SIMON MAURICE - IPHONE - OPENGL ES

Thursday, 18 June 2009

OpenGL ES 15 - Blender Models Part 2: Loading and Rendering

Glad to see you back!

I did rant a bit about file formats in the last entry but I should have mentioned something else. One thing which I like to do is to have one format for static models (ie the world or scene such as walls, floors, terrain etc) and another for animated objects (people, objects etc).

The reason for this is simple. You need different information for rendering a scene than you do for rendering people, moving walkways, pulse cannons etc. For the static world rendering, you do need to have some references in there for things such as start points, where movable objects exist, visible face determination etc. In movable objects, you need things like animation keyframes (if using keyframes), joint information for skeletal animation if using etc.

If you think about it, you come across the same sort of model for characters and objects in a game whereas the rooms, corridors etc are not re-used as different instantiations.

So, I do tend to think of the different types of rendering that needs to be done separately.

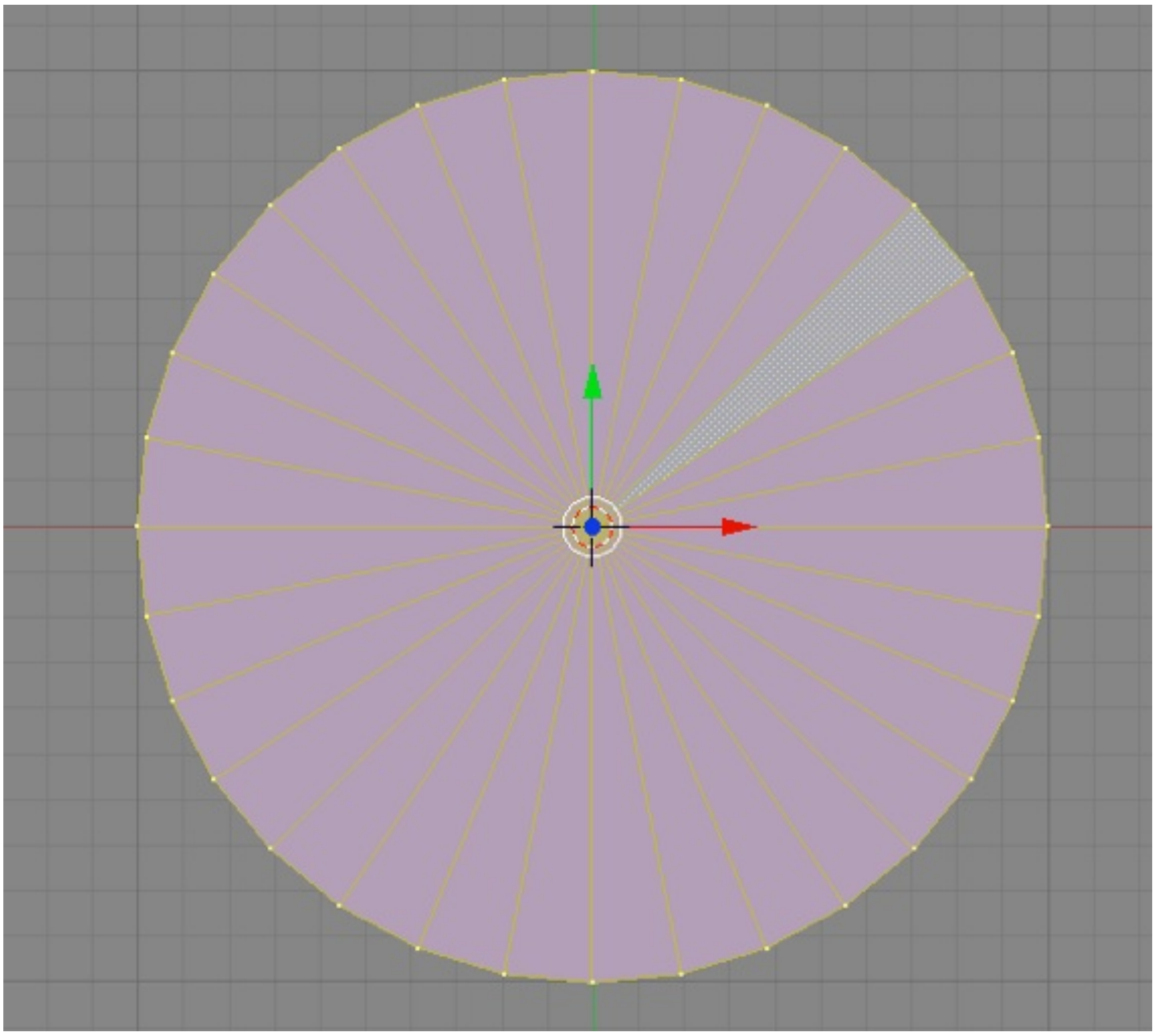
With that in mind, we're going to be looking at bringing in an object within a scene right now rather than rendering the scene itself. When I say scene, I mean rendering walls, floors, ceilings, doors etc.

Getting Back to Blender
Now, last time we went through Blender's Interactive Python Console to discover some information about what was in the current Blender scene. Now we're going to write an export script to get some data out. The simplest kind of data to export and render will be the kind like the filled circle from the last tutorial.

That is, the object will contain faces represented as triangles. That will make drawing these a breeze and a perfect example to get started with.

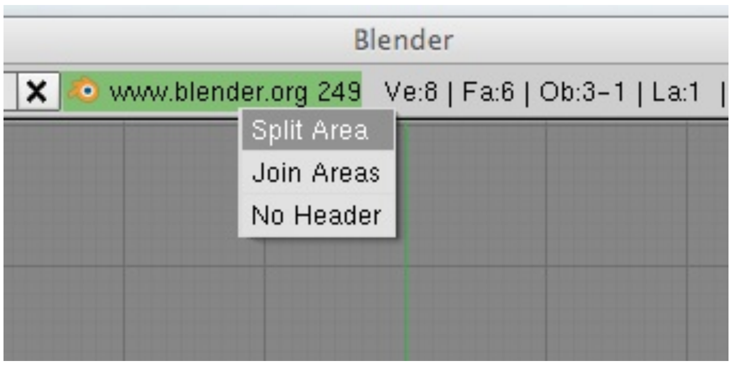
Get back into Blender (we're not ready for Xcode yet). Press **CTRL-X** to create a new scene if you didn't just launch Blender. First of all, we'll do the filled circle example from last time, write an export script for it, and load it into the iPhone. Then I'll show you that you can use this technique to render a far more complex model.

So, just quickly, delete the cube. Press **SPACE** "Add->Mesh->Circle". Make sure "Fill" is selected in the "Add Circle" window. Click OK and then press the **TAB** key to go into edit mode. You should have a circle like this:

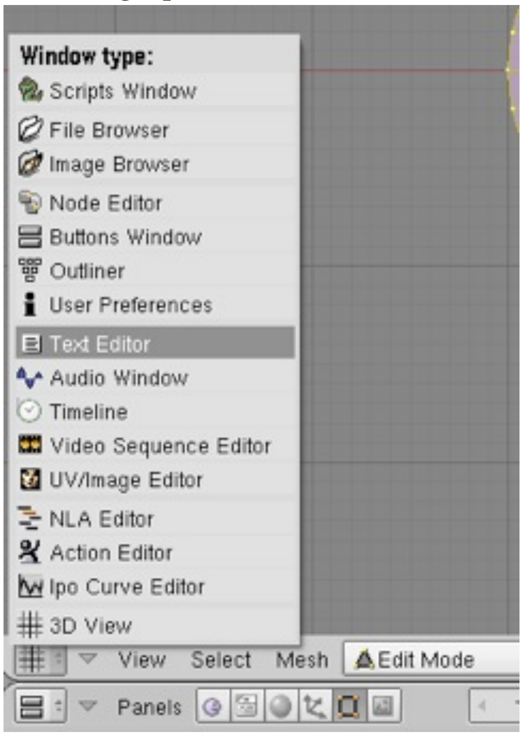


This time, instead of going into the Python Interactive Console, we're going to use the text editor features of Blender because we're creating a script, not working live.

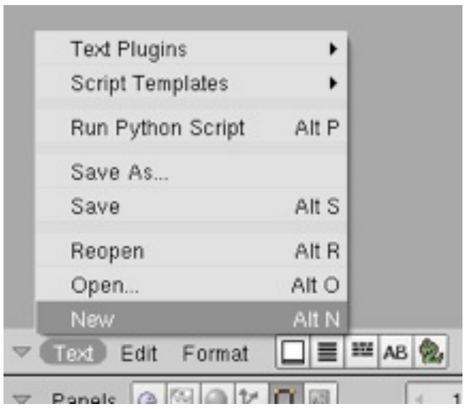
So, go to the top of the window and right click and select "Split area" to split the view in half.



In the left half, click on the hash looking icon at the lower left of the object view and bring up the Window menu. Select "Text Editor" like so:



Again, like opening the Python window, we get a grey screen. We need to create a new text item to get something to edit. Press **Option-N** or click on the Text menu and select new like so:



The scroll bar and the red cursor at the top of the screen tells us that we have a window.

Writing a Blender Script
Just as we were doing in the Interactive Python Console, Blender scripts are written in Python. I suppose, if you needed advanced functionality, you could write C functions and call that from Python but for what we're doing, we're going to stick with Python.

You don't have to edit your Python scripts in Blender. I used my editor of choice, Emacs, for my scripts but you use what you like (yeah I know, I'm an old fart so of course I use Emacs :). Just remember that you can't have hard tabs in a Python script, they must be expanded to spaces otherwise Python will spit the dummy.

As I mentioned before, we're going to write a binary format. Don't panic and don't fear binary files. They are easy to work with and far more efficient than text fields. I think most people who shy away from binary files do so because they either aren't confident, or have tried writing importers of various kinds before and it all went awry.

You won't have that problem because we're going to do it the easy and simple way so there's no chance for you to get lost. If you do get lost, get a good Hex editor (I recommend HexFiend) to help you out.

Here's the full script. Explanation is below. You can type it in if you like, just remember to convert the tabs to spaces in the Blender text editor before you execute it (Format menu -> Convert whitepace -> To Spaces"). Also, this script is included in the project download below.

```
#!BPY

import struct
import bpy
import Blender

def newFileName(ext):
    return '.'.join(Blender.Get('filename').split('.')[:-1] + [ext])

def saveAllMeshes(filename):
    for object in Blender.Object.Get():
        if object.getType() == 'Mesh':
            mesh = object.getData()
            if (len(mesh.verts) > 0):
                saveMesh(filename, mesh)
                return

def saveMesh(filename, mesh):
    # Create the file and write the header
    file = open(filename, "w")
    file.write(struct.pack("<I", len(mesh.verts)))
    file.write(struct.pack("<H", len(mesh.faces)))

    # Write an interleaved vertex and normal array
    for vertex in mesh.verts:
        file.write(struct.pack("<fff", *vertex.co))
        file.write(struct.pack("<fff", *vertex.no))

    # Write the index for each triangle coordinates
    for face in mesh.faces:
        assert len(face.v) == 3
        for vertex in face.v:
            file.write(struct.pack("<H", vertex.index))

Blender.Window.FileSelector(saveAllMeshes, "Export for iPhone",
newFileName("gldata"))
```

The Script Dissected
First of all, I need to acknowledge Micah Dowty who wrote the original script which I studied to give me a leg up into Blender scripting. This is just a variation on his script. That's basically how I got started in scripting: studying other people's script to see how Blender worked, then went to the Python Interactive Console to test what I was learning.

First of all, is a comment line: `#!BPY`. This tells Blender that this is a script it should use, that's all. It's best to make this the first line in the script.

Then we import 3 modules which we use in the script. The first one is a standard Python module (`struct`) which allows us to pack text into a binary format, `bpy` & `Blender` obviously relate to Blender.

The real code starts with three different functions. They are specified by the lines starting with `def`. They are:

• **newFileName**: this just gets the filename which the user selects and appends the extension to it (in this case gldata).
• **saveAllMeshes**: what happens here is that we loop through all objects returned by `Blender.Object.Get()`, and we check to ensure the object is of type Mesh. The other types of objects that you will get back include cameras, lightsources etc which we do not render in OpenGL. In other occasions we may use this in static scene data but not right now.
• **saveMesh**: this function actually writes the file in the format I've put it in.

The File Format
The file format is very simple. There is a header which contains an integer and an unsigned short. These hold the mesh's vertex count and the face count.

Then, I store the vertex and normal information in an interleaved array as floats so it will look like:

```
1.00000  -1.0000 0.0000          <--- Vertex
1.00000  1.0000  1.0000          <--- Normal
-1.00000 -1.0000 0.0000          <--- Vertex
1.00000  1.0000  1.0000          <--- Normal
```

And so on.

Then I write the index for each triangle's point. This will be explained later as I've just realised I've never covered `glDrawElements()`, we've only covered `glDrawArrays()`. However I have covered interleaved arrays in the tutorial "Points and Lines in a Stride".

The vertex index array is an array of unsigned shorts.

Now, the thing to point out is that I have forced the data into little endian format. The reason is quite simply, that the iPhone's CPU works as a little endian processor just like the Intel model in our desktops. You may recall the that PowerPC (like the Motorola 68000 series was big endian). The only reason that I've forced the data into little endian is just in case you've got Blender installed on a box which has a PowerPC (or UltraSPARC etc).

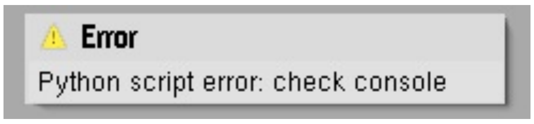
I've forced into little endian with the "<" at the beginning of the format strings in `struct.pack()`.

I really should explain the export script a bit more but right now, this is in danger of becoming more of a Blender tutorial rather than an OpenGL ES / iPhone tutorial so I'm going to move right along. At some point, I'll put more information on export scripting probably somewhere else on my website.

Saving the Mesh
So, either type the script in, copy and paste it, or just load it from the project archive (below). Once it's in the text window, we can test it out. To test it, press **Option-P** (I probably should say "Alt" instead of Option as that's how Blender refers to it but hey, we're on Macs, we should celebrate it!! :)

This will execute the script.

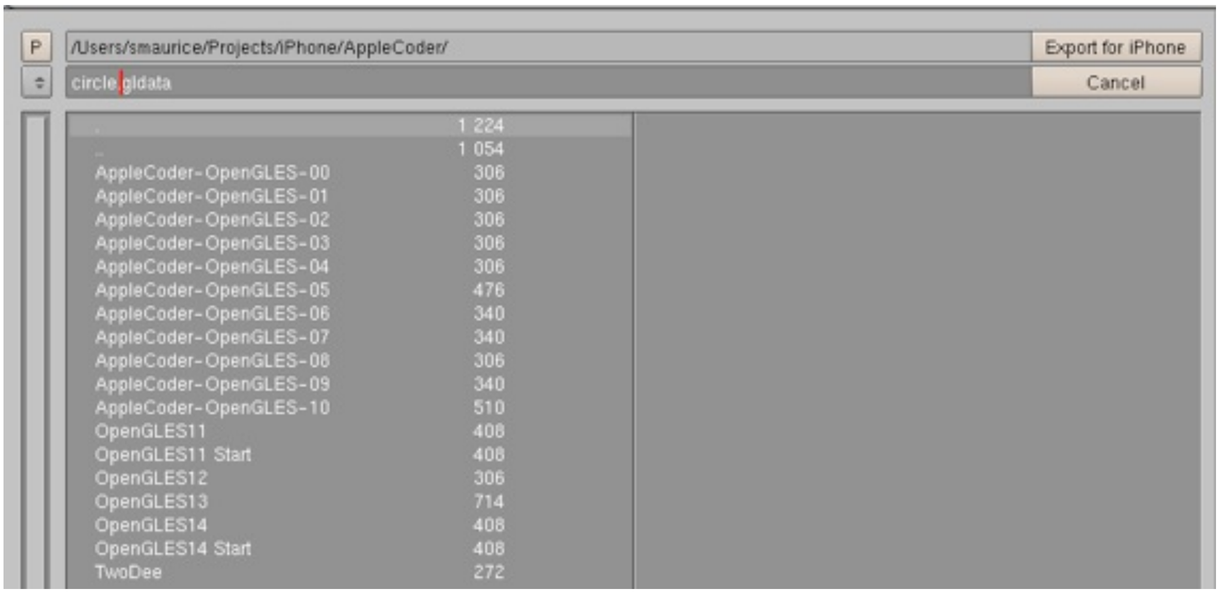
If you get an error like this:


Error
Python script error: check console

Something's gone awry. Check the console like it says (Applications/Utilities/Console). The icon looks like this:

Console

If all goes well, you should get this:



```
P   /Users/smaurice/Projects/iPhone/AppleCoder/      Export for iPhone
    circle.gldata                                     Cancel

.                          1 224
..                         1 054
AppleCoder-OpenGLES-00       306
AppleCoder-OpenGLES-01       306
AppleCoder-OpenGLES-02       306
AppleCoder-OpenGLES-03       306
AppleCoder-OpenGLES-04       306
AppleCoder-OpenGLES-05       476
AppleCoder-OpenGLES-06       340
AppleCoder-OpenGLES-07       340
AppleCoder-OpenGLES-08       306
AppleCoder-OpenGLES-09       340
AppleCoder-OpenGLES-10       510
OpenGLES11                   408
OpenGLES11 Start             408
OpenGLES12                   306
OpenGLES13                   714
OpenGLES14                   408
OpenGLES14 Start             408
TwoDee                       272
```

Enter a filename (I've called it circle.gldata) and click "Export for iPhone" to export the file into your chosen location.

If no error occurs, then you'll just be returned to the script window. One of the shortcomings of such a simple script is a lack of user feedback!! :)

Loading Our Object Into the iPhone

Now that we have our file exported, let's head off to Xcode. I've created a starter point for our object loading with this project here. This includes the Blender export script (`export_gldata.py`): OpenGLES15 Start.zip

Load up the starter project and the way we're going to handle this file is to create a new Objective-C class. Right click on "Classes" over in the "Groups & Files" section and choose "Add -> New file". Create a subclass of NSObject and I've just called it "BlenderObject". Be sure to create the header file as well.

Now, we're going to start in the header file. Import the following files:

```
#import <OpenGLES/EAGL.h>
#import <OpenGLES/ES1/gl.h>
#import <OpenGLES/ES1/glext.h>
```

We basically need to hold four pieces of information here:

- Triangle count
- Vertex count

- Vertex & normal data (the actual array)
- Vertex index array

So, in the interface, add the following variables:

```
int vertexCount;
unsigned short triangleCount;// Equivalent to len(mesh.faces)
GLfloat *vertexArray;
GLushort *indexArray;
```

Now, we need to add some methods with which we can load and draw the model.

```
- (NSError *)loadBlenderObject:(NSString *)fileName;
- (void)draw;
```

OK, we're done in the header. Hit the implementation file and we'll start to implement these two methods.

Loading the File

The first thing we need to do is open an `NSFileHandle` for reading the data. We pass the filename in without the extension as `pathForResource:` will sort out the full file path. If something goes wrong, we just return an `NSError` object.

```
    NSString *filePath = [[NSBundle mainBundle] pathForResource:fileName
                                                         ofType:@"gldata"];
    NSFileHandle *handle = [NSFileHandle
                                  fileHandleForReadingAtPath:filePath];
    if (handle == nil) {
        NSString *msg = @"Something went really, really wrong...";
        return [NSError errorWithDomain:@"BlenderObject"
                                   code:0
                               userInfo:[NSDictionary
                                    dictionaryWithObject:msg
                                  forKey:NSLocalizedDescriptionKey]];
    }
```

Next we read our simple file header. Remember, it's just an int and and unsigned short.

```
[[handle readDataOfLength:sizeof(int)] getBytes:&vertexCount];
[[handle readDataOfLength:sizeof(unsigned short)] getBytes:&triangleCount];
```

With our header read in, we now know how many vertices there are, and how many triangles or faces we have to render. So we can go right ahead and allocate space for the data.

```
vertexArray = malloc(sizeof(GLfloat) * 6 * vertexCount);
indexArray = malloc(sizeof(unsigned short) * triangleCount * 3);
[[handle readDataOfLength:sizeof(GLfloat)*6*vertexCount] getBytes:vertexArray];
[[handle readDataOfLength:sizeof(unsigned short) * triangleCount * 3]
getBytes:indexArray];
```

We've allocated the space for the vertex array. They are the size of `GLfloat` (same as a float), but we need 6 floats for each vertex (3 x co-ordinates, 3 x normal).

The index array is the same deal except we need 3 x unsigned shorts for each triangle (a triangle being made up of 3 points...).

Then we just read the data into our two memory spaces for each array.

Finally, just a: `return nil;`

That's it! That's all we need to do. How easy is that? Don't start thinking "but it's only a circle" because below I'll show you a complicated object that will put your doubts to rest.

Drawing that Bad Boy
Now we need to implement the draw: method. Note that from the last tutorial I showed you the individual vertices for each face and we could have just exported that and pumped it into `glDrawArrays()` to render that bad boy. However this is going to be a step forward in rendering efficiency.

One thing which I haven't covered yet is `glDrawElements()`. So far we've used a simple array of vertices for drawing our objects being something along the lines of:

```
    GLfloat triangleVertices[] = {
        // Triangle 1
       -5.0, 1.0, 0.0,      // Top Centre
       -6.0, -1.0, 0.0,     // Bottom left
       -4.0, -1.0, 0.0,     // Botom Right

        // Triangle 2
        -3.0, 1.0, 0.0,
        -4.0, -1.0, 0.0,
        -2.0, -1.0, 0.0
    }

    glVertexPointer(3, GL_FLAT, 0, triangleVertices);
    glDrawArrays(GL_TRIANGLES, 0, 6);
```

This obviously gives us 2 triangles.

However, there is a more efficient way both in speed and memory usage. Hence `glDrawElements()`. Can you see that the 3rd vertex of the first triangle and the 2nd vertex of the second triangle are the same co-ordinate? We can avoid having a duplicate vertex which, while in this case saves us a paltry 4 bytes, the speed increase when executed over 100's and 1000's of triangles is a leap.

What happens with `glDrawElements()` is that we only specify unique vertices and then give an array of indexes which tells OpenGL which vertex to use for each "point" of the triangle. Have a look at this:

```
   GLfloat triangleVertices[] = {
       -5.0, 1.0, 0.0,      // Top Centre
       -6.0, -1.0, 0.0,     // Bottom left
       -4.0, -1.0, 0.0,     // Botom Right

        -3.0, 1.0, 0.0,
        -2.0, -1.0, 0.0
    }
    GLushort index[] = { 0, 1, 2, 3, 2, 4 };
```

In the `triangleVertices[]` array, I've just deleted the middle vertex from the second triangle as it was a duplicate. Then, I've created an array of vertex indexes which, if you follow along, you can see the sequence.

It tells OpenGL to use vertices 0, 1, & 2 for the first triangle, then use 3, 2, & 4 for the second triangle. It's that easy.

Working out the data for `glDrawElements()` is a pain if your vertex array is specified by hand, but if you've got something like Blender on the back end generating your data, you can do it this way.

So let's have a look at the drawing code:

```
- (void)draw {
glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, sizeof(GLfloat)*6, vertexArray);
glNormalPointer(GL_FLOAT, sizeof(GLfloat)*6, &vertexArray[3]);
    glDrawElements(GL_TRIANGLES, triangleCount*3, GL_UNSIGNED_SHORT, indexArray);
    glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
}
```

Nothing special there apart from the changes to the call from `glDrawArrays()` to `glDrawElements()`. The parameters for `glDrawElements()` are really simple. They are:

```
void glDrawElements(
                  GLenum mode,
                  GLsizei count,
                  GLenum type,
                  const GLvoid * indices);
```

•`mode` is just `GL_TRIANGLES` in our case. We can feed any primitive into `glDrawElements()` as we would into `glDrawArrays()`.
•`count` is the number of vertices. Is the count of the Blender faces which in our cases are triangles.
•`type` is the kind of data ***in our index array***. In OpenGL ES you can only use `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT`. I've just chosen the latter for flexibility in the exporter.
•`indicies` is the pointer to the array of indexes for the vertices. This is an array of type from the previous parameter.

Oh, one more thing: I've used the stride value this time in `glVertexPointer()`. If you remember from Points and Lines in a Stride tutorial, you'll remember this is the size of our data structure which contains the normals as well as the co-ordinates.

That's it! We're done!

Bringing the Project Together
Head over to `EAGLView.h`. This is where we set up a variable in our implementation file

to hold our Blender object. First, import the `BlenderObject` header file:

```
#import "BlenderObject.h"
```

In the class interface, add the following variable:

```
BlenderObject *circle;
```

Now, over to `EAGLView.m` and straight into the `initWithCoder:` method, we can create the `BlenderObject` and load it from the app's `mainBundle`:

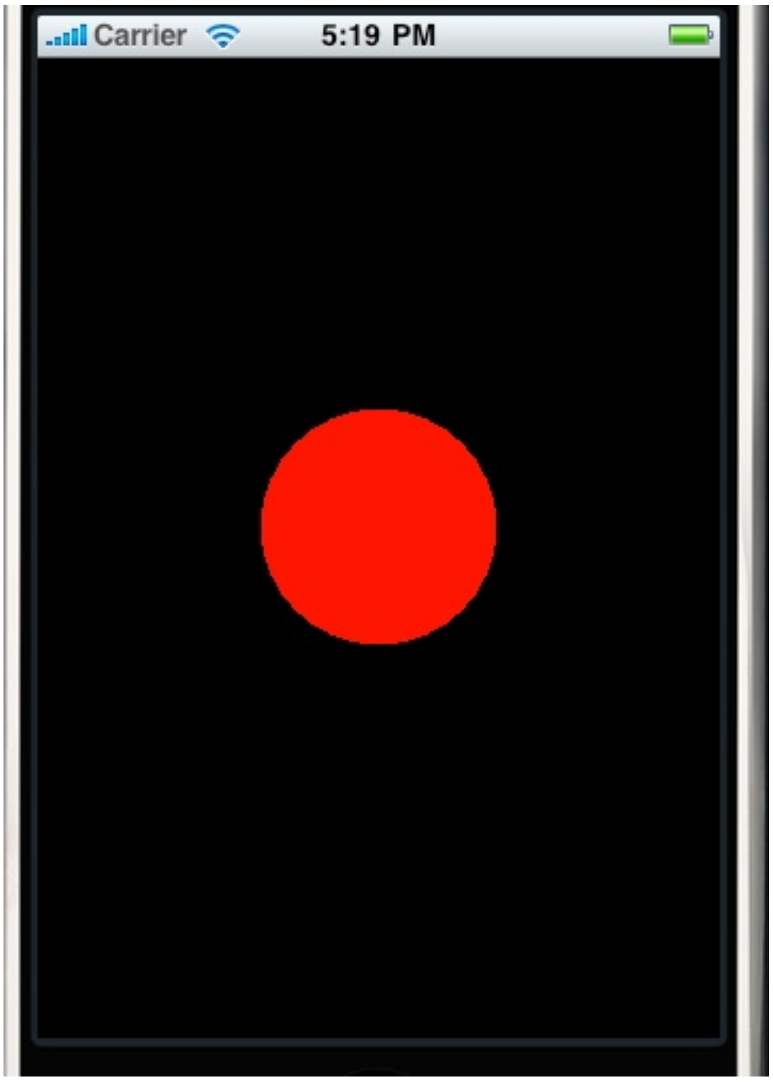
```
    circle = [[BlenderObject alloc] init];
    [circle loadBlenderObject:@"circle"];
```

Once again, note the lack of file extension for the file name. `pathForResource:` will sort this out for us.

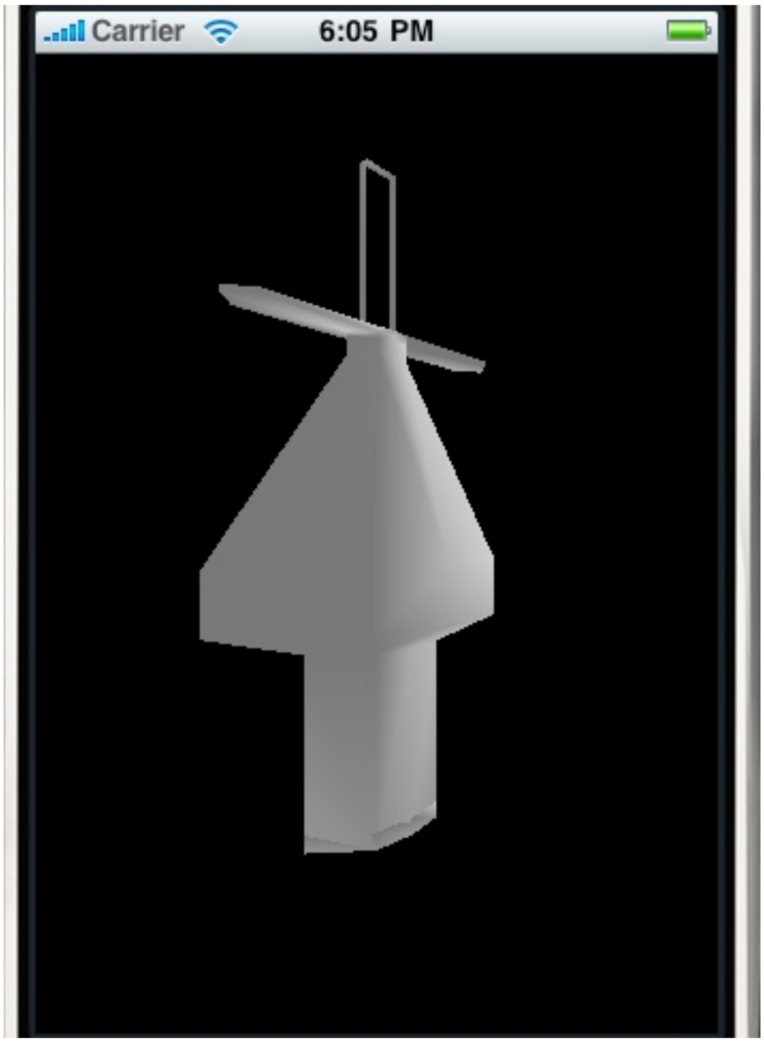
Lastly, in `drawView:` add the following code:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Because I forgot in the
starter project
glPushMatrix();
glTranslatef(0.0, 0.0, -5.0);
glColor4f(1.0, 0.0, 0.0, 1.0);
[circle draw];
glPopMatrix();
```

We need to translate our viewing position backwards (hence -ve) to be able to see the circle because it is currently at a depth of 0. All being well, you should get this:



OK, So That's Not So Exciting...
Like I said earlier, the circle thingy above is just an example to show you the code without worrying too much about creating a Blender model. Just to show you that this can scale up to some larger models quite happily, here's part of the space station which I'm working on rendered with a bit of lighting to show some detail:



That model has 230 triangles in it and is included as a Blender file in the full project download below.

Whilst in some ways we have gone backwards from having texture mapping and now we're back to just plain colours, that's easily fixed by getting a grip onto Blender's materials which, surprise, surprise, is coming in the next instalment.

It took be around 3 hours to get to scripting to this stage but fortunately for you, you've got it a bit easier because I've done the research for you. With around 20 lines of code, we loaded a model and normals. Loading the texture co-ordinates will take only a couple more lines that I'm sure you can work out. Just by learning a little bit about Blender's internals, we've actually saved ourselves a lot of work handling 3ds, OBJ or whatever file format.

We still need to deal with quads which are the kind of meshes Blender (and other 3D packages) create by default, but we handle that on export, not on import. You'll find if you add a plane or a cube, there are 4 vertices per face which will break the export script (at the assert line).

That's it for another instalment
OK, that's another lengthy one when I said I would shorten these up and post more regularly. I will probably deal less with Blender in the next one and probably just serve

up the additions to handle quads and textures and cover it quickly for those who are interested. In either case, there'll be little addition to the model loading code.

I know I've skipped over some things. I'll re-wrap them next time (or the time after, or the time after that...) Also, I know I left out one or two calls from the starter project, don't know if I forgot anything else. The full project works and loads that more complex object. Also there's some NSLog calls commented out which you can un-comment to see the data as imported.

Here's the code and models for this tutorial: OpenGLES15.zip

Until then, take care. Hooroo!
Simon Maurice

previous next