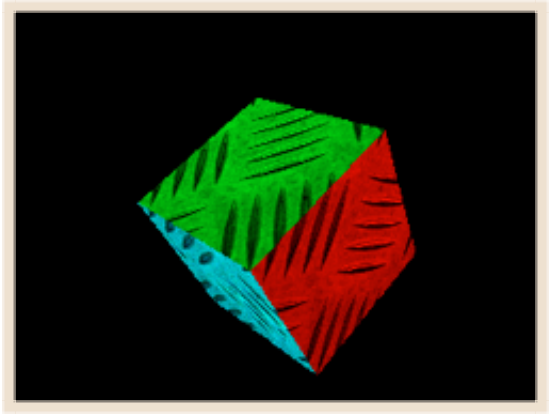
SIMON MAURICE - IPHONE - OPENGL ES



Wednesday, 1 April 2009

OpenGL ES 06 - Objects in 3D

So far, we have done a fair bit with our 2D objects. Now is the time to head off and create 3D objects. Whilst they are not significantly more complicated, they do require more vertices (if you create them using a vertex array) or more transformations if you wanted to use our square and duplicate it into a cube.

I suppose I should have covered points and lines first, but hey, we're on a roll so far with our little texture mapped square and coloured triangles so let's not slow down to less interesting shapes!

Also I do need to return to transformations and cover rotation in some more detail. And then there's the really basic beginner stuff which I haven't covered.... Well, all it means is that I've got a lot of tutorials to write yet!!

To Begin With, Gut the `drawView` Method
Say good-bye to all your hard coding work, it's time to cut everything out and return the `drawView` method to it's most basic state.

Make the drawView method look like this:

```
- (void)drawView {

// Our new object definition code goes here
```

```
    [EAGLContext setCurrentContext:context];
    glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
    glViewport(0, 0, backingWidth, backingHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);

// Our new drawing code goes here

    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];
    [self checkGLError:NO];
}
```

You can thank me right now for making everything work in a 3D space up front because otherwise we'd be working on the depth buffer etc and adding lots of new code. By now the above should be quite familiar to you.

Defining a 3D Object

We're going to produce a cube because they're easy to conceptualise, quite common in 3D graphics, and look cool rotating in 3D with texture mapping. Before we can draw a cube in 3D, we just need to remember that a 3D cube is just 6 of our squares that we've been using all along. It's easy but I'll break the definition down for you all. Let's start by defining the front face:

```
    const GLfloat cubeVertices[] = {

        // Define the front face
        -1.0, 1.0, 1.0,             // top left
        -1.0, -1.0, 1.0,            // bottom left
        1.0, -1.0, 1.0,             // bottom right
        1.0, 1.0, 1.0,              // top right
```

Almost identical to the original square except that I've brought the face forward one unit. I'll mention why in the closing of this tutorial. Now the top face:

```
        // Top face
        -1.0, 1.0, -1.0,            // top left (at rear)
        -1.0, 1.0, 1.0,             // bottom left (at front)
        1.0, 1.0, 1.0,              // bottom right (at front)
        1.0, 1.0, -1.0,             // top right (at rear)
```

Note that I've drawing the top face not only in the same direction as the front face (vertices specified in a anti-clockwise direction when viewed from the rear face's front, not the front of the object) but I've also started off at the same point? If we were to rotate the finished cube 90° along the X axis so now the top face is the front face, the first vertex specified is the top left, followed by the top right?

Next, the rear face:

```
        // Rear face
        1.0, 1.0, -1.0,             // top right (when viewed
```

```
from front)
        1.0, -1.0, -1.0,            // bottom right
        -1.0, -1.0, -1.0,           // bottom left
        -1.0, 1.0, -1.0,            // top left
```

Notice the continuation of the order and starting point of the vertices? We continue to do this throughout the remaining faces.

```
       // Bottom Face
        -1.0, -1.0, 1.0,            // Bottom left front
        1.0, -1.0, 1.0,             // right front
        1.0, -1.0, -1.0,            // right rear
        -1.0, -1.0, -1.0,           // left rear
```

Can you still see that I'm following the same order and starting point. Imagine rotating the face to the front in your mind and seeing where the points line up.

Finally, we have the left and the right face:

```
         // Left face
        -1.0, 1.0, -1.0,            // top left
        -1.0, 1.0, 1.0,             // top right
        -1.0, -1.0, 1.0,            // bottom right
        -1.0, -1.0, -1.0,           // bottom left

        // Right face
        1.0, 1.0, 1.0,              // top left
        1.0, 1.0, -1.0,             // top right
        1.0, -1.0, -1.0,            // right
        1.0, -1.0, 1.0              // left
```

Here's the full definition of the cube:

```
    const GLfloat cubeVertices[] = {

        // Define the front face
        -1.0, 1.0, 1.0,             // top left
        -1.0, -1.0, 1.0,            // bottom left
        1.0, -1.0, 1.0,             // bottom right
        1.0, 1.0, 1.0,              // top right

        // Top face
        -1.0, 1.0, -1.0,            // top left (at rear)
        -1.0, 1.0, 1.0,             // bottom left (at front)
        1.0, 1.0, 1.0,              // bottom right (at front)
        1.0, 1.0, -1.0,             // top right (at rear)

        // Rear face
        1.0, 1.0, -1.0,             // top right (when viewed
from front)
        1.0, -1.0, -1.0,            // bottom right
        -1.0, -1.0, -1.0,           // bottom left
        -1.0, 1.0, -1.0,            // top left

        // bottom face
        -1.0, -1.0, 1.0,
        -1.0, -1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0,
```

```
        // left face
        -1.0, 1.0, -1.0,
        -1.0, 1.0, 1.0,
        -1.0, -1.0, 1.0,
        -1.0, -1.0, -1.0,

        // right face
        1.0, 1.0, 1.0,
        1.0, 1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0
    };
```

If you're having problems with the co-ordinate system, then please really make an effort to visualise it in your mind. If you have problems with that, grab out a piece of paper and draw it in an isometric view. You really now need to start to get a grasp on objects in 3D.

Put the `cubeVertices` below the comment I made in the `drawView` method stating "New object definition goes here".

OK, now we need to draw this sucker.

Drawing the Cube

The easiest way for me to teach you to draw this cube is to use only the code which you have seen previously. Later on, we're going to go into some more advanced (but ultimately easier once you understand it) ways to draw 3D objects. However, for now, let me just introduce drawing in 3D first.

Let's start with some code which will need no explanation. Below where I've put "Our new drawing code goes here, add the following lines:

```
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -6.0);
    glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
```

Nothing new here. We just reset our vertex state, move the cube back into the screen away from us so we can see it, tell OpenGL about our vertex array and the format it's in, then enable OpenGL to use it.

The follow code is almost the same as you've used before:

```
// Draw the front face in Red
glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

That's nothing really new, we've set the drawing colour to red and told OpenGL to draw the vertices 0 to 4 in our array as a square. Now let's

draw the top face which is the next 4 vertices in our array:

```
// Draw the top face in green
glColor4f(0.0, 1.0, 0.0, 1.0);
glDrawArrays(GL_TRIANGLE_FAN, 4, 4);
```

Look at `glDrawArrays()`. If you remember me describing it to you, I said that the second parameter was the start offset? Well, because we're drawing the second square or face in our cube, we need to tell OpenGL to start at the offset 4 (which is `cubeVertices[4]`, the offsets 0~3 are the front face), and then draw another four vertices.

Now we can draw the rear face:

```
    // Draw the rear face in Blue
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 8, 4);
```

Same deal, this time we're starting at `cubeVertices[8]`. The same pattern continues for the final 3 faces:

```
    // Draw the bottom face
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 12, 4);

    // Draw the left face
    glColor4f(0.0, 1.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 16, 4);

    // Draw the right face
    glColor4f(1.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
```

All we've done once again is to change the colour, and change the starting offset for `glDrawArrays()`.

Now, if you were to hit "Build and Go" right now, all you'd get in an inanimate red square which would be nothing further along from where you started. In order to see all 6 sides in all their glory, let's rotate this cube along all three axes.

Right before `glLoadIdentity()`, add the following assignment:

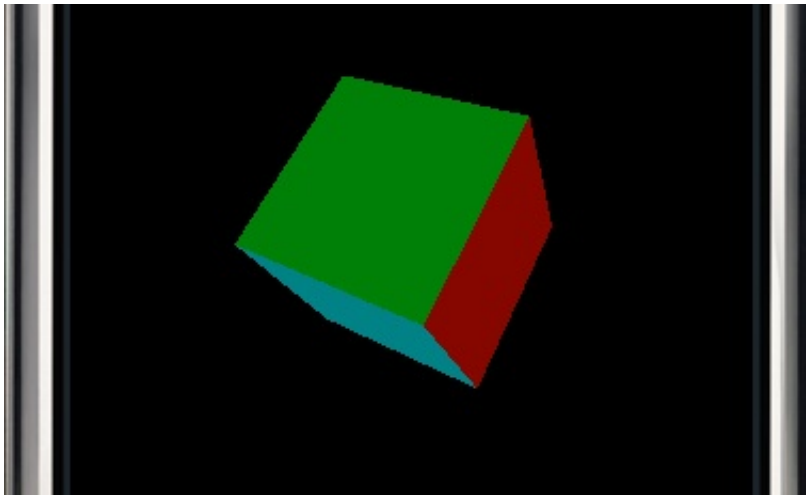
```
    rota += 0.5;
```

Our old friend Mr. `rota` has returned. Now we need our other old friend, `glRotatef()`. After the `glTranslatef()`, add the following line:

```
    glRotatef(rota, 1.0, 1.0, 1.0);
```

Previously, we'd only used `glRotatef()` along one axis. Now we're using it to rotate all three axes at the same time.

Now you can hit "Build and Go" and this is what you get:



Welcome to 3D objects.

What About Texture Mapping It?
I think we're a bit past plain coloured objects now aren't we? Let's use the texture from the last tutorial and texture map all six sides and make it more interesting.

OK, we kept our texture in the project and we kept the loading code so all we need to do is change the `drawView` method. Now you get to see just how quick texture mapping is once it's already there!

First of all, remember this from the last tutorial:

```
const GLshort squareTextureCoords[] = {
    // Front face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right
```

Well, that was good when all we had to texture map was one face. We need to expand it. However, it's easy. Remember above how I was pedantic on keeping the starting co-ordinates and the order of the co-ordinates the same when I was defining the cube? Well now you find out why.

When OpenGL renders a texture on a face of our cube, because we are starting everything after the first face at an offset (the 4, 8, 12 etc from above to `glDrawArrays()`), the texture mapping side of the rendering will also go to the same offset when looking up the texture co-ordinates for the each face. So therefore, in order to texture map 6 faces, all we need to do is to duplicate the above 4 co-ordinates 5 more times to produce the texture co-ordinate array as follows:

```
const GLshort squareTextureCoords[] = {
    // Front face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Top face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Rear face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Bottom face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Left face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Right face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right
};
```

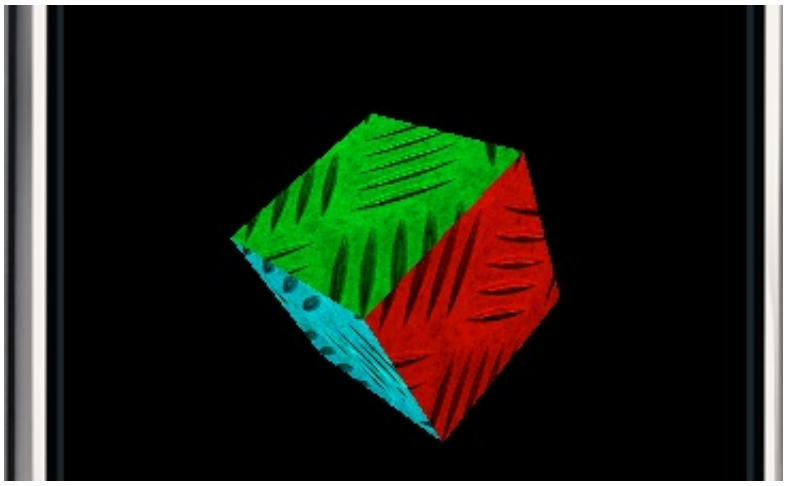
A bit of cut and paste action and job's done!

Now, all we need is a few lines of drawing code and we're ready to render our texture mapped cube:

Before drawing the first face, add the following code:

```
glTexCoordPointer(2, GL_SHORT, 0, squareTextureCoords);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

It's that easy. Don't delete the `glColour4f()`'s before each drawing code, just hit "Build and Go" first so you get the following:



Hey presto, a texture mapped 3D object, rotating in full 3D.

A Word on Texture Map Images
I forgot to mention this last tutorial. Whilst I encourage you experiment with your own textures for texture mapping, just be aware that the texture sizes must be a power of two. That is the width and height of the image has to be 1, 2, 4, ... 32, ... 512, 1024... The width and height don't have to be equal but they must be a power of 2. So 32 x 512 is a valid size just the same as 64 x 64 is. But 30 x 30 is not.

A Word on `glRotatef()` and Your Object's Vertices
Again, I encourage you to create your own objects. Have you noticed that I've always had the centre of my triangles, squares and now cube at 0,0,0? The extremes of each object are equally spaced apart from 0,0,0? That's because when it comes to rotation, OpenGL will rotate quite naturally around the object's centre as specified in relation to 0,0,0 in the model matrix. It won't adjust your model to 0,0,0 and then rotate it. If your object is not centred at 0,0,0 in you're object matrix, your object will rotate "lop-sided".

That's it for Today
That's the way today's tutorial is ending. I hope you've found in beneficial and it's been as much fun for you as I've had creating it. As always, click on the Email Me link at the bottom if you have any questions.

Actually, in hindsight, I should have switched to anti-clockwise ordering of the vertices now we're in 3D objects. Never mind, I'll go into detail on using anti-clockwise versus clockwise in the next tutorial and we'll create a cube going anti-clockwise.

Here's the code for this tutorial:

AppleCoder-OpenGLES-06.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