raywenderlich.com

Beginning OpenGL ES 2.0 with GLKit Part 2

Ray Wenderlich on October 21, 2011

Note from Ray: This is the fifth iOS 5 tutorial in the iOS 5 Feast! This tutorial is a free preview chapter from our new book iOS 5 By Tutorials. Enjoy!

Welcome back to our Beginning OpenGL ES 2.0 with GLKit series!

In the first part of the series, we showed you how to create a simple GLKView and GLKViewController based project from scratch. In the process, you learned all about what these classes can do for you, and why you'd want to use them.

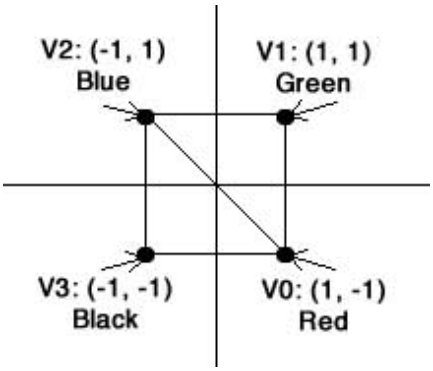
In this second and final part of the tutorial, now that you know the basics we can get to the fun stuff! We'll use some OpenGL commands and a new GLKit class called GLKBaseEffect to render a square to the screen. And then we'll use some of the new handy functions in the GLKMath library to make it rotate around!

This tutorial continues where the last one left off, so if you haven't done it already complete the previous tutorial first.

OK, time to draw something to the screen with OpenGL!

Creating Vertex Data for a Simple Square

Let's start things nice and simple by rendering a square to the screen. The square will be set up like the following:



When you render geometry with OpenGL, keep in mind that it can't render squares – it can only render triangles. However we can create a square with two triangles as you can see in the picture above: one triangle with vertices (0, 1, 2), and one triangle with vertices (2, 3, 0).

One of the nice things about OpenGL ES 2.0 is you can keep your vertex data organized in whatever manner you like. Open up HelloGLKitViewController.m and create a plain old C-structure and a few arrays to keep track of our square information, as shown below:

```
typedef struct {
    float Position[3];
    float Color[4];
} Vertex;

const Vertex Vertices[] = {
    {{1, -1, 0}, {1, 0, 0, 1}},
    {{1, 1, 0}, {0, 1, 0, 1}},
```

```
    {{-1, 1, 0}, {0, 0, 1, 1}},
    {{-1, -1, 0}, {0, 0, 0, 1}}
};

const GLubyte Indices[] = {
    0, 1, 2,
    2, 3, 0
};
```

So basically we create:

- a structure to keep track of all our per-vertex information (currently just color and position)
- an array with all the info for each vertex
- an array that gives a list of triangles to create, by specifying the 3 vertices that make up each triangle

We now have all the information we need, we just need to pass it to OpenGL!

Creating Vertex Buffer Objects

The best way to send data to OpenGL is through something called Vertex Buffer Objects.

Basically these are OpenGL objects that store buffers of vertex data for you. You use a few function calls to send your data over to OpenGL-land.

There are two types of vertex buffer objects – one to keep track of the per-vertex data (like we have in the Vertices array), and one to keep track of the indices that make up triangles (like we have in the Indices array).

So first add the following instance variables to your private HelloGLKitViewController category:

```
GLuint _vertexBuffer;
GLuint _indexBuffer;
```

Then add a method above viewDidLoad to create these:

```
- (void)setupGL {

    [EAGLContext setCurrentContext:self.context];

    glGenBuffers(1, &_vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);

    glGenBuffers(1, &_indexBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, GL_STATIC_DRAW);

}
```

You can see that it's pretty simple. The first thing it does is set the current OpenGL context to the current context. This is important in case some other code has changed the global context.

It then calls glGenBuffers to create a new Vertex Buffer object, glBindBuffer to tell OpenGL "hey when I say GL_ARRAY_BUFFER, I really mean vertexBuffer", and glBufferData to send the data over to OpenGL-land.

Also add a new method to delete the vertex and index buffers:

```
- (void)tearDownGL {

    [EAGLContext setCurrentContext:self.context];

    glDeleteBuffers(1, &_vertexBuffer);
    glDeleteBuffers(1, &_indexBuffer);

}
```

And before we forget, add this to the bottom of viewDidLoad:

```
[self setupGL];
```

And this inside viewDidUnload, right after the call to [super viewDidUnload]:

```
[self tearDownGL];
```

Introducing GLKBaseEffect

In OpenGL ES 2.0, to render any geometry to the scene, you have to create two tiny little programs called shaders.

Shaders are written in a C-like language called GLSL. Don't worry too much about studying up on the reference at this point – you don't even need them for this tutorial, for reasons you'll see shortly!

There are two types of shaders:

- **Vertex shaders** are programs that get called once per vertex in your scene. So if you are rendering a simple scene with a single square, with one vertex at each corner, this would be called four times. Its job is to perform some calculations such as lighting, geometry transforms, etc., figure out the final position of the vertex, and also pass on some data to the fragment shader.
- **Fragment shaders** are programs that get called once per pixel (sort of) in your scene. So if you're rendering that same simple scene with a single square, it will be called once for each pixel that the square covers. Fragment shaders can also perform lighting calculations, etc, but their most important job is to set the final color for the pixel.

GLKBaseEffect is a helper class that implements some common shaders for you. The goal of GLKBaseEffect is to provide most of the functionality available in OpenGL ES 1.0, to make porting apps from OpenGL ES 1.0 to OpenGL ES 2.0 easier.

To use a GLKBaseEffect, you do the following:

1. **Create a GLKBaseEffect**. Usually you create one of these when you create your OpenGL context. You can (and should) re-use the same GLKBaseEffect for different geometry, and just reset the properties. Behind the scenes, GLKBaseEffect will only propogate the properties that have changed to its shaders.
2. **Set GLKBaseEffect properties**. Here you can configure the lighting, transform, and other properties that the GLKBaseEffect's shaders will use to render the geometry.
3. **Call prepareToDraw on the GLKBaseEffect.** Any time you change a property on the GLKBaseEffect, you need to call prepareToDraw prior to drawing to get the shaders set up properly. This also enables the GKBaseEffect's shaders as the current shader program.
4. **Enable pre-defined attributes.** Usually when you make your own shaders, they take parameters called attributes and you write code to get their IDs. For GLKBaseEffect's built in shaders, these are already predefined as constants such as GLKVertexAttribPosition or GLKVertexAttribColor. So you need to enable any parameters that you want to pass in to the shaders, and give them pointers to data.
5. **Draw your geometry.** Once you have everything set up, you can use normal OpenGL draw commands such as glDrawArrays or glDrawElements, and it will be rendered using the effect you've set up!

The nice thing about GLKBaseEffect is if you use them, you don't necessarily have to write any shaders at all! Of course you're still welcome to if you'd like – and you can mix and match and render some things with GLKBaseEffect, and some with your own shaders. If you look at the OpenGL template project, you'll see an example of exactly that!

In this tutorial, we're going to focus on just using GLKBaseEffect, since the entire point is to get you up-to-speed with the new GLKit functionality – plus it's plain easier!

So let's walk through the steps one-by-one in code.

1) Create a GLKBaseEffect

The first step is to simply create a GLKBaseEffect. Up in your private HelloGLKitViewController category, add a property for a GLKBaseEffect:

```
@property (strong, nonatomic) GLKBaseEffect *effect;
```

And synthesize it after the @implementation below:

```
@synthesize effect = _effect;
```

Then in setupGL, initialize it right after calling [EAGLContext setCurrentContext:⋯]:

```
self.effect = [[GLKBaseEffect alloc] init];
```

And set it to nil at the bottom of tearDownGL:

```
self.effect = nil;
```

Now that we've created the effect, let's use it in conjunction with our vertex and index bufferes to render the square. The first step is to set our effect's projection matrix!

2) Set GLKBaseEffect properties

The first property we need to set is the projection matrix. A projection matrix is how you tell the CPU how to render 3D geometry onto a 2D plane. Think of it as drawing a bunch of lines out from your eye through each pixel in your screen. Whatever the frontmost 3D object each line hits determines the pixel that is drawn to the screen.

GLKit provides you with some handy functions to set up a projection matrix. The one we're going to use allows you to specify the field of view along the y-axis, the aspect ratio, and the near and far planes:

[Illustration from Vicki]

The field of view is similar to camera lenses. A small field of view (for example 10) is like a telephoto lens – it magnifies images by "pulling" them closer to you. A large field of view (for example 100) is like a wide angle lens – it makes everything seem farther away. A typical value to use for this is around 65-75.

The aspect ratio is the aspect ratio you want to render to (i.e. the aspect ratio of the view). It uses this in combination with the field of view (which is for the y-axis) to determine the field of view along the x-axis.

The near and far planes are the bounding boxes for the "viewable" volume in the scene. So if something is closer to the eye than the near plane or further away than the far plane, it won't be rendered. This is a common problem to run into – you try and render something and it doesn't show up. One thing to check is that it's actually between the near and far planes.

Let's try this out – add the following code to the bottom of update:

```
float aspect = fabsf(self.view.bounds.size.width / self.view.bounds.size.height);
GLKMatrix4 projectionMatrix = GLKMatrix4MakePerspective(GLKMathDegreesToRadians(65.0f),
aspect, 4.0f, 10.0f);
self.effect.transform.projectionMatrix = projectionMatrix;
```

In the first line, we get the aspect ratio of the GLKView.

In the second line, we use a built in helper function in the GLKit math library to easily create a perspective matrix for us – all we have to do is pass in the parameters discussed above. We set the near plane to 4 units away from the eye, and the far plane to 10 units away.

In the third line, we just set the projection matrix on the effect's transform property!

We need to set one more property now – the modelViewMatrix. The modelViewMatrix is the transform that is applied to any geometry that the effect renders.

The GLKit math library once again comes to the rescue here with some really handy functions that make performing translations, rotations, and scales easy, even if you don't know much about matrix math. To see what I mean, add the following lines to the bottom of update:

```
GLKMatrix4 modelViewMatrix = GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);
_rotation += 90 * self.timeSinceLastUpdate;
modelViewMatrix = GLKMatrix4Rotate(modelViewMatrix, GLKMathDegreesToRadians(_rotation), 0, 0,
1);
self.effect.transform.modelviewMatrix = modelViewMatrix;
```

If you remember back to where we set up the vertices for the square, remember that the z-coordinate for each vertex was 0. If we tried to render it with this perspective matrix, it wouldn't show up because it's closer to the eye than the near plane!

So the first thing we need to do is to move this backwards. So in the first line, we use the GLKMatrix4MakeTranslation function to create a matrix for us that translates 6 units backwards.

Next, we want to make the cube rotate for fun. So we increment an instance variable that keeps track of the current rotation (which we'll add in a second), and use the GLKMatrix4Rotate method to modify the current transformation by rotating it as well. It takes radians, so we use the GLKMathDegreesToRadians method to that conversion. Yes, this math library has just about every matrix and vector math routine you'll need!

Finally, we set the model view matrix on the effect's transform property.

Before we forget, add the rotation instance variable to your HelloGLKitViewController's private category:

```
float _rotation;
```

We'll play around with more GLKBaseEffect properties later, since there's a lot of cool stuff and we've barely scratched the surface here. But let's continue on for now, so we can finally get something rendering!

3) Call prepareToDraw on the GLKBaseEffect

This step is about as simple as it gets. Add the following line to the bottom of glkView:drawInRect:

```
[self.effect prepareToDraw];
```

w00t! Just remember that you need to call this after any time you change properties on a GLKBaseEffect, before you draw with it.

4) Enable pre-defined attributes

Next add this code to the bottom of glkView:drawInRect:

```
glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBuffer);

glEnableVertexAttribArray(GLKVertexAttribPosition);
glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (const
GLvoid *) offsetof(Vertex, Position));
glEnableVertexAttribArray(GLKVertexAttribColor);
glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (const
GLvoid *) offsetof(Vertex, Color));
```

If you've programmed with OpenGL ES 2.0 before this will look familiar to you, but if not let me explain.

Every time before you draw, you have to tell OpenGL which vertex buffer objects you should use. So here we bind the vertex and index buffers we created earlier. Strictly, we didn't have to do this for this app (because they're already still bound from before) but usually you have to do this because in most games you use many different vertex buffer objects.

Next, we have to enable the pre-defined vertex attributes we want the GLKBaseEffect to use. We use the glEnableVertexAttribArray to enable two attributes here – one for the vertex position, and one for the vertex color. GLKit has predefined constants we need to use for these – GLKVertexAttribPosition adn GLKVertexAttribColor.

Next, we call glVertexAttribPointer to feed the correct values to these two input variables for the vertex shader.
This is a particularly important function so let's go over how it works carefully.

- The first parameter specifies the attribute name to set. We just use the predefined constants GLKit set up.
- The second parameter specifies how many values are present for each vertex. If you look back up at the Vertex struct, you'll see that for the position there are three floats (x,y,z) and for the color there are four floats (r,g,b,a).
- The third parameter specifies the type of each value – which is float for both Position and Color.
- The fourth parameter is always set to false.
- The fifth parameter is the size of the stride, which is a fancy way of saying "the size of the data structure containing the per-vertex data". So we can simply pass in sizeof(Vertex) here to get the compiler to compute it for us.
- The final parameter is the offset within the structure to find this data. We can use the handy offsetof operator to find the offset of a particular field within a structure.

So now that we're passing on the position and color data to the GLKBaseEffect, there's only one step left…

5) Draw your geometry

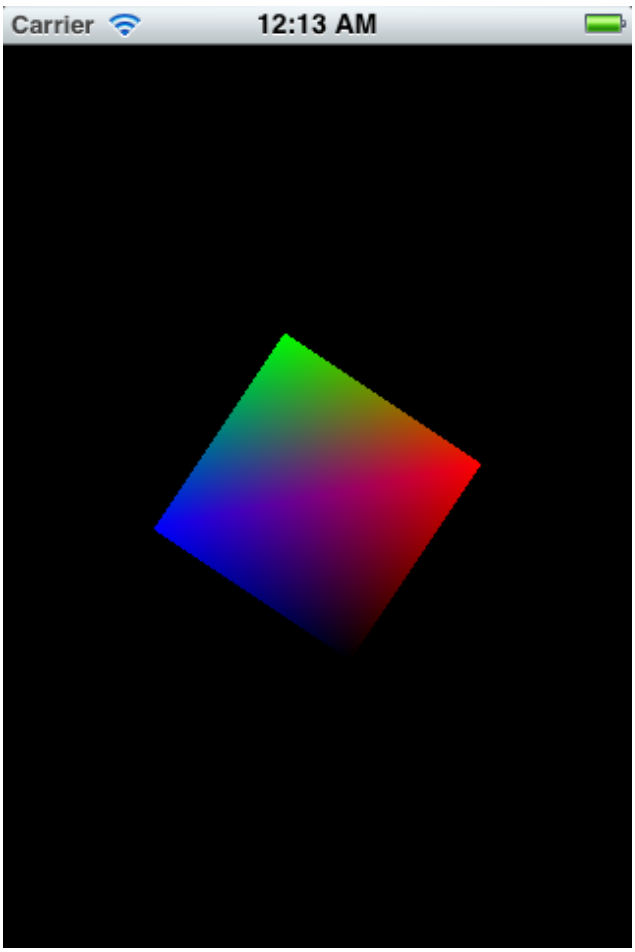
This is a very simple one-liner at this point. Add this to the bottom of glkView:drawInRect:

```
glDrawElements(GL_TRIANGLES, sizeof(Indices)/sizeof(Indices[0]), GL_UNSIGNED_BYTE, 0);
```

This is also an important function so let's discuss each parameter here as well.

- The first parameter specifies the manner of drawing the vertices. There are different options you may come across in other tutorials like GL_LINE_STRIP or GL_TRIANGLE_FAN, but GL_TRIANGLES is the most generically useful (especially when combined with VBOs) so it's what we cover here.
- The second parameter is the count of vertices to render. We use a C trick to compute the number of elements in an array here by dividing the sizeof(Indices) (which gives us the size of the array in bytes) by sizeof(Indices[0]) (which gives us the size of the first element in the arary).
- The third parameter is the data type of each individual index in the Indices array. We're using an unsigned byte for that so we specify that here.
- From the documentation, it appears that the final parameter should be a pointer to the indices. But since we're using VBOs it's a special case – it will use the indices array we already passed to OpenGL-land in the GL_ELEMENT_ARRAY_BUFFER.

Guess what – you're done! Compile and run the app and you should see a pretty rotating square on the screen!



Where To Go From Here?

Here is the example project with all of the code from the above tutorial series.

At this point, you have hands-on experience with making a simple GLKit based app with OpenGL ES 2.0 – completely from scratch!

If you're new to OpenGL ES 2.0, you've also gotten a great grounding on some of the most important techniques, such as vertex and index buffers and vertex attributes.

However, there's more cool stuff in store for you with GLKit. If you want to learn more, check out our new book iOS 5 By Tutorials, where we move to full 3D, and demonstrate some of the cool effects you can get with GLKitBaseEffect, such as lighting, fog, and more!

If you have any questions or comments on this tutorial or on OpenGL ES 2.0 or GLKit in general, please join the forum discussion below!

Ray Wenderlich

Ray is part of a great team - the raywenderlich.com team, a group of over 100 developers and editors from across the world. He and the rest of the team are passionate both about making apps and teaching others the techniques to make them.

When Ray's not programming, he's probably playing video games, role playing games, or board games.