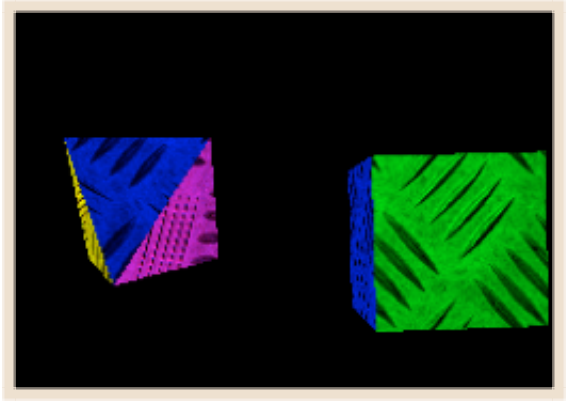
SIMON MAURICE - IPHONE - OPENGL ES



Friday, 3 April 2009

OpenGL ES 07 - Translating Objects Independently

Originally, I thought I would head into lighting next but then I had another thought. We still haven't completed much of the basics with objects and transformations, particularly if we want to transform (animate) two different objects differently within our worldspace.

Remember how we've just applied `glTranslatef()` and `glRotatef()` to the whole scene? If we wanted to do something differently, I used a convenience function, `glLoadIdentity()`. However, `glLoadIdentity` is "expensive" in terms of computational power and scene rendering so now, I'll introduce a more efficient method.

To do this, I am going to introduce a new object as well. We are going to add a pyramid, then move it with `glTranslatef()`, and rotate it with `glRotatef()` independently to the cube **without** making another call to `glLoadIdentity()` to reset our verticies.

Adding a Pyramid to Our World
Rather than just add a triangle, let's stick with 3D objects and we'll add a pyramid (just think of the big stone things on the plains of Giza).

One thing I avoided saying in the last tutorial was was that the cube we created was a complex object; i.e. an object made up of more than one primitive. Technically speaking, our square was also a complex object but since we drew it with a single OpenGL function call, we can treat it as a simple object. I avoided calling it a complex object so as not to make it sound like "hard work".

Now that you've done one complex object, congratulations! However, it's time to create your second!

Pyramids are not hard to do. They are quite simply made up of a square for the base attached to four triangles which meet at the centre of the square above it. Once you break down in your mind the shape you wish to create into more simple objects, you'll have no problem creating whatever you want. The only thing which changes is the quantity of primitives which go into creating your object.

So, fire up Xcode and open the project from the last tutorial. Nothing to delete this time, we're adding code and changing a couple of lines of code.

In order to help you with the pyramid, I'm going to break it down into it's individual components, let's start with the base:

```
    const GLfloat pyramidVertices[] = {
        // Our pyramid consists of 4 triangles and a square base.
        // We'll start with the square base
        -1.0, -1.0, 1.0,            // front left of base
        1.0, -1.0, 1.0,             // front right of base
        1.0, -1.0, -1.0,            // rear left of base
        -1.0, -1.0, -1.0,           // rear right of base
```

So, we've just started creating a new object as we have done in the past. The square is the base so all vertices are set at the same Y co-ordinate of -1.0.

Now we can create the front face of the pyramid, this time we are creating a triangle instead of a square:

```
        // Front face
        -1.0, -1.0, 1.0,            // bottom left of triangle
        1.0, -1.0, 1.0,             // bottom right
        0.0, 1.0, 0.0,              // top centre -- all triangle
vertices
                                    //     will meet here
```

The only real difference to the triangles we have created in the past is that this one is set on an angle, tilting backwards toward the centre of the square (i.e. rotated around the X axis).

Then we can continue specifying our other three triangular faces just in

the same manner. Here's the resulting full declaration of the pyramid:

```
// Our new object definition code goes here
    const GLfloat pyramidVertices[] = {
        // Our pyramid consists of 4 triangles and a square base.
        // We'll start with the square base
        -1.0, -1.0, 1.0,            // front left of base
        1.0, -1.0, 1.0,             // front right of base
        1.0, -1.0, -1.0,            // rear left of base
        -1.0, -1.0, -1.0,           // rear right of base

        // Front face
        -1.0, -1.0, 1.0,            // bottom left of triangle
        1.0, -1.0, 1.0,             // bottom right
        0.0, 1.0, 0.0,              // top centre -- all triangle
vertices
                                    //     will meet here

        // Rear face
        1.0, -1.0, -1.0,   // bottom right (when viewed through
front face)
        -1.0, -1.0, -1.0,           // bottom left
        0.0, 1.0, 0.0,              // top centre

        // left face
        -1.0, -1.0, -1.0,           // bottom rear
        -1.0, -1.0, 1.0,            // bottom front
        0.0, 1.0, 0.0,              // top centre

        // right face
        1.0, -1.0, 1.0,             // bottom front
        1.0, -1.0, -1.0,            // bottom rear
        0.0, 1.0, 0.0               // top centre
    };
```

Add the above pyramid definition to the `drawView` method at the same spot as the `cubeVerticies[]` definition.

Just before we move on, I do want to make some points about the pyramid definition.

Firstly, this is the first object we are going to render made up of squares and triangles. The base of our pyramid is a square and the four sides are triangles. Since we are calling `glDrawArrays()` independently for each primitive, it doesn't matter that we have different primitives in a single definition. All will become clear when we draw the pyramid below.

Secondly, note that once again, I have specified all vertices in an anti-clockwise direction. Even though the rear face "appears" to be specified in a clockwise direction when viewed "through" the front face, from OpenGL's perspective, it is written from an anti-clockwise direction.

Actually, I think I will introduce a 3D graphics term at the end of this tutorial to discuss this further.

Drawing the Pyramid
Now head down to the drawing code for the cube. Firstly, delete the `glLoadIdentity()` line, that's no longer required.

After the assignment to rota += 0.5, we're going to add the code for drawing the pyramid. Now, what we want to do is to move the pyramid using `glTranslatef()` and then rotate it using `glRotatef()` without affecting the cube. In other words, we need to be able to call `glTranslatef()` and `glRotatef()` without affecting the translation and drawing operations performed on any other object drawn after it.

OpenGL provides us with a convenient way to do this using the following function pair:

```
glPushMatrix();

// Translation and drawing code goes here....

glPopMatrix();
```

What OpenGL is doing for us is allowing us to use a stack which our matrices are copied onto with the call to `glPushMatrix()`. I know I've avoided most 3D terms but just think of our `pyramidVertices[]` and `cubeVertices[]` objects as the matrices we are "pushing" onto OpenGL's stack.

Note: If you're unsure what I mean by a stack, check this out or, better yet, you really need to get yourself a good C or Objective C programming book.

So, with our data safely copied and out of the way, we can transform away like the OpenGL gurus we are! Let's start drawing the the pyramid:

```
glPushMatrix();
{
    glTranslatef(-2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 0.0, 0.0);
    glVertexPointer(3, GL_FLOAT, 0, pyramidVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
```

First thing to note is the curly bracket after the `glPushMatrix()`. This is not required at all but as you're learning this, it just makes it clear where the matrices are pushed and then popped off the stack.

The first four lines of code after the `glPushMatrix()` call should be self explanatory to you by now. All we're doing is calling `glTranslatef()` to move the pyramid away from (0, 0, 0) to the left and back 8 points into the screen (further away from the viewer). Then we are rotating the pyramid around the X axis only, rather than all three

axes as per the cube example in the last tutorial. Finally, we just tell OpenGL about the data and enables it to be used.

Now, after we've done our transformations, we can start to draw the pyramid:

```
        // Draw the pyramid
        // Draw the base -- it's a square remember
        glColor4f(1.0, 0.0, 0.0, 1.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

The first four co-ordinates of our pyramid declaration above contains the square base. So, we will colour it in red (it will still be texture mapped as the code for texture mapping is already there - see the full implementation of `drawView` below), and draw a square using the `GL_TRIANGLE_FAN` methodology. We 3-co-ordinate vertex at vertex 0 in the array (`pyramidVerticies[0~3]`), and use 4 vertices.

Ok, that's the square done, let's now deal with the first triangle:

```
        // Front Face
        glColor4f(0.0, 1.0, 0.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 4, 3);
```

Apart from the change in colour, we are changing the drawing methodology being `GL_TRIANGLES` for obvious reasons, we're starting at array element 4 (`pyramidVerticies[4~6]`), and then proceeding to draw three vertices. So you can see that a square and triangles can easily co-exist in the one data structure.

Next, we can just continue drawing our remaining three triangles:

```
        // Rear Face
        glColor4f(0.0, 0.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 7, 3);

        // Right Face
        glColor4f(1.0, 1.0, 0.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 10, 3);

        // Left Face
        glColor4f(1.0, 0.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 13, 3);
    }
    glPopMatrix();
```

Each time, we're just changing the colour, and changing the starting offset. If you recall, OpenGL knows that each vertex has 3 co-ordinates (X, Y, Z) as we passed that to `glVertexPointer()` above.

Finally, we close the curly brackets and call `glPopMatrix()`.

Now, at this point we can draw the cube. However, note that you can

repeat drawing the same object over and over again just by surrounding your transformations around `glPushMatrix()` and `glPopMatrix()`.

Drawing the Cube - Revised

Here is the full drawing code for the cube. The most obvious change is the surrounding of the code with a `glPushMatrix()` and `glPopMatrix()` function calls:

```
    glPushMatrix();
    {
        glTranslatef(2.0, 0.0, -8.0);
        glRotatef(rota, 1.0, 1.0, 1.0);
        glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
        glEnableClientState(GL_VERTEX_ARRAY);

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

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

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

        // 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);
    }
    glPopMatrix();
```

The only other change is the information passed to `glTranslatef()`, moving our cube off the centre of our display, 2.0 units to the right and back into the screen 8.0 units.

Just for completeness, here is the full `drawView` method code (below is a few notes):

```
- (void)drawView {

    // Our new object definition code goes here
    const GLfloat pyramidVertices[] = {
        // Our pyramid consists of 4 triangles and a square base.
        // We'll start with the square base
        -1.0, -1.0, 1.0,            // front left of base
        1.0, -1.0, 1.0,             // front right of base
        1.0, -1.0, -1.0,            // rear right of base
        -1.0, -1.0, -1.0,           // rear left of base
```

```
        // Front face
        -1.0, -1.0, 1.0,            // bottom left of triangle
        1.0, -1.0, 1.0,             // bottom right
        0.0, 1.0, 0.0,              // top centre -- all triangle
vertices
                                    //     will meet here

        // Rear face
        1.0, -1.0, -1.0,     // bottom right (when viewed through
front face)
        -1.0, -1.0, -1.0,           // bottom left
        0.0, 1.0, 0.0,              // top centre

        // left face
        -1.0, -1.0, -1.0,           // bottom rear
        -1.0, -1.0, 1.0,            // bottom front
        0.0, 1.0, 0.0,              // top centre

        // right face
        1.0, -1.0, 1.0,             // bottom front
        1.0, -1.0, -1.0,            // bottom rear
        0.0, 1.0, 0.0               // top centre
    };

    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
    };
```

```
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
};

[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);

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

// Our new drawing code goes here
rota += 0.5;

glPushMatrix();
{
    glTranslatef(-2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 0.0, 0.0);
    glVertexPointer(3, GL_FLOAT, 0, pyramidVertices);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Draw the pyramid
    // Draw the base -- it's a square remember
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    // Front Face
    glColor4f(0.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLES, 4, 3);
```

```
            // Rear Face
            glColor4f(0.0, 0.0, 1.0, 1.0);
            glDrawArrays(GL_TRIANGLES, 7, 3);

            // Left Face
            glColor4f(1.0, 1.0, 0.0, 1.0);
            glDrawArrays(GL_TRIANGLES, 10, 3);

            // Right Face
            glColor4f(1.0, 0.0, 1.0, 1.0);
            glDrawArrays(GL_TRIANGLES, 13, 3);
        }
        glPopMatrix();

        glPushMatrix();
        {
            glTranslatef(2.0, 0.0, -8.0);
            glRotatef(rota, 1.0, 1.0, 1.0);
            glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
            glEnableClientState(GL_VERTEX_ARRAY);

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

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

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

            // 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);
        }
        glPopMatrix();


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

        [self checkGLError:NO];
    }
```

Note that we have not added a new texture co-ordinate array for the pyramid? We are just going to use the same co-ordinate array because it will just work for this demonstration.

OK, make the changes to your project (only need to change `drawView`) and hit “Build & Go”. You should get the following on screen:

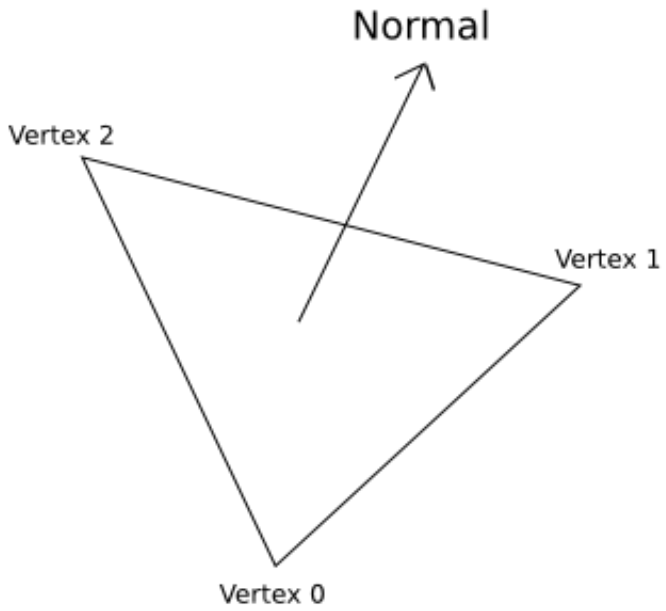
All you need to notice is that the two objects are now being transformed independently of each other. The pyramid is being rotated around the X axis only while the cube continues to rotate around all three axes. Once again, the colours only make it easier for you to identify each face.

Introducing a 3D Concept: Normals

Remember earlier I was making the note about the co-ordinates for each vertex (point) of the rear triangle being specified in an anti-clockwise

direction despite it "appearing" when viewed from the front that they were specified in a clockwise direction?

Well, that's because you should always specify your data in reference to the face's "normal". Simply put, the normal of an object's face is an imaginary line drawn out perpendicular to the object's face. To illustrate, have a look at the following image:



In the above image, the triangle's normal is represented by the arrow coming out of the face. Now that the triangle's normal has been represented, you can now see that the 3 vertices are specified in an anti-clockwise direction. We will use normals extensively when I cover lighting so I do want you to be aware of what a normal is now.

That's It for this Topic
As usual, here's the source code for this tutorial.

AppleCoder-OpenGLES-07.zip

I had been planning to cover lighting next but I might actually create a 3D world and show you how to move through it. Then I think I can use that 3D world to add lighting. Let me sleep on it and we'll see what comes up next.

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

Until then, hooroo!
Simon Maurice

previous next