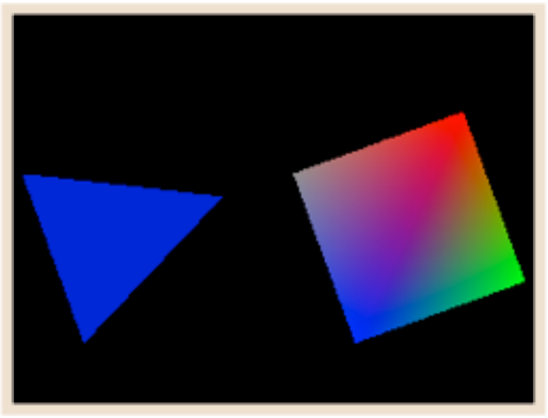
SIMON MAURICE - IPHONE - OPENGL ES



Tuesday, 31 March 2009

OpenGL ES 04 - Colour and Shading

Like I said in the last tutorial, I'm getting a bit bored with plain white objects on the screen, let's add some colour just like the original Apple template had before we removed it. Do pay attention to this one because some of the concepts that I will introduce will also come into play when we start texture mapping (which will be Real Soon Now).

In OpenGL ES, colour can be set as a single block colour for the entire object, or can be multi-coloured and have shading so the colours drift through the spectrum from one colour to the next. Single colour is not very complicated so let's colour up our objects in a single colour.

Like all things with OpenGL, changing the colour puts OpenGL in a "state" where all following drawing operations will be in that colour, even if we call our "reset" being `glLoadIdentity()` (this is because `glLoadIdentity()` operates on the actual vertices only). So by adding a single line of code, we can make our two objects appear in any colour; anything's better than white, but I'm just going to go with blue for now.

Fire up Xcode and head off to `drawView`. After the first `glLoadIdentity()` call, add the following GL function call:

```
    glLoadIdentity();
    glColor4f(0.0, 0.0, 0.8, 1.0);
```

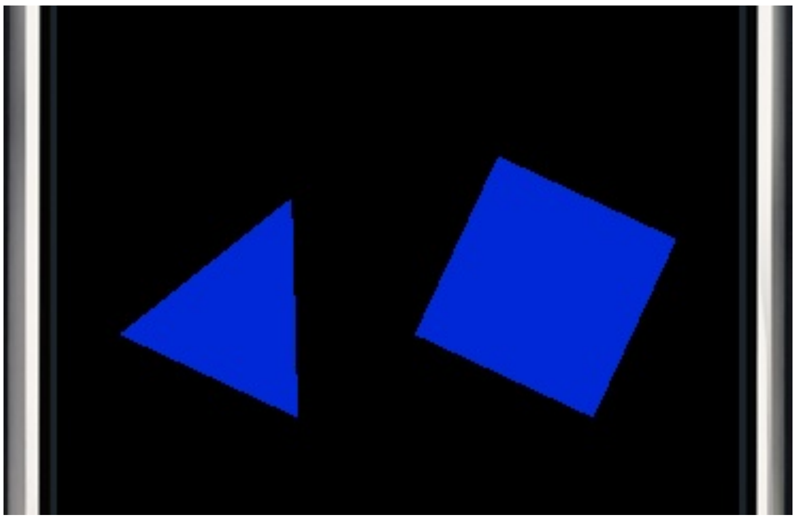
This function call, `glColor4f()` tell OpenGL to commence drawing (and filling) in this colour which is a blue colour. The parameters are:

```
    glColor4f(  GLfloat red,
                 GLfloat green,
                 GLfloat blue,
                 GLfloat alpha);
```

In OpenGL ES, you must provide colours with these four parameters (ie RGBA), there is no option for RGB colours. In case you don't know, alpha is the level of transparency with 1.0 being fully solid, down to 0.0 which is fully transparent.

The red, green, & blue parameters are floating point values being between 0.0 and 1.0 with 0.0 being nil intensity, and 1.0 being full intensity. Therefore white would be (1.0, 1.0, 1.0, 1.0).

Add the line and hit "Build and Go". Our two objects will look like this:



Better than plain white but not very inspiring. The psychedelic colours of the Apple rotating square was more interesting so let's have a look at how to create that.

Multiple Colours
Having an object with multiple colours is not too much more work. We need to define and array just like the vertex arrays we have been using and then tell OpenGL to get it's colours from that array. Each colour in the colour array, represents a colour at each vertex (point) within our object's vertex array.

Let me make that a bit clearer as we colour up the square. Have a look at the following code where I have defined a colour array to go with the square's vertex array:

```
const GLfloat squareVertices[] = {
        -1.0, 1.0, 0.0,               // Top left
        -1.0, -1.0, 0.0,              // Bottom left
        1.0, -1.0, 0.0,               // Bottom right
        1.0, 1.0, 0.0                 // Top right
    };

    const GLfloat squareColours[] = {
        1.0, 0.0, 0.0, 1.0,// Red - top left - colour for
squareVertices[0]
        0.0, 1.0, 0.0, 1.0,   // Green - bottom left -
squareVertices[1]
        0.0, 0.0, 1.0, 1.0,   // Blue - bottom right -
squareVerticies[2]
        0.5, 0.5, 0.5, 1.0    // Grey - top right-
squareVerticies[3]
    };
```

I hope this illustrates that each colour we have given, represents a vertex for the square. Before we can run this though, we need to add some more code for colouring the square:

```
    glLoadIdentity();
    glTranslatef(1.5, 0.0, -6.0);
    glRotatef(rota, 0.0, 0.0, -1.0);
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColorPointer(4, GL_FLOAT, 0, squareColours);      // NEW
    glEnableClientState(GL_COLOR_ARRAY);                // NEW
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glDisableClientState(GL_COLOR_ARRAY);               // NEW
```

There are three new lines of code. Let's deal with them one-by-one:

```
   glColorPointer(4, GL_FLOAT, 0, squareColours);
```

This is in fact similar to the function which we call to set up our co-ordinate vertex array. The four parameters are:

- Size - the number of colours in the array
- Data Format - we've used `GL_FLOAT` here because our vertex array contains floating point numbers. You can also use bytes if you want to specify the colours in 0-255 format.
- Stride - Again, this tells OpenGL to skip a number of bytes between each value if your data contains other information.
- Array Points - where the data is stored.

Note that when specifying the data format, `GL_FLOAT` is the parameter format (an enumeration) telling OpenGL what format; `GLfloat` is the data type for declaring a floating point number for OpenGL.

OK, so that function call tells OpenGL where the data is and what format it is in. However, like the co-ordinate vertex array which tells OpenGL the co-ordinates of the object, we need to put OpenGL into the required "state" which will make OpenGL use our colours when rendering the object.

So this call:

```
glEnableClientState(GL_COLOR_ARRAY);
```

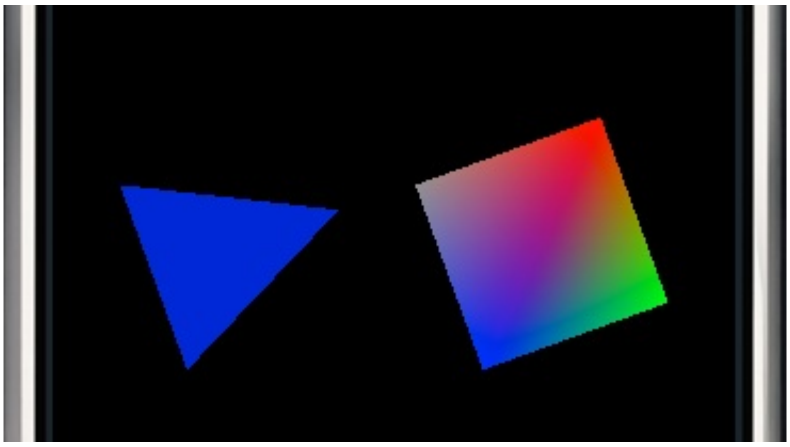
This enables the appropriate state in the OpenGL engine. Instead of passing `GL_VERTEX_ARRAY`, we just tell OpenGL that it's a colour array with `GL_COLOR_ARRAY`.

Next we simply draw the square as per normal. After the square has been drawn, we need to then disable the colour array. If we don't, the next time we draw the triangle, it too will be coloured like the square. So we call:

```
glDisableClientState(GL_COLOR_ARRAY);
```

This takes the colour array off the list of OpenGL's current state. If we didn't do this, the first call to `drawView` will have the triangle in blue, the second call to `drawView` will use the colour array to colour the triangle. However, there being only three vertices in the triangle's co-ordinate array (`triangleVerticies[]`), it will only use the first three colours.

So, make the changes to `drawView`, adding those three new lines as shown above, then hit "Build and Go" and you're display should like this:



If you like, you can turn off rotation (comment the `glRotatef()` function call) so you can see the relation of the square's vertex array vertices with the colour array.

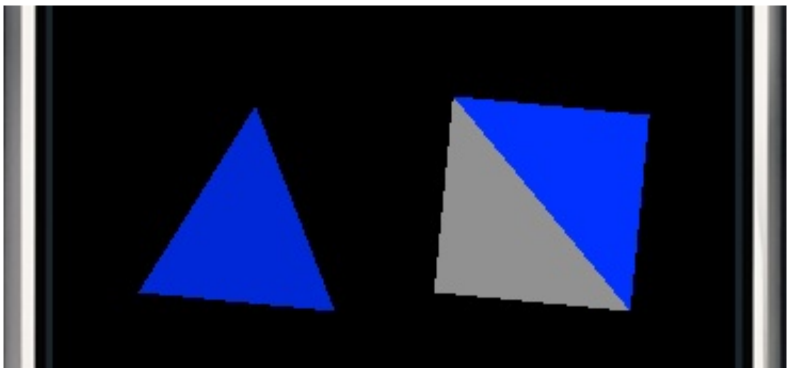
Shading
Notice how the square gradually goes from one colour to the next? OpenGL achieves this through shading. There are two shading models

which can be used by OpenGL: `GL_FLAT` & `GL_SMOOTH`. What you're seeing already is the `GL_SMOOTH` shading which is the default.

Just to show the difference, before the `glLoadIdentity()` for the square, insert the following line:

```
    glShadeModel(GL_FLAT);
```

The `glShadeModel()` function changes the OpenGL state to the flat shading model from the smooth shading model. Again, OpenGL changes it's state and retains this state until you tell it otherwise so you could put that in the `setupView` method if you like. After doing a "Build and Go", the shading of the square will change to the following:



Let me explain what's happening here.

The triangle is being rendered as per normal. Being a flat colour, shading does not affect the drawing of the triangle. With the square, you can clearly see now the two triangles which OpenGL uses to make up the square. Due to the flat shading model, OpenGL only uses the *last* colour for filling the each triangle, being `squareColours[2]` (blue) and `squareColours[3]` (grey). Please review the Square Primitives tutorial if you're unsure why these two colours represent the last two points in rendering the two triangles which make up the square.

As a recap: `GL_SMOOTH` is smooth shading, which means that when it comes to filing the square, OpenGL takes the defined colour in our `squareColours[]` array for each vertex in our `squareVertices[]` array, and uses interpolation for each pixel in the square between the points to smoothly change the colour between each of the four points. In other words, it gives us that coloured square shown originally.

`GL_FLAT` uses the colour defined for the last vertex of the object and fills the entire primitive with that colour. Squares are made up of two triangles so we have the square coloured in two halves.

Conculsion

Well I hope this has been useful for you. In reality, you probably just want to leave the shading as `GL_SMOOTH` unless you're doing one of those retro-3D games from the C64 days. `GL_SMOOTH` is the default so you don't need to enable it.

Also, please note, that what you used above for the colour points is also used for texture mapping so I'll be coming back to this in a tutorial or two.

Texture mapping is just around the corner now. I'm going to show you how to create a 3D object in the next tutorial. It will be flat coloured but that's okay because we'll texture map it in the following tutorial.

Here's the finished code for today's tutorial.

AppleCoder-OpenGLES-04.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

previous next