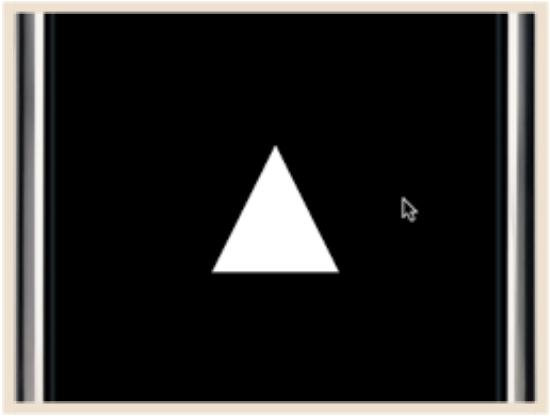


Saturday, 28 March 2009

OpenGL ES 01 - Drawing Primitives 1 - Triangles

Primitives are the basic drawing elements which make up complex objects. In OpenGL ES the primitives you can use are Points, Lines, & Triangles. These are pretty self explanatory and I doubt you need an explanation of what these look like.

First of all, let's look at some code and then we can talk about what's going on so you can then use it to create some of your own code.

Primitive #1 - Triangles
Triangles are the most "complex" of the primitives but they're so easy to use and so useful, this will be the first OpenGL primitive that you'll draw. When drawing a triangle, all we need to do is to feed OpenGL with the three co-ordinates in 3D space for the triangle and it will render it quite happily.

To get started, make a copy of the project from the OpenGL ES 00 tutorial or just download it from here: AppleCoder-OpenGLES-00.tar.gz
Open it up in Xcode and go straight to the `EAGLView.m` file and find the `drawView` method. This is where the magic starts!

First, we need to define the triangle. In order to do this, you need to understand there are two types of co-ordinates that we are going to deal with: Model and World. The Model co-ordinates are in reference to the actual primitive that we are drawing, the world co-ordinates tell

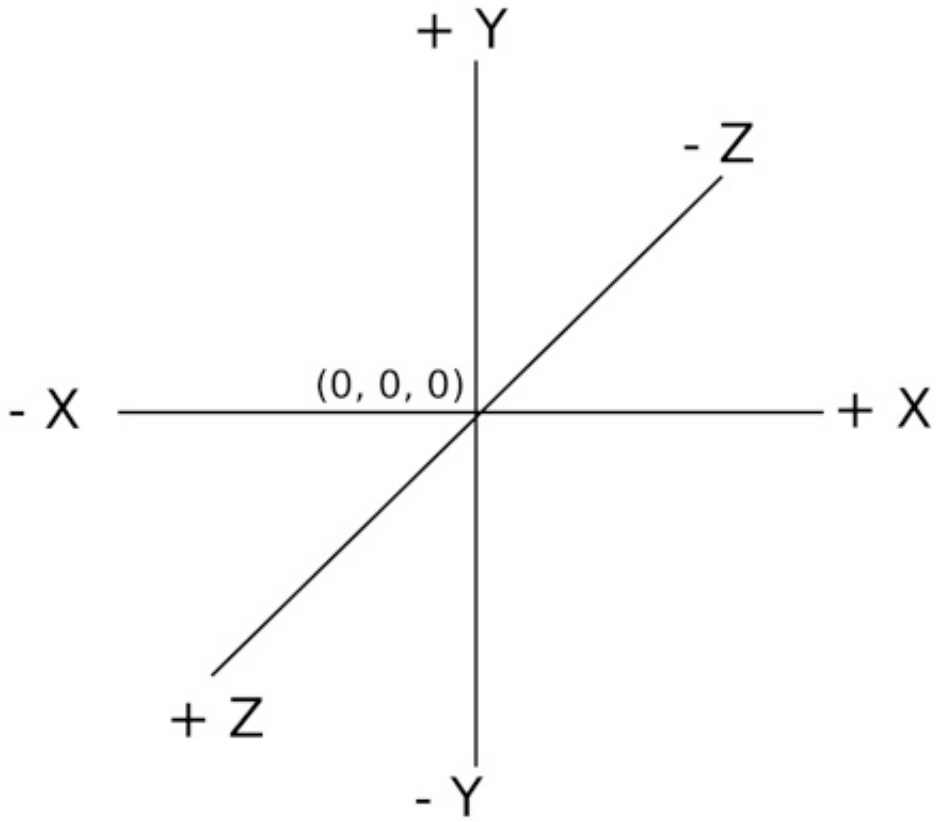
OpenGL where it is in relation to viewer (the viewer is always at **world** co-ordinates of (0.0, 0.0, 0.0).

This first example will illustrate this. First, we define the triangle in the **model** space using 3 x 3D co-ordinates (X, Y, Z):

```
const GLfloat triangleVertices[] = {
        0.0, 1.0, -6.0,// Triangle top centre
        -1.0, -1.0, -6.0,// bottom left
        1.0, -1.0, -6.0,// bottom right
    };
```

As to be expected, there are 3 co-ordinates giving the triangle and note that they are described sequentially in an anti-clockwise direction. Whilst they can be described in a clockwise or anti-clockwise direction, they must be described sequentially and be consistent. However, I would encourage you to start off using anti-clockwise co-ordinates as it's needed for some more advanced functions much later on.

While this tutorial is supposed to be purely iPhone OpenGL ES, for the beginners, I'll describe the 3D co-ordinate system briefly here. Take a look at this picture:



Sorry about my drawing skills but this is a representation of what either the model space or world space looks like. Just imagine this is you're computer screen with X and Y being horizontal and vertical as you would expect, and Z being *depth*. The centre co-ordinates being (0.0, 0.0, 0.0).

So, looking at our triangle as described in the vertices above, the first point (0.0, 1.0, -6.0) would be dead centre on the Y axis, up 1 point and back into the screen 6 points. The second co-ordinate is to the right of

the Y axis 1.0 points, below the X axis (hence -1.0 for the Y value), and still back into the screen -6.0 points. The same applies to the third co-ordinate.

The reason why we have set the object back (ie negative Z value) is so that it will be visible (remember, our viewer or "camera" is at (0.0, 0.0, 0.0) so it would "fail" OpenGL's depth test and not render it at all.

I can hear you screaming "*Hey, I thought you said this was Model co-ordinates which are not world co-ordinates!!*". Yes, that's true, but when we get to rendering this triangle next, OpenGL will simply place the object at (0.0, 0.0, 0.0). So we set it *back* into the screen so it is visible. When we get into transformations (moving, rotating etc), you'll see ways that you don't need to set the object into a negative Z value to make it visible. Until then, leave the Z co-ordinate at -6.0.

The Drawing Code

So all we've done so far is to describe the triangle. We now need to tell OpenGL where that data is stored and how to draw it. This is accomplished with only a few lines of code. Go back to the `drawView` method and implement it as follows:

```
- (void)drawView {

const GLfloat triangleVertices[] = {
        0.0, 1.0, -6.0,                     // Triangle top centre
        -1.0, -1.0, -6.0,                   // bottom left
        1.0, -1.0, -6.0                     // bottom right
    };

    [EAGLContext setCurrentContext:context];
    glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
    glViewport(0, 0, backingWidth, backingHeight);

// -- BEGIN NEW CODE

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLES, 0, 3);

// -- END NEW CODE

    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];

[self checkGLError:NO];
}
```

As you can see, in 4 lines of code we can render a triangle. Let me break this down line by line and you'll see it's actually quite simple.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

This line simply clears the screen. The control bits we feed it tells OpenGL to use the colour we set up in the `setupView` method from the last tutorial (black colour) and to clear the depth buffer. Note that if

we did not clear the depth buffer **and** have depth buffer turned on (as we do), the scene would not render. If we did not enable the depth buffer, we would not need to pass `glClear()` the `GL_DEPTH_BUFFER_BIT`.

So, we've cleared whatever was previously drawn on this buffer (remember, it's double buffered animation; draw on one buffer while another buffer is displayed).

```
glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
```

This function tells OpenGL where our data is and what format it is in. There are 4 parameters for this function and is quite simply broken down as:

1. Size - this is the number of values in each co-ordinate. In our case it is 3 for the values being (X, Y, Z). If you were doing 2D drawing and didn't have a depth (ie Z value), then you could pass 2 here.
2. Data Type - `GL_FLOAT` means that we are passing floating point values. You could also use integer values if you want but you need to get used to floating point values as 3D worlds are floating point.
3. Stride - the stride tell OpenGL to ignore a certain number of bytes between each co-ordinate. Don't worry about this, keep it as zero. You only use this when you are loading vertex data from file in a format which has additional padding data or colour data from, say, a 3D program like Blender.
4. Pointer to the Data - exactly as it appears, the data itself.

So, we've told OpenGL to clear the buffer, told it where the data is for our object and it's format, now we need to tell OpenGL something quite important:

```
    glEnableClientState(GL_VERTEX_ARRAY);
```

OpenGL is a "State" machine. This means that you turn on and off functionality by calling enable and disable commands. Previously, we had used `glEnable()` which affects the "server" side of OpenGL. `glEnableClientState()` affects our program side (ie the client side). So, what we've done is told OpenGL that our vertex data is in a vertex array and turned on OpenGL's functionality for drawing a vertex. A vertex could be a colour array in which case we would call `glEnableClientState(GL_COLOR_ARRAY)` or perhaps a texture co-ordinate array in the case of texture mapping (stop salivating! You need to get through the basics before I cover texture mapping!!).

As we get further into OpenGL, we'll use the different client states and this will become clearer with use.

Now comes the the command to make OpenGL render a triangle:

```
    glDrawArrays(GL_TRIANGLES, 0, 3);
```

Once this function is called, OpenGL takes the information which we had fed it from the previous two functions and executes it. On the screen will be a triangle solid white in colour (white is the default drawing colour). Triangles are filled objects, if you needed an unfilled triangle, then you need to draw it differently.

Breaking down this function, the three arguments are:

1. Drawing Method - in this case, we have passed `GL_TRIANGLES` which seems fairly obvious as we are drawing a triangle. However, the power of this first argument will become apparent when we use this function to draw a square.
2. First Vertex - our array consists of only three points so we want OpenGL to draw from the first co-ordinate in our array which is specified as Zero just as in accessing a standard array. If we had multiple primitives in our vertex array, we could put the offset in here. I'll cover the use of this in a later tutorial when I show you how to build complex objects. For now, just use 0 here.
3. Vertex Count - This tells OpenGL how many vertices are in our array that it needs to draw. Again, we are drawing a triangle so only three points are required. A square would have 4 points, a line 2 points (or more) and a Point would be 1 or more (in the case of rendering multiple Points).

Once you've entered the code and you're `drawView` method looks like mine above, hit "Build and Go" running it in the simulator. You're simulator should look like this:

As promised, there's a solid white triangle in the centre of the screen.

Before we go onto other primitives, try changing the Z value and you will see what I mean if you change it to 0.0. Nothing will render.

There's been a lot of typing over a couple of lines of code, but I hope it's been worth it for you to see just how OpenGL ES works. If you've ever tried to follow a "standard" OpenGL tutorial and just got stuck, I hope you can start to see the difference between OpenGL and OpenGL ES.

Looking Forward
The next tutorial will focus on expanding the code above and producing a square. Below is the link to the project code for you to download if you couldn't get yours working.

If you have any questions, just email me (link below). And here's the source code for this tutorial:

AppleCoder-OpenGLES-01.zip

The home of the tutorials is in the "Tutorials" section of the iphonedevsdk.com forums. Check out the thread there.

Until next time, hooroo!
Simon Maurice