reader doesn't learn nearly as much as if they figured it out themselves. So in short, don't rely on other people entirely for your learning (same reason you always need to have multiple sources of information for a large paper).

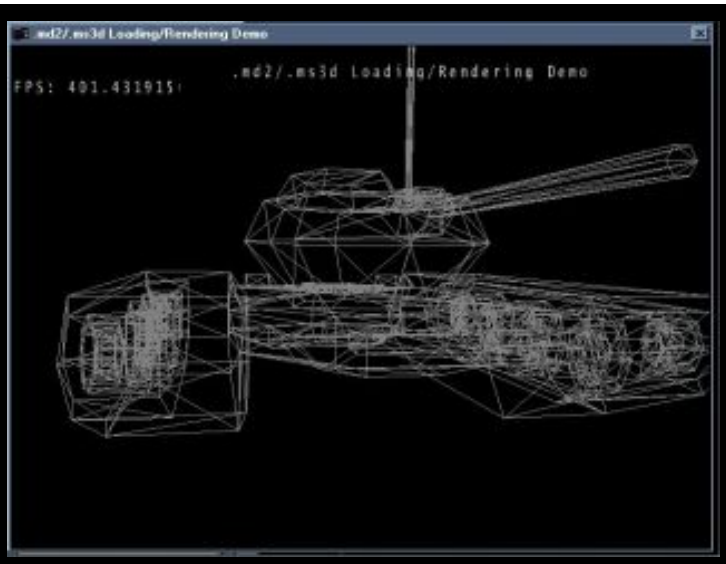
Ok, enough of that little 'sidetrack.' Here is a little section of code that will draw a model for us. First you load it:

```
tank.Load("Models/tank1/tank.ms3d");
```

Then you render it:

```
glDisable(GL_TEXTURE_2D);
glColor4f(0.5f, 0.5f, 0.5f, 1.0f);
glScalef(0.1f, 0.1f, 0.1f);
tank.Render();
```

That's all there is to it! Now, normally to render a model, you would want to have a skin for it, and use texturing, but since I'm far from being artistically-inclined, my skin would've be pathetic. So, I think the tank looks better in wireframe mode! Here is the fruit of your creations:



Beautiful, isn't it? Well, that's all there is to loading .MS3D's!!!

The Demo:

This tutorial's source makes a demo that displays two .md2's and one .ms3d. Here are the controls this time:

```
'1' - Change To Alien .MD2 Model
'2' - Change To Hobgoblin .MD2 Model
'3' - Change To Tank .MS3D Model
'W' - Turn To Aliased Wireframe Mode
'A' - Turn To Antialiased Wireframe Mode (SLOW)
'T' - Turn To Textured/Filled Mode
'R' - To Make One Of The .MD2s Run
```

And use the arrow keys to rotate the model.

Engine Updates:

Being that this engine is a **VERY** important tool for our game coding needs, I randomly look through files, finding errors, changing code, and sometimes completely changing some things all together. I have changed the TGA loader into a general TEXTURE class that now has TGA and BMP loading support. I am also in the process of designing another implementation for a particle engine (using inheritance this time), so that you can choose your favorite method... This new implementation should be ready by the 6th tut.

Conclusion:

Well, loading models can add a completely new look and feel to your game. And in the next tutorial, we will be adding a completely new feel and sound to your games. I have been on a coding 'high' for the past week, and have been coding every spare moment of my time. I also had a really great idea for a tutorial for our intersection testing tutorial (2 tuts away)! Not only is it going to be fun, but you'll learn a lot from it too.

Ok, in this tutorial the thanks go out to: cmx (for modeling the incredible tank), Reed Hawker (for designing the alien, and skin at http://www.polycount.com), Hunter, Burnt Kona, Fafner, and Deranged (for a collective effort on designing the hobgoblin. Also at http://www.polycount.com), and finally I would like to thank Justin "BlackScar" Eslinger for help with .md2 loading.

DOWNLOADS:

* DOWNLOAD the Visual C++ Code For This Lesson.

Back To NeHe Productions!