OpenGL

Jesus Ramos

Panther Linux Users Group

What is OpenGL?

OpenGL (Open Graphics Library) is just that... An open standard (and open source) graphics library used for drawing to the screen.

OpenGL provides an interface that is OS independent for drawing to the screen and is supported by just about every major graphics card vendor out there (to varying degrees though).

Unlike other libraries such as XNA, SDL, AWT, and SWING, OpenGL gives you more control over what's being drawn but comes at the cost of managing more than you normally have to.

Why Use OpenGL?

If XNA can get the job done easily why shouldn't I just use it?

OpenGL has a lot of advantages over other libraries (even D3D / DirectX):

- Cross-Platform
- Faster draw cycles
- OS-independent wrappers for window management
- Easy to implement matrix transforms and linear algebra functions
- Provides easy to program C-like GLSL (Graphics Library Shading Language)

GLUT (Graphics Library Utility Toolkit)

Managing windows and keystrokes is OS-dependent but luckily for us there's GLUT. GLUT allows us to do generic things such as creating windows or setting keyboard handlers and specify OS-independent handlers for that.

No need to worry about process ID's or setting up interrupt handlers you just tell GLUT what your functions are for handling those things and it will go ahead and call those functions when appropriate.

GLEW (OpenGL Extension Wrangler)

Unfortunately for us not all graphics cards and computers are created equal. Some cards support more or less than others. GLEW allows us to easily check what features are supported by a card so that we can disable or use different features if we can as the features we may attempt to use are not implemented on the graphics hardware.

A common example of this is various levels of shader support, extension support for optimized drawing, etc...

Vertices

Graphics cards are in reality stupid pieces of hardware. They can only draw triangles, they can only use float precision (4 bytes) and they are just mountains of hardware to do the same thing on different data really fast and tons of it at the same time.

Rather than telling the card to draw a rectangle or draw a polygon we tell the graphics card that we want to draw a set of vertices and to interpret them in a certain way.

Vertices (cont.)

Example:

```
glBegin(GL_QUADS);
glVertex2f(0.0, 0.0);
glVertex2f(0.0, 1.0);
glVertex2f(1.0, 1.0);
glVertex2f(1.0, 0.0);
glEnd();
```

This draws a simple rectangle although in reality the graphics hardware is drawing 2 triangles.

Matrices

Matrix processing is well suited to graphics hardware because of its parallel nature.

OpenGL uses transformation matrices (specifically Model-View and Projection) to allow for easy manipulation of vertices as operations to translate and rotate these vertices are just simple matrix multiplication or addition that can be done very quickly on graphics hardware.

Lucky for you and many others, you don't need a lot of knowledge about Linear Algebra to deal with these matrices as OpenGL has support for functions to do the math for you and set values appropriately.

Model-View Matrix

The model view matrix is used when drawing things to the screen and you want to position things in either 2d or 3d space. At this stage all vertices that will be drawn have not been flattened yet.

The Model-View matrix applies transformations such as translate and rotate so it is easy to move and rotate objects that you draw on the screen.

Projection Matrix

The Projection Matrix is employed to convert what is essentially 3d vertices to flattened 2d representations that we can view on our monitors.

This matrix is responsible for setting things such as the coordinate plane in which you want to draw, the field of view of the screen, and the camera if you wish to support moving field of view (like in 3d video games).

Matrix Example

Changing the default coordinate system of OpenGL from (-1, 1) based coordinates to something that makes a little more sense:

```
/* Set projection matrix to change coord system */
glMatrixMode(GL_PROJECTION);
/* Load 1 filled identity matrix */
glLoadIdentity();
/* Make bottom left (0, 0) and top right (640, 640) */
glOrtho(0.0, 640, 0.0, 640, 0, 1);
/* Switch to the modelview matrix where we actually draw */
glMatrixMode(GL_MODELVIEW);
```

Applying `glOrtho(...)` to the projection matrix applies what's called an Orthographic Projection so that 3d objects can be converted to 2d representations. This allows us to create a coordinate system which can then be translated to screen coordinates.

Drawing Caveats

One issue some people run into when working with coordinate systems in OpenGL is that in mathematics (0, 0) or the origin is at the bottom left of the "screen". This is not true for drawing systems where the origin as actually at the top left and the Y-axis extends downwards.

If we try to apply a rotation to an object we want to draw and we multiply the model-view matrix by a rotation matrix with a certain angle with origin at the bottom left, we actually rotate the object in the wrong direction. We can fix that by modifying our coordinate system to fall in line with the screen's way of drawing by doing:

```
glOrtho(0.0, 640.0, 640.0, 0.0, 0.0, 1.0);
```

or if you're lazy just using rotating by the negative of the angle you're trying to rotate by.

What is a Shader?

A shader is a transformation that is applied to each and every vertex drawn while the shader is active. Examples of this can be seen in many modern video-games and are really useful because changing effects is as simple as swapping out the shader that you're using with another without having to modify vertex drawing code.

Vertex Shader

Vertex Shaders are used to modify location of vertices or you can use a technique called geometry instancing to generate more vertices from a given vertex. One example of this is implementation of beer goggles in video games such as The Witcher where you see multiple renderings of the same view translated left and right so it seems disorienting or stretching of distances between vertices to cause distortion.

Fragment Shader

Fragment shaders are used for coloring of pixels. This can be used to achieve effects such as gradients and motion blur (more like abused for motion blur) or cartoonish shading effects such as cell shading. Blatant examples of this can be found in Borderlands where everything is cell shaded (which is why if you see textures popping in sometimes they look flat and then they get filled in) and motion blur in just about any racing game.

GLSL

OpenGL provides a language for writing shaders that is C-like in it's syntax and provides many nice features that allow you to use the full power of the graphics hardware and optimizations.

One of the nicer features of this is that shaders are compiled at run-time to provide the best performance on the hardware it's going to run on and this allows you to replace shaders easily without recompiling source code.

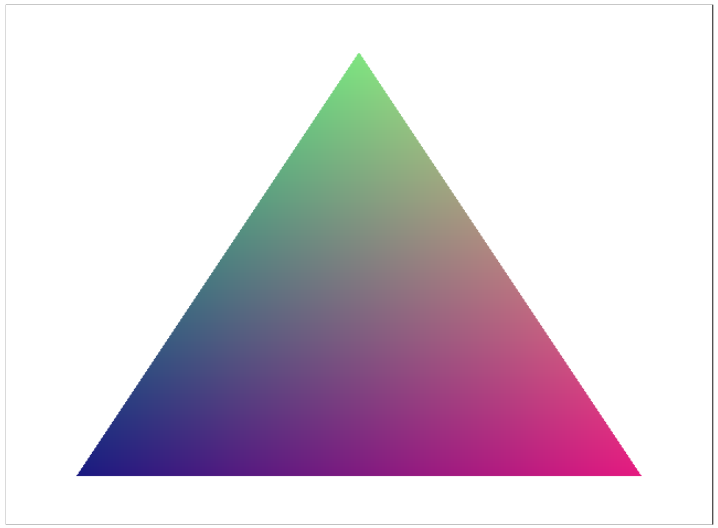
Gradient Shader Example

This is an implementation of a gradient shader for a 640 x 480 sized window. Note: there are ways to pass parameters to shaders but that is a bit more complicated.

```
void main(void)
{
    gl_FragColor[0] = gl_FragCoord.x / 640.0;
    gl_FragColor[1] = gl_FragCoord.y / 480.0;
    gl_FragColor[2] = 0.5;
}
```

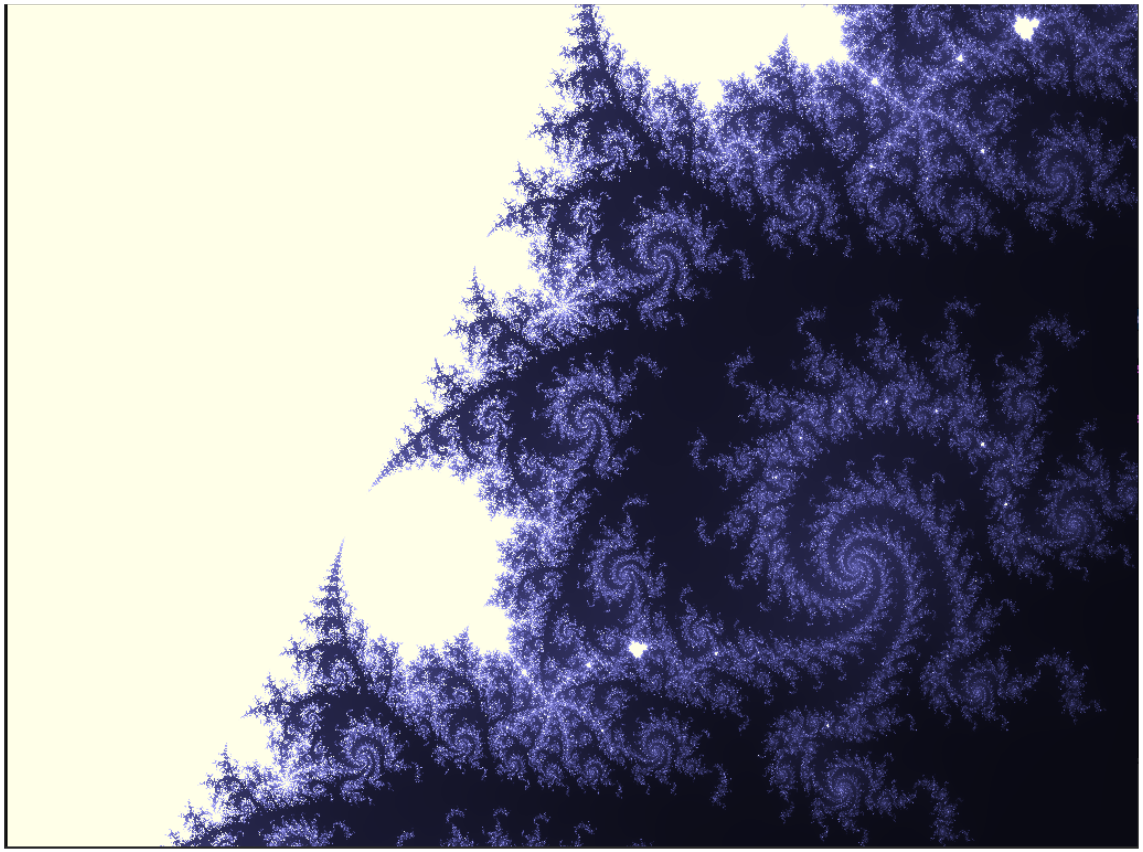
Gradient Shader Example (cont.)

The code draws a simple triangle without specifying color information or anything, it just feeds vertices into the card. The shader actually handles coloring it.



Cooler Shader Example

The mandelbrot set rendered using the shader to offload the heavy math.



More Complicated Examples

DEMO TIME! :)

All demos for this presentation are written in C and OpenGL (obviously) and require FreeGLUT, and some require GLEW. They work on Windows (Cygwin, MingW32), Linux, and Mac provided the dependencies are correctly installed. Depending on your linux distro you may need to remove some of the linker flags such as -lXm and -lXmu as they are not required in some cases.

If you would like to see more complicated examples or some cool stuff written in OpenGL you can check out my GitHub account which contains some examples (more to come probably) as well as some projects that I did with it as small demos at http://github.com/jesus-ramos.

Questions or Comments?

For any questions or comments you can contact me at jramo028@fiu.edu or for a copy of this presentation you can visit http://plug.cs.fiu.edu which contains a link to past presentations as well as my GitHub page where the demos and examples for this presentation are located.

Computer graphics is a very big topic and there's a lot in it that I cannot cover in such a simple presentation but if there's more interest I can make this a series of presentations. This was a broad overview of the key points to OpenGL and computer graphics in general so there might be some things I may have missed.

I'm always willing to help and feel free to borrow any code from this presentation or my GitHub page and mess around with it or use it in a project if you want to.