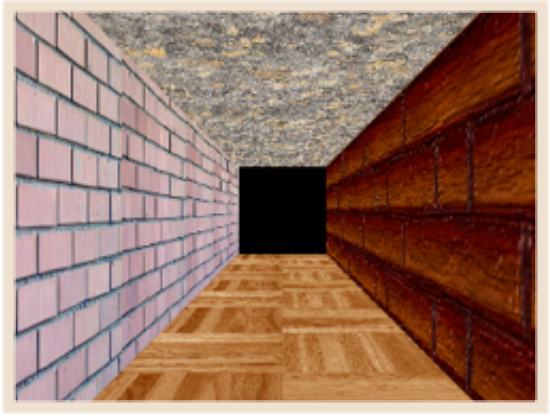
SIMON MAURICE - IPHONE - OPENGL ES



Thursday, 23 April 2009

OpenGL ES 11 - Single Texture, Multiple Looks, Render to Texture, and Getting Inspired in Maths

In my last year of high school, I was doing an advanced maths subject which was as interesting as watching paint dry. There were quite a few times I just didn't bother turning up because teaching me those advanced concepts in a classroom environment just didn't inspire me.

One day, sat up the back of the class as always, and thinking about something more interested, I hadn't even heard what was being taught that day. Wasn't interested as I knew I could learn it in my own time. Except this day. I looked up at the whiteboard and my jaw must have dropped. What was on the whiteboard was essentially this:

$$[\,a\ b\ c\,] \,.\, [\,d\ e\ f\,]$$

I sat bolt upright. I knew that was matrix maths, something which I had been exposed to in reading about graphics programs but I hadn't quite understood. Well, I was interested once again, and inspired to pay attention and even enjoy myself.

Inspiration is a funny thing. I mention this because no matter how hard

I tried, I couldn't quite get that tutorial on blending right. So I just posted it and decided to come back later when I was all "inspired".

The reality will be that I'll never be inspired looking at that cube and pyramid, trying to deal with increasingly complex OpenGL ES concepts. Its time to start to (slowly) increase the complexity of the scenes in what we're experimenting in. That way, the inspiration shouldn't run dry.

The increase in scene complexity will move along slowing, we can't make any huge jumps at any single time for fear of leaving people behind. But increase the complexity we will and the outcome for you will be learning scene management, as well as concepts in OpenGL ES.

Today's tutorial will just set us in a tunnel. But before we hit the scene, we have some other bits of housekeeping to do. First of all, we need cover one final topic on loading textures in the iPhone, something I left until last as it was the least important (read: I forgot about it). Then we're going to draw our tunnel using 4 different textures, but loaded from a single texture file. Finally, we'll do some basic render to texture, touching base with blending once again.

Let's get cracking. Here's the base project for this tutorial, the completed project is at the end. OpenGLES11 Start.zip

iPhone's Y != OpenGL's Y
OpenGL's Y co-ordinates are at odds with many platforms, including the iPhone's. What I mean is that when we use CoreGraphics to load a texture, it's actually rendered by OpenGL upside down compared to what we want (ie rotated 180º around the X axis).

To fix this, do not pass `glRotatef()` function calls. That would get the result on the screen but it would also waste CPU/GPU time.

The only resolutions to this would be:

1. Alter the texture before bundling in an image editor so they are loaded the right way. This works but has the potential of reducing cross platform compatibility. Also requires more work. I would not do this.
2. Alter your texture co-ordinate array so that it references the right co-ordinates. That is, the bottom left of the image would be (0, 1). OpenGL doesn't care how the data is for your texture, it just needs to know where to find the texture co-ordinate for each vertex. Again, it works but doesn't really give a solid solution.

Change the `loadTexture[]` method so that the image is flipped when loaded. This works but adds loading time to your app. Each time

you load the textures for a level of a game for example, the textures are altered either by the CPU or GPU (depending on implementations).

"Best practice" is probably option 3 because that is where the platform specific code for the textures is contained. You may not necessarily be programming on a platform which has to deal with this, or may need to deal with it differently.

The real world answer depends on what you're doing. If you're writing an iPhone app which may go to another platform later, then I'd just set the texture co-ordinate array to deal with it. There will be no performance penalty on loading or rendering, and no extra work beforehand in an image editor.

I'm going to change the `loadTexture[]` method. For the purposes of these tutorials, it's all good. Just don't make OpenGL do the work for you.

The only changes we need to make to the `loadTexture[]` is call

```
    CGContextRef textureContext =
CGBitmapContextCreate(textureData,
                                    texWidth, texHeight,
                                    8, texWidth * 4,

CGImageGetColorSpace(textureImage),

kCGImageAlphaPremultipliedLast);

// Rotate the image -- These two lines are new
CGContextTranslateCTM(textureContext, 0, texHeight);
CGContextScaleCTM(textureContext, 1.0, -1.0);
```

That gets our images set the right way around.

Multiple Textures from a Single Texture
As part of the fun today, we're going to use just one single texture (ie one single call to the `loadTexture[]` method) but get four separate textures out of it: two wall textures, a floor texture, and a ceiling texture.

In the last tutorial, we loaded six individual textures, one for each side of the cube. Each time we load a texture and send it to OpenGL, it makes a copy of it so the texture resides in OpenGL's controlled space. Hence we are able to `free()` the memory we had reserved with `malloc()` for the texture image in the `loadTexture[]` method. Obviously, each time we copy a texture it takes up time, we need to execute a call to bind OpenGL to a new texture each time we want to change it etc.

This method does give us some efficiencies but it's not really about

performance. Whilst is a handy skill, I just want to show you ways you can use parts of textures.

The first thing I did was to create a new blank image in the Gimp with a size of 512x512 pixels, room enough for four 256x256 textures. Then I just placed each of the four textures onto this canvas so they didn't overlap. Saving it as a PNG file, I of course got this:

At least some of you out there by now (hopefully all of you) will have worked out that I'll be able to access each individual texture by controlling my texture co-ordinates.

Before we can define our texture co-ordinates, we need to define what the vertices are going to be. We only need to describe a single square for all of them like follows:

```
const GLfloat elementVerticies[] = {
    -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
};
```

OK, so if we wanted to render the wood texture (that's the top left square from the combined texture image above), we would need to

specify the following texture co-ordinates:

```
const GLfloat combinedTextureCoordinate[] = {
    // The wood wall texture
    0.0, 1.0,       // Vertex[0~2] top left of square
    0.0, 0.5,       // Vertex[3~5] bottom left of square
    0.5, 0.5,       // Vertex[6~8] bottom right of square
    0.5, 1.0,       // Vertex[9~11] top right of square
```

As before, they are just co-ordinate pairs. Each co-ordinate pair indicates to OpenGL where in the texture to obtain the texture pixel for each vertex (OpenGL just interpolates between each vertex to fill it).

Since the "bottom" of the wood texture is half the height of our full sized texture, the Y co-ordinate is 0.5 instead of 0.0 like previously. Actually in the past we used `GLshort` data for the texture co-ordinates but now we need to use `GLfloats` to represent fractional values.

Similarly, the X co-ordinate which represents the end of the wood texture (the rightmost edge) is half way along our loaded texture and thus has a co-ordinate of 0.5.

Here's the full co-ordinate array for our texture:

```
const GLfloat combinedTextureCoordinate[] = {
    // The wood wall texture
    0.0, 1.0,       // Vertex[0~2] top left of square
    0.0, 0.5,       // Vertex[3~5] bottom left of square
    0.5, 0.5,       // Vertex[6~8] bottom right of square
    0.5, 1.0,       // Vertex[9~11] top right of square

    // The brick texture
    0.5, 1.0,
    0.5, 0.5,
    1.0, 0.5,
    1.0, 1.0,

    // Floor texture
    0.0, 0.5,
    0.0, 0.0,
    0.5, 0.0,
    0.5, 0.5,

    // Ceiling texture
    0.5, 0.5,
    0.5, 0.0,
    1.0, 0.0,
    1.0, 0.5
};
```

I hope that's straight forward now you can see the full co-ordinate array. The co-ordinate array refers to the co-ordinates inside the texture so they don't even need to be edge bound by the four textures we started with.

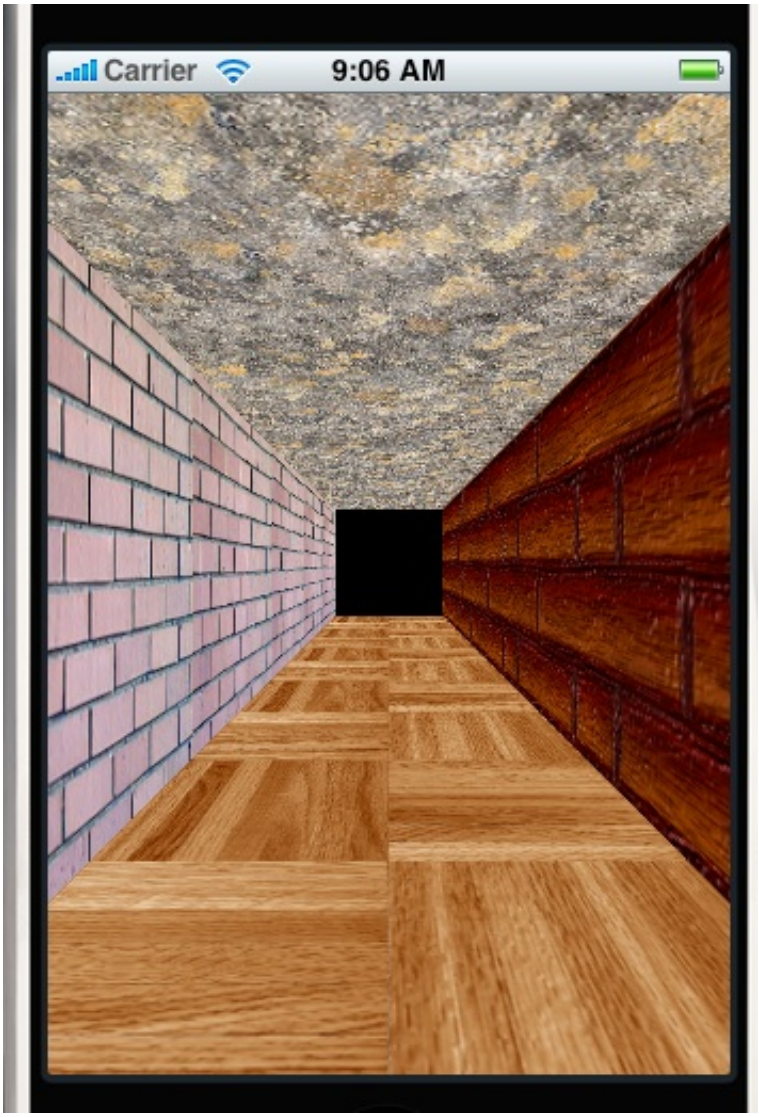
One final thing. I've made some `#define`'s to make code reading easier for getting the correct offset in the array for each texture's co-ordinates:

```
#define WOOD_TC_OFFSET     0
#define BRICK_TC_OFFSET    8
#define FLOOR_TC_OFFSET    16
#define CEILING_TC_OFFSET  24
```

Drawing the Tunnel

I'm probably going to disappoint some of you right now. The tunnel you see is not dynamically rendered. It's static. Soon enough, we'll be drawing from a 3D world map and be able to move inside the tunnel (and other rooms) but we're going to build up to it, introducing new OpenGL concepts on the way. But for now, we'll just draw it in a straightforward manner.

This is what the rendered tunnel looks like:



The first thing to note is that we only have one object. The drawing code for the tunnel doesn't create any more objects, we just use the

single one that we have and manipulate it to look like there multiple. In fact there are 5 on the left wall, 5 on the right wall, and 10 each for the floor and ceiling (only the centre halves for each are viewable).

So I guess it's fair to say this tutorial covers a fair bit of code reuse!

Before we can draw anything, we need to set the state of OpenGL to draw texture mapped squares. The code, you should be familiar with by now is:

```
    glBindTexture(GL_TEXTURE_2D, textures[0]);
    glVertexPointer(3, GL_FLOAT, 0, elementVerticies);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnable(GL_TEXTURE_2D);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

All we've done is told OpenGL about our vertex array, told it to use it, turned on texture mapping and told it to use it. We can't tell OpenGL about our texture co-ordinate array because that will change depending on what structure in the scene we are drawing (eg floor, ceiling, or walls).

Firstly, we'll draw the floor. The floor consists of two of the squares we defined earlier, moved down to the lower half of the screen, rotated 90º so is like a floor, and drawn side by side. The code looks like this:

```
    // Draw the Floor
    // First, point the texture co-ordinate engine at the right
offset
    glTexCoordPointer(2, GL_FLOAT, 0,
&combinedTextureCoordinate[FLOOR_TC_OFFSET]);
    for (int i = 0; i < 5; i++) {
        glPushMatrix();
        {
            glTranslatef(-1.0, -1.0, -2.0+(i*-2.0));
            glRotatef(-90.0, 1.0, 0.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();

        glPushMatrix();
        {
            glTranslatef(1.0, -1.0, -2.0+(i*-2.0));
            glRotatef(-90.0, 1.0, 0.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();
    }
```

We have a for loop which repeats the process five times. Each time the loop executes, two squares of our floor are drawn. It's just worth pointing out the change to the call to `glTexCoordPointer()`:

```
glTexCoordPointer(2, GL_FLOAT, 0,

&combinedTextureCoordinate[FLOOR_TC_OFFSET]);
```

The parameter for the pointer to the array needs to be passed the correct address of the starting co-ordinate. Hence the #defines earlier to make reading the code easier.

If you're following along in Xcode, you can hit "Build & Go" and you'll see the following in the simulator:

If you're not following along in Xcode, then you'll just have to take my word for it... :)

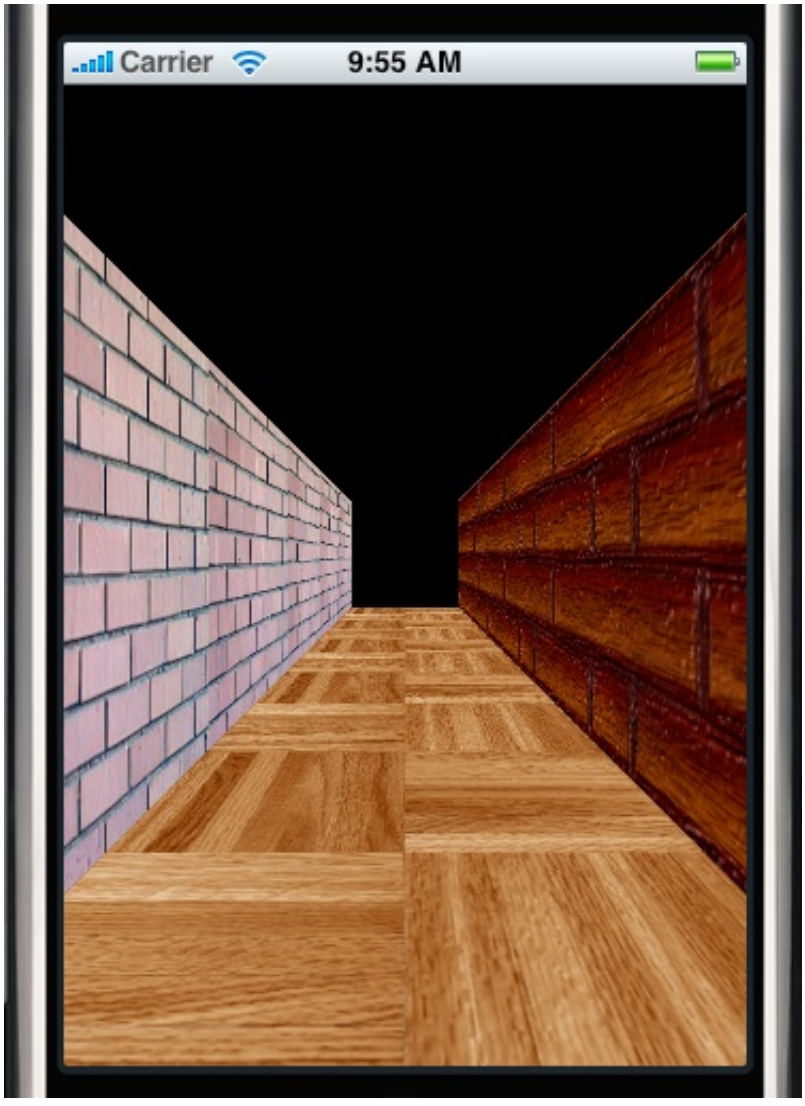
Now some walls. The process is the same but this time we move them along the X axis only and rotate them on the Y axis.

```
    // Draw the walls
    // This time we'll change the texture coordinate array during
the drawing
    for (int i = 0; i < 5; i++) {
        glPushMatrix();
        {
            glTexCoordPointer(2, GL_FLOAT, 0,
&combinedTextureCoordinate[BRICK_TC_OFFSET]);
            glTranslatef(-1.0, 0.0, -2.0+(i*-2.0));
            glRotatef(-90.0, 0.0, 1.0, 0.0);
```

```
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();

        glPushMatrix();
        {
            glTexCoordPointer(2, GL_FLOAT, 0,
&combinedTextureCoordinate[WOOD_TC_OFFSET]);
            glTranslatef(1.0, 0.0, -2.0+(i*-2.0));
            glRotatef(-90.0, 0.0, 1.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();
    }
```

Note where I've put the call to change the texture co-ordinate array's offset? That's because I have used a different texture between the left and the right walls.

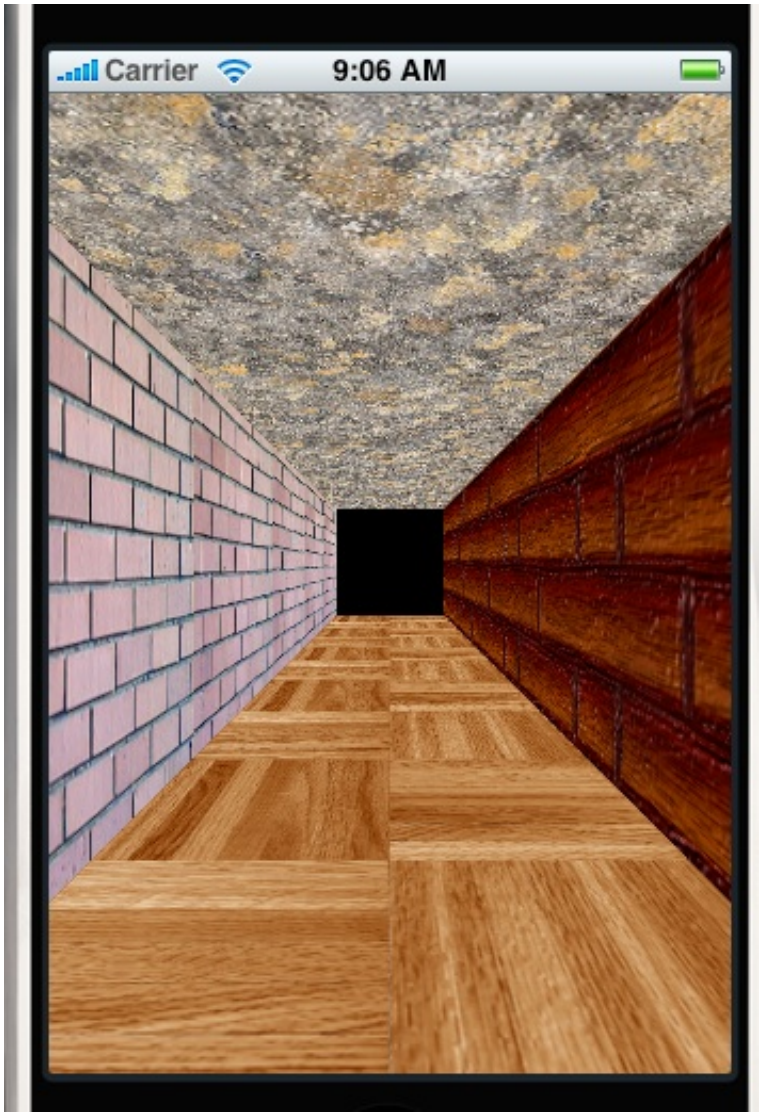


Finally it's time to draw the ceiling. Same deal as the floor, just increased in the Y direction rather than decreased:

```
    // Draw the ceiling
```

```
    // Start by setting the texture coordinate pointer
    glTexCoordPointer(2, GL_FLOAT, 0,
&combinedTextureCoordinate[CEILING_TC_OFFSET]);
    for (int i = 0; i < 5; i++) {
        glPushMatrix();
        {
            glTranslatef(-1.0, 1.0, -2.0+(i*-2.0));
            glRotatef(90.0, 1.0, 0.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();
        glPushMatrix();
        {
            glTranslatef(1.0, 1.0, -2.0+(i*-2.0));
            glRotatef(90.0, 1.0, 0.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();
    }
```

That gives us our completed tunnel.



Granted, its nothing special at the moment but will give us a decent place in which to begin experimenting. From here we can start to deal with topics such as scene management, rendering floors (or landscapes)

and walls from a world map etc.

Render to Texture: The Easy Way
Render to texture is the process of creating a new texture (exactly like one we would get from `loadTexture[]`) from what has already been rendered in our scene. So instead of loading a texture from our app's bundle, we let OpenGL render a scene, then we copy all or part of that scene into a buffer, then add it back it before presenting it as a finished screen.

Whew! Take a deep breath. It's not as complex as what it sounds, and more useful than it sounds.

An easy example of render to texture is to produce a reflection of part of your render buffer. You would want to do this because when it's been rendered to the buffer, effects such as lighting, transparency and blending have already been done. So if you were to copy that and use it for a reflection effect, you would not need to recalculate all the lighting, shadows etc for the reflected image.

I wasn't going to cover render to texture yet as it's probably a bit further on than where I'm up to so far, but there's been a few requests so I will cover the easy way to achieve this effect now. We won't see the best yet because we have not yet covered details such as lighting but you'll get the idea.

In our tunnel world, we're going to add a couple more texture mapped squares at the end of the tunnel and use render to texture to produce a reflection effect (albeit cheaply).

First thing, we'll add the new objects.

That's Not Light; There's Romo at the End of my Tunnel
First thing we're going to do is to load two new textures. One is just for a background effect, then we're going to blend a second texture of Tony Romo, our hero, in front of it, just so I can show the render to texture effect at least with blending (which so far I have covered very poorly but covered quickly nonetheless).

Add the two new lines to your `initWithCoder[]` method:

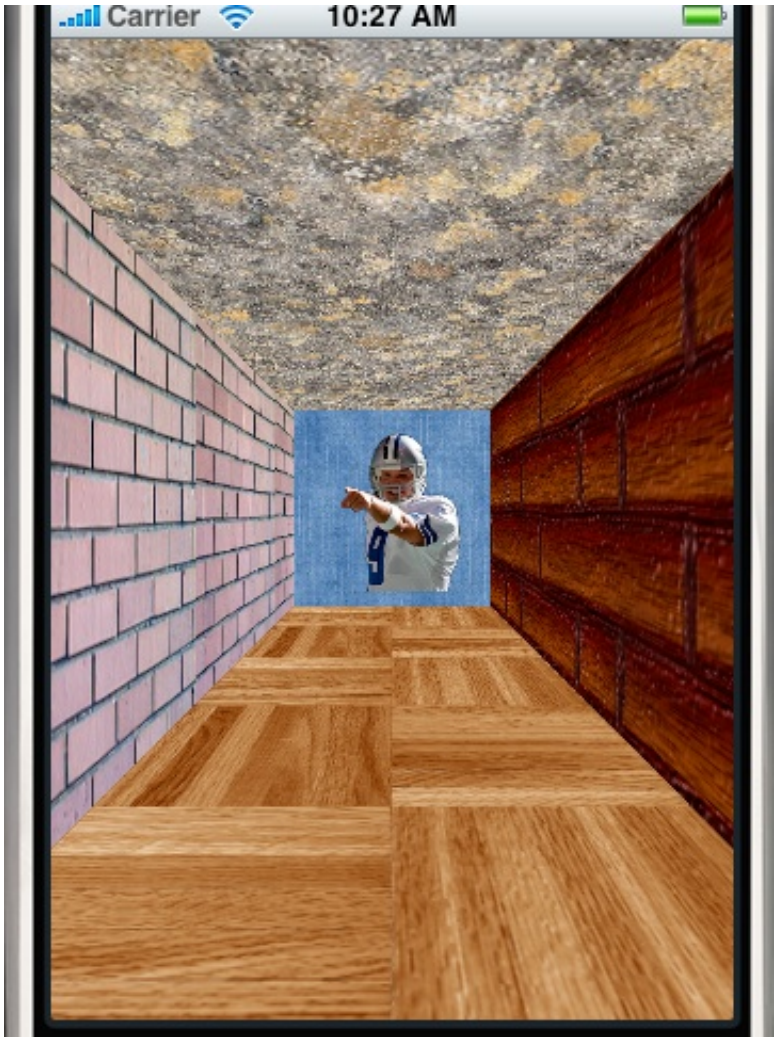
```
[self loadTexture:@"bluetex.png" intoLocation:textures[1]];
[self loadTexture:@"romo.png" intoLocation:textures[2]];
```

With our textures loaded, after drawing the ceiling, we will then generate the image for the end of the tunnel. First we need to set up a "standard" texture co-ordinate array for both of our textures, then we draw the opaque texture's square first, followed by the partially transparent textured square using blending.

```
    const GLfloat standardTextureCoordinates[] = {
        0.0, 1.0,
        0.0, 0.0,
        1.0, 0.0,
        1.0, 1.0
    };
    // Draw the Blue texture
    glBindTexture(GL_TEXTURE_2D, textures[1]);
    glTexCoordPointer(2, GL_FLOAT, 0,
standardTextureCoordinates);
    glPushMatrix();
    {
        glTranslatef(0.0, 0.0, -6.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();
    glBindTexture(GL_TEXTURE_2D, textures[2]);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glPushMatrix();
    {
        glTranslatef(0.0, 0.0, -5.9);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();
    glDisable(GL_BLEND);
```

Having had an abortive tutorial on blending, this is a much better example. Remember from the blending tutorial that you needed to draw the opaque object first followed by the partially transparent object in front of it? That's all we did. The image of Tony Romo is just one which I quickly deep etched so that despite being the same image size as the blue texture, doesn't cover it up.

The above code produces the following image:



By way of an example of this blending. Go back to the code and comment out the `glEnable(GL_BLEND)`, you only need to comment out the one line. Here's what you get:

Notice now that the textured image of Romo obscures the blue textured square behind it? That's non-blended. I need you to see this to show you that render to texture takes part of an image we have *rendered already*. Including all effects.

Un-comment the line `glEnable(GL_BLEND)`. We'll leave this effect in for the render to texture. Now that we're set up, I'll talk you through the render to texture process.

Render to Texture: The Process
The process for executing render to texture is straightforward (well, the method we're covering today is). It's a 4 step process:

1. Create the render texture which is our target texture to which we copy the contents of our rendered image. You do this in almost exactly the same way as you do for a texture you would load with `loadTexture[]`.
2. Render your scene. We've done that; we've got Tony Romo (our hero) in a tunnel with some blending effect to prove that we're copying from a rendered section of our buffer.
3. Copy the portion of the image that we want (or the entire image)

into our texture that we set up in step 1.
4. Render the new texture.

Now that you know the four steps, only step 3 is completely new, the others will have new elements but will be very familiar.

Create the Render Texture
Normally for any image we want to use as a texture, it's just a case of loading it, formatting it, then sending it to OpenGL via a call to `glTexImage2D()`. In the case of a render texture, we don't need to do much of that but we still need to get OpenGL to allocate space in it's operational area to store the pixels copied from the rendered scene. It's easy. It's simply a case of heading to `initWithCoder[]` and, after the loading of the textures, add the following lines of code for the render texture:

```
// Render to Texture texture buffer setup
glBindTexture(GL_TEXTURE_2D, textures[3]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA,
                                                    GL_UNSIGNED_BYTE,
nil);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

The first thing we do is, as per normal, tell OpenGL which texture we are working with by choosing the next available texture id. Then, the next line, we are creating the memory for it in almost exactly the same way as we created a texture which we loaded from a file.

There are two differences. First, we've hard coded the texture size of 128x128 pixels; just a size which sounded good to me at the time, you can choose your own texture size if you like (note, *the texture size must still a power of 2*).

The final parameter I've set to `nil` (or `NULL` if you prefer). The final parameter normally contains a buffer which contains our image data. Since we have no image data, we just let OpenGL know that and it will create the texture for itself but not fill any data into the allocated memory.

Finally we just set the filter parameters and we're done with step one.

Render the Scene
This part is self explanatory. All we do is write everything to our render buffer and then go to step 3.

Copy the Portion of Our Render Buffer to Our Render Texture
There are multiple ways to do this. In this tutorial I'm only going to discuss one way which is almost the fastest but guaranteed to work on all implementations (ie does not require extensions, even if those

extensions are now fairly commonplace). It also has the advantage of being easy to implement.

We're going to copy a subsection of our render buffer, at the same time grabbing all the effects from blending, and, if we had others, those effects too.

Down in the draw view method after all the previous drawing code we can add the following lines of code:

```
    glBindTexture(GL_TEXTURE_2D, textures[3]);
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 100, 150, 128,
128);
```

Believe or not, you've just done a render to texture. The first line is well known, we just need to tell OpenGL which texture we want it to work on. The call to `glCopyTexSubImage2D()` is what does the heavy lifting. This function takes a subsection of the current render buffer and copies a sub-portion of a size and origin into the active texture.

In the case above, we've copied a portion at origin (100, 150) of size 128x128 pixels into our texture. The parameters for this are:

```
glCopyTexSubImage2D(
```

target	always `GL_TEXTURE_2D` in OpenGL ES
level	detail level, will cover when we hit MIP maps
xoffset & yoffset	in the target, specifies if you want the copied pixels to start somewhere other than (0, 0)
x & y	this is the source (render buffer) origin co-ordinates originating at the bottom left (0,0)
width & height	Size of the image to copy

```
);
```

Just remember that since you're copying from a 2D view, there is no Z co-ordinates. Also you're 3D world co-ordinates do not apply here, you need to convert them back into view space. That's something for another day as this one is getting really long already and iWeb is starting to lag behind as I type!! :)

Anyway, texture created from render, now we can just use as we please.

Rendering our New Texture
The final step should require no explanation. Here's the code:

```
    glPushMatrix();
```

```
{
    glTranslatef(0.0, -1.0, -2.0);
    glRotatef(-75.0, 1.0, 0.0, 0.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
```

All we've done is moved it around a little and rotated it so it's floating in space in front of us. Here's what we get:

There we have Mr Romo, our hero, render to texture. Done.

Conclusion of Another

OK, that's it for today. As usual you can check in at the iphonedevsdk.com forum website in the tutorials section to keep up with this series. Emails directly to me a are of course welcome (link below).

Soon enough, we'll have this done in complete 3D.

As usual, here's the code for this tutorial:

OpenGLES11.zip

Until next time, hooroo!
Simon Maurice

previous next