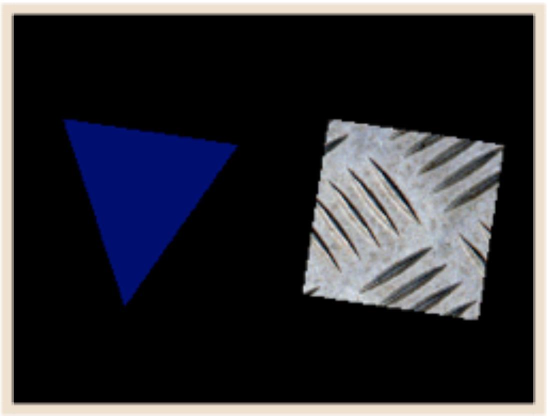
SIMON MAURICE - IPHONE - OPENGL ES



Wednesday, 1 April 2009

OpenGL ES 05 - Texture Mapping Our Square

I've decided to bring forward texture mapping because it's probably easier to texture map a single faced object rather than a multi-faced (or 3D object). Also, it seems this is where most iPhone OpenGL ES programmers seem to get a bit stuck so I'll run through texture mapping now.

I know that I have skipped a lot of the detail of OpenGL in favour of enabling you to get objects on the screen and experiment with them rather than go through pages and pages of OpenGL history, differences between OpenGL ES and OpenGL etc etc. However, this time I am going to go through some of the technical details that I have skipped in the past.

That, coupled with the fact I need to cover quite a bit of detail, means that this is going to be a *long* tutorial.

Having said that, the majority of the code has to do with loading the texture into our program and getting it into the OpenGL engine so OpenGL can use it. It's not complicated, it just requires a bit of work in the iPhone SDK.

Getting Ready for the Texture

Before we can use a texture, we need to load it into our application, format it for OpenGL, and then tell OpenGL where to find it. Once we have done that, the rest is as easy as colouring our square was in the last tutorial.

Fire up Xcode and open `EAGLView.h` in the editor. First of all we need to provide a variable which OpenGL requires. Add the following declaration:

```
GLuint textures[1];
```

Obviously, this is an array of 1 `GLuint`. You've seen me use `GLfloat` before and, once again, `GLuint` is just an OpenGL connivence `typedef` for an unsigned integer. You should always use the `GLxxxx` typedefs rather than the Objective C types because the OpenGL typedefs have been defined for the OpenGL implementation, not the development environment.

Later on, we will call the OpenGL function `glGenTextures()` to populate this variable. Just remember you've defined it for now and we'll cover `glGenTextures()` and this variable later on.

Down in the method prototypes, add the following method prototype:

```
- (void)loadTexture;
```

This is where we're going to put the code to actually load the texture.

Add CoreGraphics Framework to Your Project

In order to load the texture and process it, we will use the CoreGraphics framework as it provides all the methods we need without needing to write all that low level code you see in Windows OpenGL tutorials.

In the Xcode "Groups & Files" side bar, right click on the "Frameworks" group and choose Add -> Existing Frameworks...

In the search box, enter "CoreGraphics.framework" and look for the folder in the results which matches your application target (iPhone SDK 2.2.1 in my case). Click on the folder, and add it to your project (the folder icon for the framework is fine.

Next, we need to add the texture image to our project so it is included in the application's bundle. Download the texture checkerplate.png and save it to your project directory. Add the image to your project's Resources group by right clicking on the Resources group and selecting Add -> Existing Files... Choose the image and it should appear in the resources group.

Loading the Texture into Our Application and OpenGL

Switch now to `EAGLView.m` and we'll implement the `loadTexture` method.

```
- (void)loadTexture {

}
```

The following code all goes sequentially in this method so just keep adding each line after the other. The first thing we need to do is to get the image into our application by using the following code:

```
CGImageRef textureImage = [UIImage
imageNamed:@"checkerplate.png"].CGImage;
if (textureImage == nil) {
    NSLog(@"Failed to load texture image");
    return;
}
```

A `CGImageRef` is a CoreGraphics data type which collects all the information about the image. To get this information all we do is use the `UIImage` class method `imageNamed`: which creates an `autorelease'd` `UIImage` finding the file by it's name in our Application's main bundle. `UIImage` automatically creates the `CGImageRef` and is accessible by the `UIImage` property `CGImage`.

Now, we just need to get the size of the image for later reference:

```
    NSInteger texWidth = CGImageGetWidth(textureImage);
    NSInteger texHeight = CGImageGetHeight(textureImage);
```

The `CGImageRef` data contains the image's width and height but we can't access it directly, we need to use the above two accessor functions.

The `CGImageRef`, just like the data type's name suggests, does not hold the image data, only a reference to the image's data. So we need to allocate some memory to hold the actual image data:

```
   GLubyte *textureData = (GLubyte *)malloc(texWidth * texHeight
* 4);
```

The correct amount of data to allocate is the width multiplied by the height, **multiplied by 4.** Remember from the last tutorial that OpenGL only accepts `RGBA` values? Each pixel is 4 bytes in size, one byte for each of the `RGBA` values.

Now, we have some absolute mouthfuls of function calls:

```
    CGContextRef textureContext = CGBitmapContextCreate(
                    textureData,
                    texWidth,
                    texHeight,
                    8, texWidth * 4,
                    CGImageGetColorSpace(textureImage),
                    kCGImageAlphaPremultipliedLast);

    CGContextDrawImage(textureContext,
                    CGRectMake(0.0, 0.0, (float)texWidth,
(float)texHeight),
                    textureImage);

    CGContextRelease(textureContext);
```

Firstly, as the function name suggests, this is a CoreGraphics function that returns a Quartz2D graphics context reference. Basically what we're doing is pointing CoreGraphics at our texture data and telling it the format and size of our texture.

Then, we actually draw the image into our allocated data (`textureData` pointer) from the data pointed to context reference we created before. This context contains all the information it needs to copy the data into our `malloc()`'d space in the right format for OpenGL.

We're really finished now with CoreGraphics and we can release the textureContext handle which we created.

I know I've sped through the above code, but we're more interested in the OpenGL side of things. You can reuse that code for any PNG format graphics texture that you add to your project in this way.

Now, onto the OpenGL programming.

Now, remember that variable we declared originally in the header file? We are now going to use it. Have a look at the next line of code:

```
    glGenTextures(1, &textures[0]);
```

We are going to copy the texture data from our application into the OpenGL engine so we need to tell OpenGL to allocate memory for it (we can't do it directly). Remember textures[] was defined as a GLuint? Once we call `glGenTextures`, OpenGL creates a "handle" or "pointer" which is a unique reference to each individual texture we load into OpenGL. The value that OpenGL returns to us isn't important to us, just every time we want to refer to this `checkerplate.png` texture, we just need to refer to textures[0]. We know what we're talking about and so does OpenGL.

We can allocate space for multiple textures at one time. For example, if we needed 10 textures for our application, we can do the following:

```
    GLuint textures[10];
    glGenTextures(10, &textures[0]);
```

For this example, we only need one texture so we're allocating one.

Next we need to activate the texture which we just generated:

```
     glBindTexture(GL_TEXTURE_2D, textures[0]);
```

The second parameter is obvious, it's the texture we just created. The first parameter is *always* `GL_TEXTURE_2D` because that's all OpenGL ES accepts at this point. "Full" OpenGL allows for 1D and 3D textures but I'm sure this is still required in OpenGL ES for future compatibility.

Just remember to use it to activate textures.

Next, we send our texture data (pointed to by `textureData`) into OpenGL. OpenGL manages the texture data over on "it's" side (the server side) so the data is converted in the required format for the hardware implementation, and copied into OpenGL's space. It's a bit of a mouthful but most parameters will always be the same due to the limitations of OpenGL ES:

```
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texWidth, texHeight,
0, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
```

Going through the parameters, they are:

•`target` - basically, this is always `GL_TEXTURE_2D`
•`level` - specifies the level of detail of the texture. 0 means the full detail the image allows, higher numbers goes into nth level mipmap image reduction
•`internal_format` - the `internal format` and `format` listed below **must** be the same. Hence `GL_RGBA` for both.
•`width` - width of the image
•`height` - height of the image
•`border` - must always be set to 0 as OpenGL ES does not support texture borders.
•`format` - must be the same as `internal_format`
•`type` - the type of each pixel. Remember there were four bytes to a pixel? Therefore each pixel is made of 1 unsigned byte (RGBA remember).
•`pixels` - point to the actual image data

So, while there are quite a few parameters, most are either common sense, always the same, or just require you to type in the variables you defined previously (`textureData`, `texWidth`, & `texHeight`). And just remember all you're doing is handing control of you're texture data over to OpenGL.

Now that you've passed the data to OpenGL, you can free the `textureData` that we allocated earlier:

```
    free(textureData);
```

There are only three more function calls to make:

```
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
    glEnable(GL_TEXTURE_2D);
```

These three calls just make final settings to OpenGL and then puts OpenGL in the texture mapping "state".

The first two calls tell OpenGL how to process the textures when magnifying (up close - `GL_TEXTURE_MAG_FILTER`) and minimising (far away - `GL_TEXTURE_MIN_FILTER`). You do need to specify at least one of these for texture mapping to work and the `GL_LINEAR` option has been set for both.

Finally, we just call `glEnable()` to ask OpenGL to use textures when you tell it to in the drawing code.

Finally, we need to add a call to this method within the `initWithCoder` initialiser.

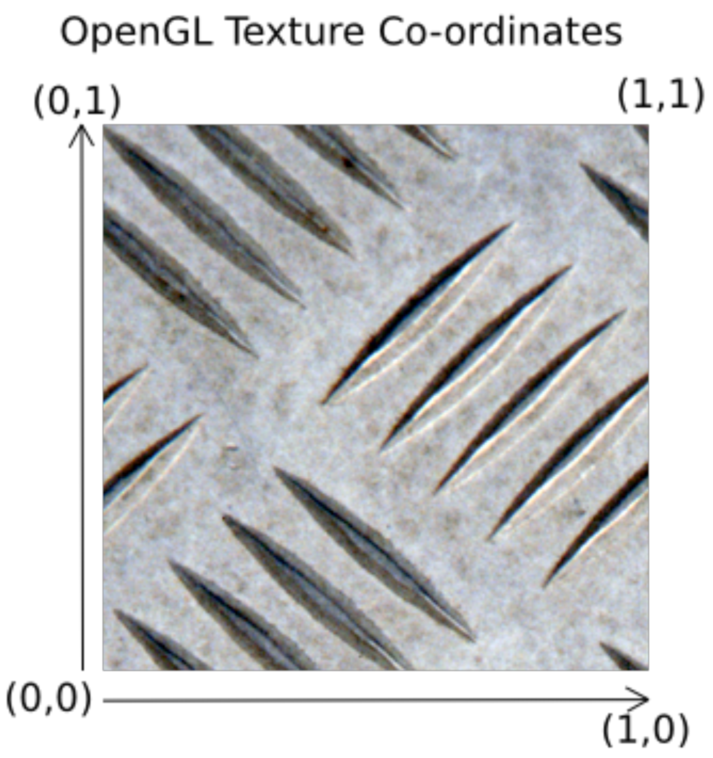
```
        [self setupView];
        [self loadTexture];// Add this line
```

Just add the second line after the call to the `setupView` method.

Adjustments to `drawView`
That's the hard work done. Changes to `drawView` method are no harder than colouring a the square from the previous tutorial. First, comment out the `squareColours[]` array as we won't use it.

Now, remember when we were colouring the square, for each vertex of the square, we provided a colour value. When it comes to texture mapping, we need to do exactly the same thing but instead of telling OpenGL the colour for each vertex, we tell OpenGL what co-ordinate of the **texture** corresponds to that vertex.

Before we can do that, we need to know what are the co-ordinates of the texture. OpenGL locates the origin of a texture (0, 0) at the lower left, with increasing values from 0 -> 1 along each axis. Have a look at the following image of our texture:



Refer this back to our squareVertices[].

```
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
    };
```

Can you see that the first texture co-ordinate we need to specify is for the top left of the texture? That would be texture co-ordinate (0, 1). Our second vertex is the bottom left of the square, therefore it is texture co-ordinate (0, 0). We then go to the bottom right and that's texture co-ordinate (1, 0), finally ending on the top right we end on texture co-ordinate (1, 1). Thus, we specify the `squareTextureCoords[]` as follows:

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

Note we have used `GLshort` instead of `GLfloat` for this. Add the above code to your project.

Can you see how this is similar to what we did with the colour array?

Right, now we need to modify the drawing code. Leave the triangle drawing code as is and start from the `glLoadIdentity()` before the square drawing code. The square drawing code now looks like the following:

```
    glLoadIdentity();
    glColor4f(1.0, 1.0, 1.0, 1.0);      // NEW
    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);

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

    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
// NEW
```

OK, there are four new lines of code, and I've deleted the code for colouring the square from the previous tutorial. The first line is a call to `glColor4f()` which I'll cover in detail below.

The next three should be quite familiar to you by now. Instead of referring to object vertices or colours, we are just referring to textures instead.

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

The first call is to tell OpenGL where our texture co-ordinate array is stored and what format it is in. The difference is that we say there are 2 values for each co-ordinate (it's a 2D texture of course), the data type we used was a `GLushort` which translates to `GL_SHORT` here, there is no `stride` (0), and the pointer to our co-ordinates.

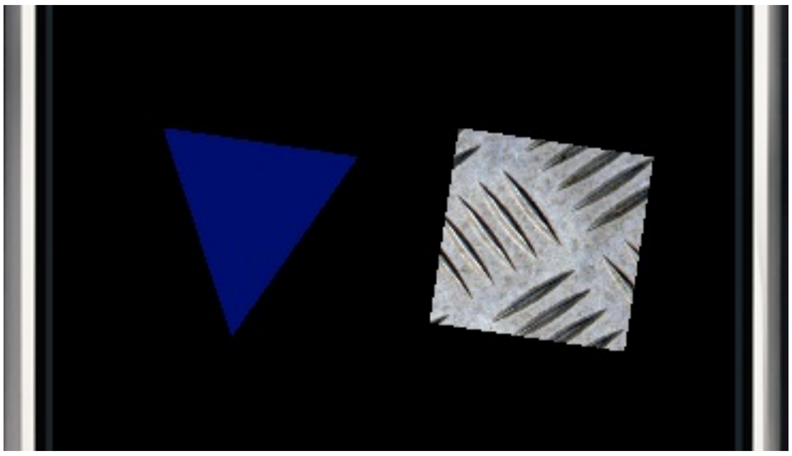
Next we just tell OpenGL to enable the client state for texture mapping with the co-ordinate array we've just specified.

Next the unchanged glDrawArrays() is called before:

```
  glDisableClientState(GL_TEXTURE_COORD_ARRAY);               //
NEW
```

Remember when we were colouring the square differently to the triangle we turned off the colour array? Again, we need to do this for texture mapping otherwise OpenGL will use the texture for the triangle.

Make the changes to the code and hit "Build and Go", you should get the following:



Our checkerplate texture in now mapped onto the square and the triangle is as per normal.

Further Experimentation
First of all, let me talk about the new `glColor4f()` line which we added to the square drawing code:

```
    glColor4f(1.0, 1.0, 1.0, 1.0);      // NEW
```

This of course changes the drawing colour to white, fully opaque. Can you guess why I added this line? OpenGL is of course a "state" machine so once we set something, it stays in that state until we tell it otherwise. So the colour was set as the blue until we made it white.

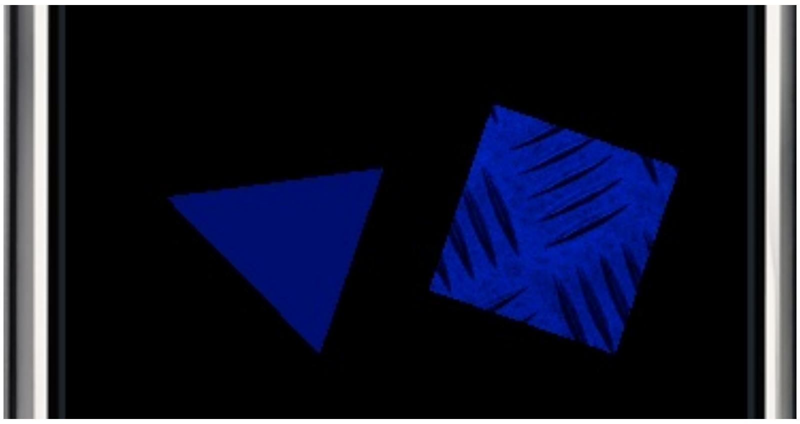
OK, now when texture mapping, OpenGL performs a multiplication between the current colour set (the blue) and the current texture pixel. That is:

```
                          R    G    B    A
    Colour Set:           0.0, 0.0, 0.8, 1.0
    Texture Pixel Colour: 1.0, 1.0, 1.0, 1.0
```

So, when OpenGL draws that pixel, it multiples

```
     Colour_Red * Pixel_Colour_Red =
Rendered_colour
        0.0      *     1.0             = 0.0
    Colour_Green * Pixel_Colour_Green
        0.0      *     0.0             = 0.0
     Colour_Blue * Pixel_Colour_Blue
        0.8      *     1.0             = 0.8
```

Comment out that `glColor4f()` before the square drawing code, you should get something like this:



When we had white, the multiplation would have been:

```
Set Colour :     1.0, 1.0, 1.0, 1.0
                  is mulitplied by
Pixel Colour :   0.8, 0.8, 0.8, 1.0

         Result: 0.8, 0.8, 0.8, 1.0
```

So that’s why we need to set the colour as white.

OK, That’s It!
That will do for this tutorial, I know I’ve covered a lot but I hope you can see that the actual texture mapping code is not really significant, it’s more the loading and setting up the texture that really takes the time. Next time, we’re off into texture mapped 3D objects, blending and other fun stuff.

As usual, here’s the code and you can email me with any questions.

AppleCoder-OpenGLES-05.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

8970

Made on a Mac

Email Me