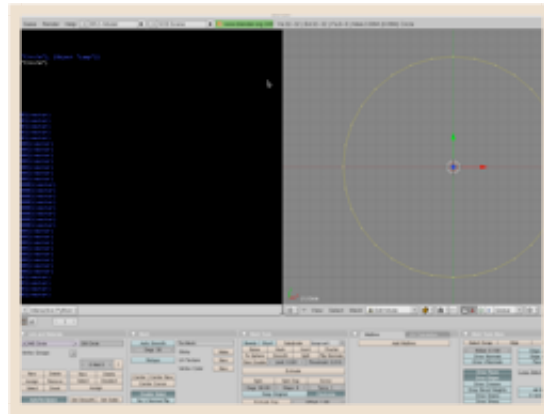


# SIMON MAURICE - IPHONE - OPENGL ES

[WELCOME](#)[IPHONE OPENGL](#)[FAQ](#)[ABOUT ME](#)[DRACONIA](#)

Ads by Google

[3DS Objects](#)[3D Tracking](#)[Free 3D Models](#)[Collada](#)

Ads by Google

[Interactive](#)[Brick Textures](#)[Animation](#)[Brazil Export](#)

Wednesday, 17 June 2009



## OpenGL ES 14 - Blender Models Part 1: Learning Some Blender Internals

G'day everyone. Been a while since I last posted. I've been knee deep in Blender and busy writing my own game at the moment so that's the reason for the delay.

In light of how much time I have at the moment, I'm going to be making a minor change to the way I'm posting here. Rather than doing lengthy individual tutorials, I'm going to increase this to a shorter, more frequent format. Essentially to "chunk things down" as IT consultants love to say. I'll get back to posting a few times a week this way.

Anyway, like I've said, I've been having a go at Blender. Normally, I'm a LightWave user so I've had to go and learn Blender which is something which I just simply was avoiding. In the past two weeks, I've got control of the user interface as well as the scripting side of the application to export models in formats which are suitable for different projects.

Using Blender isn't what these tutorials are about. Using the models (or objects) and scenes you create in Blender *is* what this is about. I'm not going to teach you Blender, there are a whole plethora of resources on

the web for that, but I will show you the things which I have learnt in getting what you create in Blender to work for you.

Whilst using Blender is quite well documented (despite the different nuances of the different versions), documentation for the scripting side of things is a bit on thin ground. So I will go into some detail on the exporting from Blender rather than just serve up a “black box” for you to use.

I won't be presenting exporting to OBJ, 3ds, COLLADA etc formats. Not only are there plenty of resources on the internet already for working with these objects, I find them a bit too heavy-weight for the iPhone. Whilst these are not too bad on the desktop with those powerful multi-core thingys that we all have these days, loading up the iPhone's processor with unnecessary work only makes your app/game/whatever slower to load, slower to work/play, & limits you more on scene/world size and so on.

All of that leads to a less than ideal user experience.

If you want to go down the LightWave/3ds/COLLADA/whatever format anyway. That's fine. There's be enough in this Blender series for you to be able to work with those files as well. I do have working COLLADA, LightWave, and 3ds loaders for the desktop which I have written (hence no copyright problems with providing it here), once I've ported them to the iPhone they will be presented in due course.

What I will be showing you first is how to get models or objects out of Blender whilst writing as little code as possible both on the Blender export side, and the loading side on the iPhone. Less code = less work. Less testing. Less debugging. Less problems.

I know some of you reading this right now will be thinking: “what, another file format? Aren't there enough already?”. Well, here's the thing. File formats for modelling and rendering software (Blender, LightWave, Maya, 3ds etc) are designed to hold far, far more information than you would use. Further, these file formats are optimised for those individual applications and care nothing for *your* application.

Handling those files at run time really loads up the processor as we need to not only read the data in from those files, we need to handle many special case situations in order to get these into a format suitable for rendering with OpenGL ES on the iPhone.

Do that pre-processing on the desktop, not on the iPhone. Leverage the multi-core thingy's power advantage over the iPhone's CPU. Make the data ready to go once loaded and you'll write less code.

### **A Quick Word on Choosing File Formats**

Go to any Blender or similar forum and check out answers to questions such as “how to get Blender objects into OpenGL”. The answers are varied depending on what the responder’s preferences are. Some say 3ds. Some say OBJ. Others say they wrote an export script to a C header file. Then someone will say “OBJ is only easy to use but useless for real world work as doesn’t support this or that”. Someone else will then respond saying “all those binary file formats are nasty! Work with non-proprietary text files”.

No wonder there’s such a state of confusion.

Ultimately, I think the least accepted answer is the most correct answer. Roll your own exporter. Or, at least, use one that you can just load the data with no or limited app side processing.

Look, I’m not going to tell you which file format to use. I’m just trying to show you there’s another way to using someone else’s file format. If you need your own file format, then you’ll love this. If you use a heavyweight file format because you want the flexibility of app side editing, then you’ll at least get an understanding of what those Blender exporters do. You may need to know the detail as importers and exporters are not perfect.

What I present here is just one man’s point of view, just like someone responding on a forum recommending someone to use LightWave’s file format. I’m not saying this is the only way, but at least you can have enough information to make up your own mind with all the information.

Anyway, enough of a rant, let’s get on with it.

### **Blender Models and How to Extract Them**

Blender’s a damn fine tool. Coming from Lightwave (and Quicksilver & Imagine before that), I’ve found it a little quirky at first, but, once I got the hang of it, it’s a really efficient modeller. I’d pay money for it. Fortunately we don’t have to because of the dedicated work of lots of individuals in the Blender community. It actually is an Open Source project that I wouldn’t mind contributing to (and may do).

Our primary use of a 3D software package is for modelling rather than rendering. Blender can give LightWave a run for it’s money in many regards as far as modelling goes. So much so that ultimately, I think with perseverance, anything I can do in LightWave, I can do in Blender. It will just be a different process and take more steps.

I should point out that whilst I do have much experience with

LightWave, I'm no artist. I'm sure I only use a small amount of it's power.

Like all good 3D programs, Blender comes with an interface to work with programmatically (not just the graphical environment). This allows us access to the "internals" of Blender which means we can control the scene, adjust models and animations, and of course, export them. Thankfully, the Blender developers chose a common scripting language in the form of Python rather than roll their own custom scripting language (eg LightWave uses LScript, or it's C API can also be used). On less thing to learn!

### **Before we start, there's one more thing...**

When it comes to working with Blender, I've decided to take a "from step one" approach. I'm going to show it to you the way that I learned to do what I now can do with object exporting. There are probably things which I've learnt along the way which I would normally skip over covering in this blog, resulting in you needing to fill the gaps yourself.

Whilst this may be a little annoying to some who are a little further along in Blender, I think it's beneficial because it can show people who are wanting to write their own scripts or to roll their own custom formats, just how things work.

### **Getting Started**

OK, here we go at last (so much for shorter blogs). First, I'm using Blender 2.49 which, at the time of typing, is the latest version. You will need to have a basic understanding of how Blender works. For that, I read the Wikibook "[Blender 3D: From Noob to Pro](#)" which is an excellent resource. If you haven't looked at this yet, don't walk, *run* to check this out.

Wikibooks are really good for actively developed Open Source projects. They are far better than printed books because they're not etched in stone. As Blender changes, the books are updated to keep pace with development. If I needed to pay for access to such a book, I would have no problems doing so.

So, get Blender, learn a few basics of it, then you're ready for what comes next.

The next thing which would help is a little understanding of the scripting language Python. About 10 years ago, I bought the O'Reilly book *Mastering Python* (I think that was the title) to learn the language. Today, there are plenty of resources at the [Python](#) website to learn.

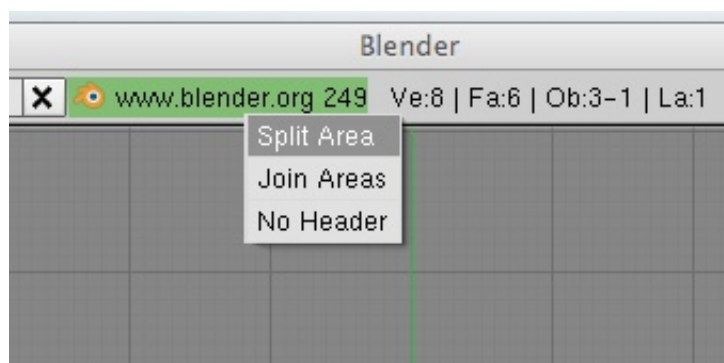
I haven't used Python as I prefer Perl for scripting but that's just me;

I'm not much of a fan of object oriented programming (don't get me started on C++, that language is pure evil). Despite not having used Python over the years, it is easy to learn and use. If you don't know the language, I think you'll still find the explanations below quite self explanatory, especially if you've ever used an ALGOL based language like Pascal, Modula-2 or Modula-3 (now there's a couple of really fine languages!).


### Discovering the Python Interactive Console

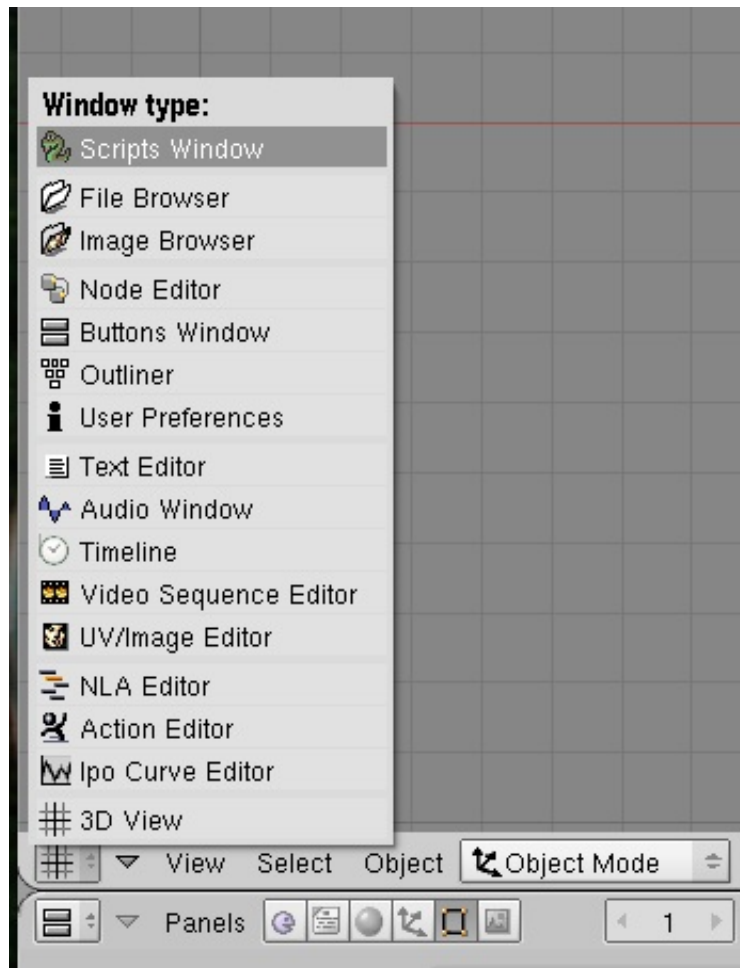
When you launch Blender, you're presented with a cube which, in most cases, you delete straight away. In this case, leave it there.

Right click up the top of the screen like the image below and click split area.

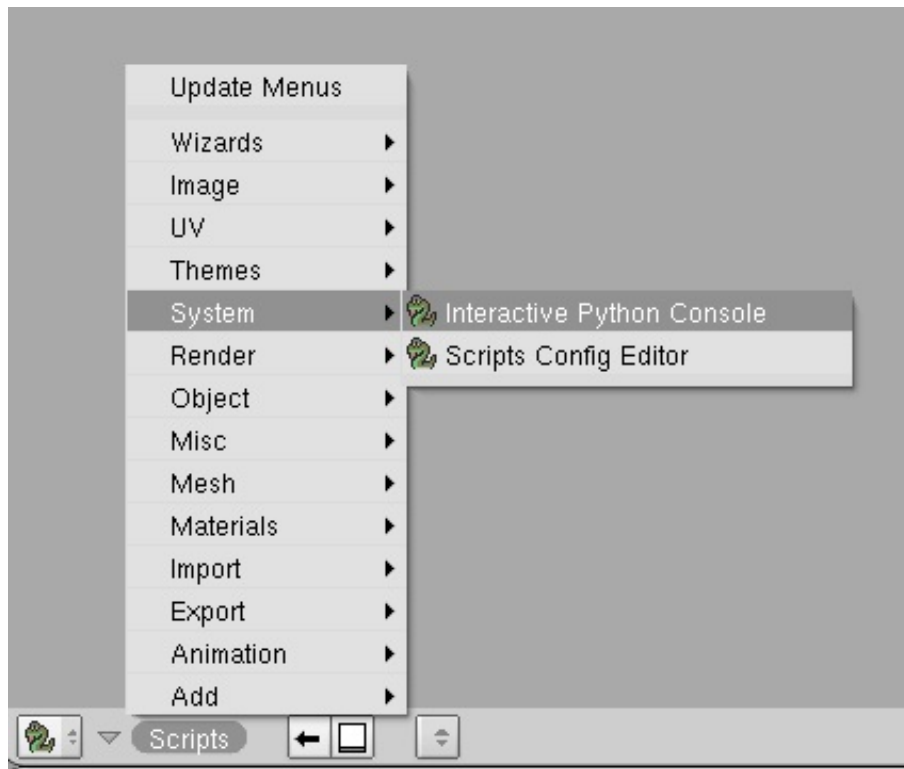


That should split the screen in to two and give you the cube in both.

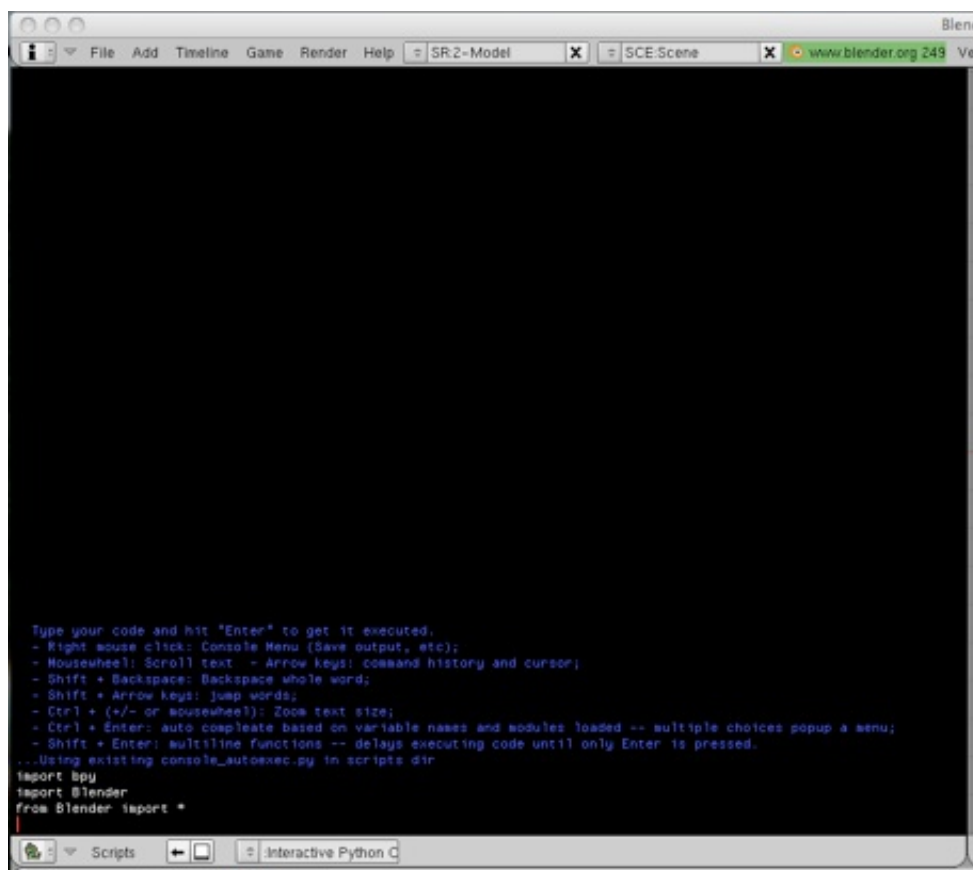
Over to the left half of the screen, that is where I normally set my scripts window, so click on the little hash icon  at the top of the window thingy at the bottom and choose: "Scripts Window".



This gives us a blank grey view in this half of the Blender window. Finally, create a new interactive console by clicking on the “Scripts” menu beside the green snake icon, and selecting “System -> Interactive Python Console”.



Righto, now you should have an console which will look something like this:



This window is where you can type in Python commands directly to interact with Blender immediately. This is different to a script only in that commands are executed one by one rather than in a single run like a script file.



Right now, you can see that there are a few “import” statements. This is just like Pascal or Modula-2 where you imported units or modules to bring functionality (such as `PrintLn` in Pascal’s IO unit). In this case, the console has imported Blender specific modules. This is one good thing about Python, you use less imports to get basic things done than Pascal or Modula-2.

### Getting Information About Our Current Scene

First thing to type in is: `print Blender.Object.Get()` and press return. You should get this:

```
print Blender.Object.Get()
[[Object "Camera"], [Object "Cube"], [Object "Lamp"]]
```

**Note:** One thing I should have pointed out, Python is case sensitive so “`blender.object.get()`” will not work.

What we’ve done is asked Blender to give us a print out of all objects in our current scene. You can see the object heirachy: most objects that you work with will be accessed via the Blender module. The object we’re interested in is the one named “Cube”, so grab a reference to it by entering:

```
cube = Blender.Object.Get("Cube")
```

So, already you can see that if we pass nothing to the `Get()` call, we get all objects. If we name an object, we get that object.

The variable `cube` now points to the cube in the centre of the right hand side window. Let’s start to discover some information about how Blender organises it’s objects.

An object that we would take an render in OpenGL is made up from *Mesh* data (not strictly true I know, but will suffice for now). So, the first thing we need to do with any object that we grab a hold of is to actually get the mesh data. This is done with the following command:

```
mesh = cube.getData()
```

Again, this command will not print out any response. The variable “`mesh`” now holds all the vertex, normal etc information about the cube.

Some, not all, objects are made up of *Faces*. To see if an object has any faces, you can enter:

```
print len(mesh.faces)
```



```
mesh = cube.getData()
print len(mesh.faces)
6
```

I don't think I need to point out that a cube does have 6 faces...

So we have a cube, and we know that it's made up of 6 faces. So, for each face, there should be a set of vertex co-ordinates. Shouldn't there? How else can the size and shape of any object be known? It's these co-ordinates that we are interested in, so let's find out something about them.

Here's going to be the first example of entering a loop in the interactive console. Type in the following:

```
for face in mesh.faces:
    print len(face.v)
```

Note that you will need to press return twice to get the for loop to execute. Also note that after you pressed return after the first line, the console immediately indented the next line. This is because Python uses white space indentation instead of curly braces to signify blocks of code.

The code above just says for each face in the object's array of faces, held in memory at mesh.faces, print out the quantity of vertices. Obviously, this is just like Objective-C's properties where v is the name of the variable for vertices.

You should get the following:

```
for face in mesh.faces:
    print len(face.v)
4
4
4
4
4
4
```

This tells us that the cube's faces are made up of quads. There are 4 vertices to each face. Therefore to draw this cube in OpenGL ES, we would need to use either GL\_TRIANGLE\_STRIP or GL\_TRIANGLE\_FAN with the four co-ordinates.

Let's drill down a bit deeper into the cube object and discover information about each face. Type in the following:

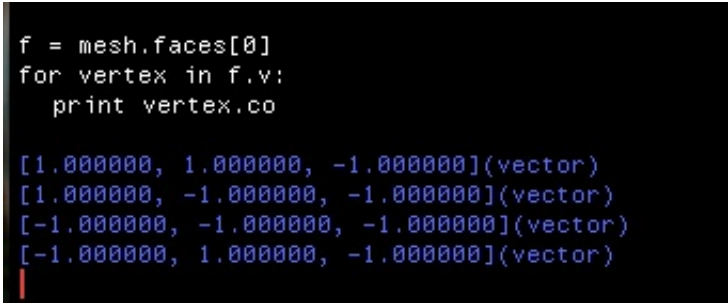
```
f = mesh.faces[0]
```

We've just created a new variable `f` which refers to the faces array, object 0. Just like an Objective-C property once again. Press return and there is once again no output (unless you've made an error).

Now type:

```
for vertex in f.v:  
    print vertex.co
```

Press return twice (once to get a new line, once again to execute the code) and you should get this:



```
f = mesh.faces[0]  
for vertex in f.v:  
    print vertex.co  
  
[1.000000, 1.000000, -1.000000](vector)  
[1.000000, -1.000000, -1.000000](vector)  
[-1.000000, -1.000000, -1.000000](vector)  
[-1.000000, 1.000000, -1.000000](vector)
```

The output is simply the co-ordinates for each vertex in this cube's face. That's the co-ordinates we need to draw this face. In other words, this is the vertex array which, so far, we have entered by hand in Xcode.

To get all faces' vertices in one hit, we could type:

```
for face in mesh.faces:  
    for vertex in face.v:  
        print vertex.co
```

And you'll get this:

```

for face in mesh.faces:
    for vertex in face.v:
        print vertex.co

[1.000000, 1.000000, -1.000000](vector)
[1.000000, -1.000000, -1.000000](vector)
[-1.000000, -1.000000, -1.000000](vector)
[-1.000000, 1.000000, -1.000000](vector)
[1.000000, 0.999999, 1.000000](vector)
[-1.000000, 1.000000, 1.000000](vector)
[-1.000000, -1.000000, 1.000000](vector)
[0.999999, -1.000001, 1.000000](vector)
[1.000000, 1.000000, -1.000000](vector)
[1.000000, 0.999999, 1.000000](vector)
[0.999999, -1.000001, 1.000000](vector)
[1.000000, -1.000000, -1.000000](vector)
[1.000000, -1.000000, -1.000000](vector)
[0.999999, -1.000001, 1.000000](vector)
[-1.000000, -1.000000, 1.000000](vector)
[-1.000000, -1.000000, -1.000000](vector)
[-1.000000, -1.000000, -1.000000](vector)
[-1.000000, -1.000000, 1.000000](vector)
[-1.000000, 1.000000, 1.000000](vector)
[-1.000000, 1.000000, -1.000000](vector)
[1.000000, 0.999999, 1.000000](vector)
[1.000000, 1.000000, -1.000000](vector)
[-1.000000, 1.000000, -1.000000](vector)
[-1.000000, 1.000000, 1.000000](vector)

```

Vertex information isn't all that's stored. We also have normals, UV coordinates. For example, to get the normals, instead of saying:

```
print vertex.co
```

type

```
print vertex.no
```

This will give you the vertex normal for each vertex.

### Not Everything is Quads

Blender, by default, will organise the object into quads because this is what 3D artists like. On the other hand, we programmers prefer triangles because that's what graphics hardware likes to use in realtime 3D rendering.

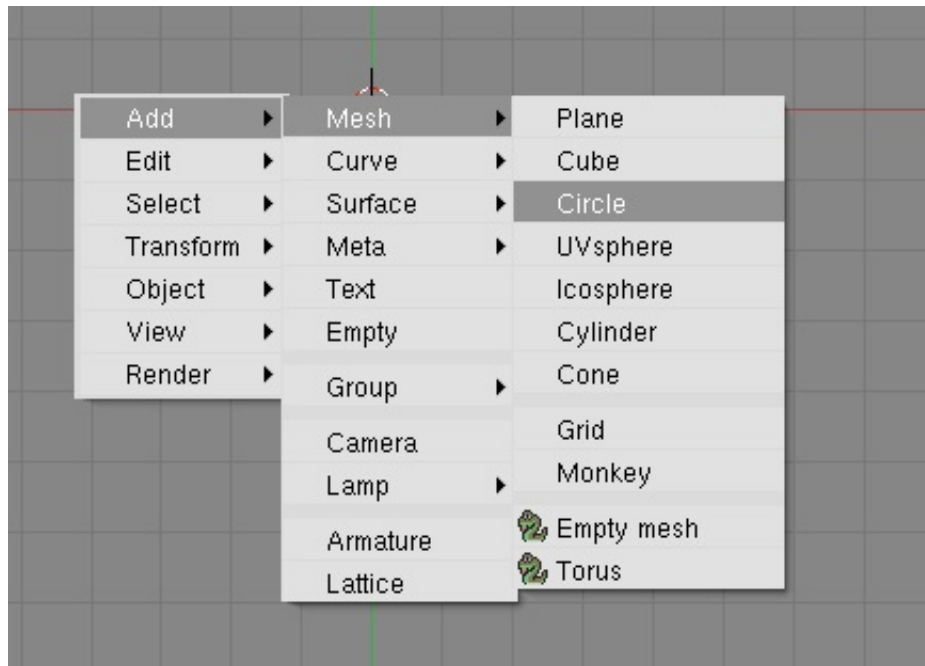
So, let's have a look at another object and see how it gets stored in Blender.

Move your mouse over to the object display (right hand split in my case) and press **A-key** to select all vertices (if they are not already selected, when selected they are highlighted in pink). Press the forward delete key and enter to delete the cube. Doesn't matter if you take the camera and light source with it.

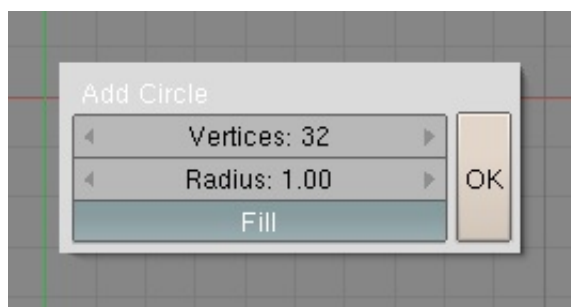


Put the cursor at the intersection of the green and red grid lines so it is centred. If you have a blue and red grid lines, you're in a different view so press the **Numpad-7** key to switch to top view or, go View-Top if you've got one of those new iMac keyboards without the numeric keypad (that would annoy me to death not having a numeric keypad).

Press **Space** to bring up the menu, choose "Add->Mesh->Circle" like this:

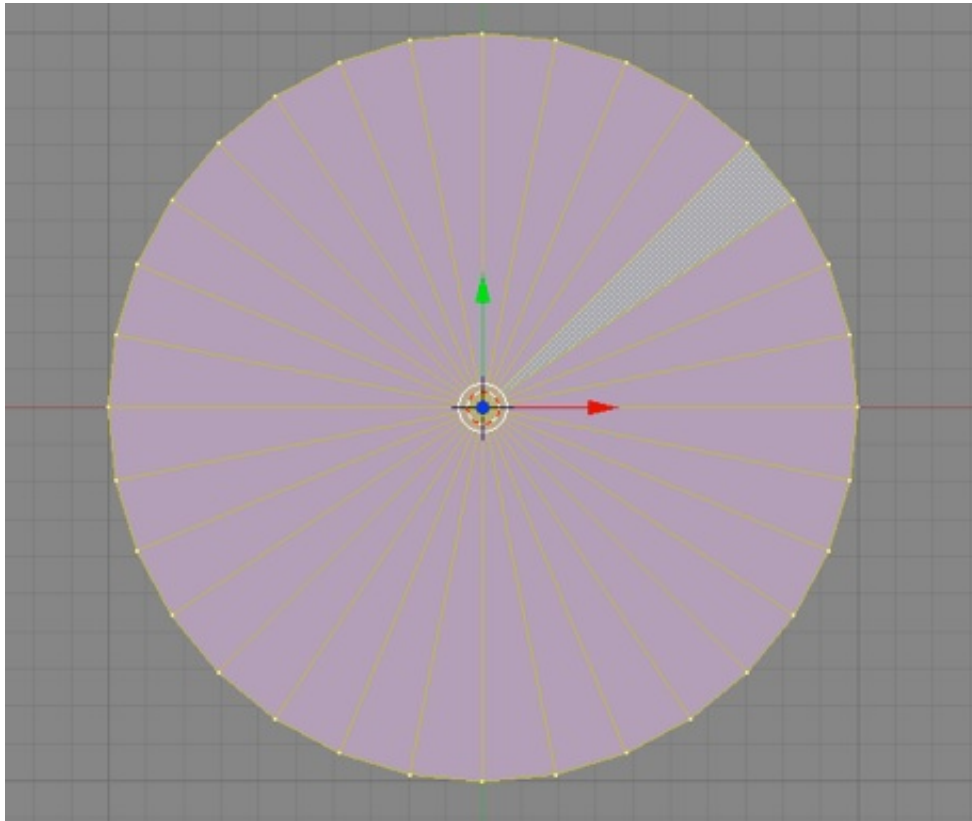


In the Add Circle window, click on the "Fill" button so it's selected (it's the darker of the two colours and leave the rest as is.



Click OK.

This gives us a "filled" circle. Press the **TAB** key to go into Edit Mode so we can have a look at how Blender has constructed this. You should see this:



You may notices straight away that the circle is divided into triangles as opposed to quads. Let's confirm this in the interactive python console.

## Discovering the Circle

With your mouse over the split with the console (left hand side for me), let's start by doing the discovery as we did before with the cube. Here's what I entered:

```
print Blender.Object.Get()
[[Object "Circle"], [Object "Cube"]]
circle = Blender.Object.Get("Circle")
mesh = circle.getData()
print len(mesh.faces)
32
for face in mesh.faces:
    print len(face.v)
```

OK, so quite similar to what we did with the cube. The difference being is that Blender has constructed the circle out of triangles rather than quads. This is significant when it comes to drawing the circle in OpenGL.

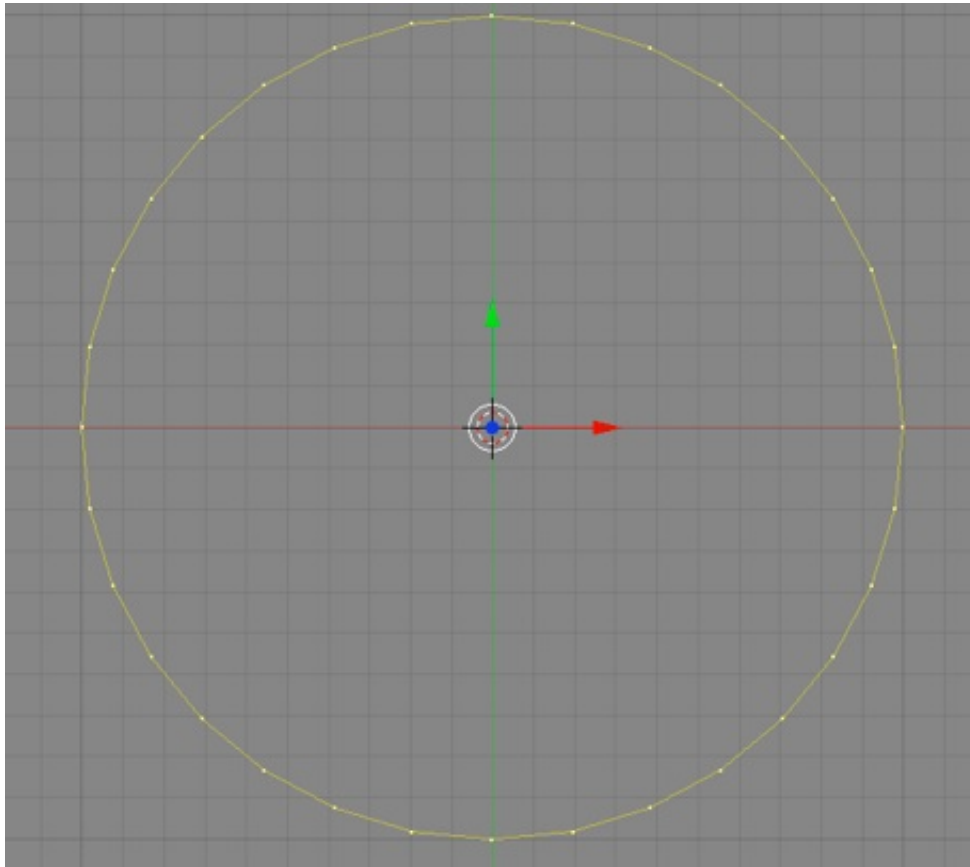
Note also that there are 32 faces. The circle was built from 32 vertices, each one ended up as a face.

You can then go on to discover the vertex co-ordinates etc like we did before.

### Faceless Objects

Right, before we start doing a simple export, let's look at an object without a faces and see how they're stored. Delete the circle and create a new circle using the same method as before **but this time, do not select "Fill"**, make sure the "Fill" button is the lighter colour.

Press **TAB** to enter edit mode and you'll see the difference straight away.



Now, without my help, go into the Python console, and discover how many faces there are.

Who was surprised by the output. I got this:



```

print Blender.Object.Get()
[[Object "Camera"], [Object "Circle"], [Object "Lamp"]]
circle = Blender.Object.Get("Circle")
mesh = circle.getData()
print len(mesh.faces)
0

```

So this is an object without face data. Therefore, in order to get at the vertex co-ordinates, we can simply access them at the mesh level rather than the face level.

```

print len(mesh.verts)
32
for v in mesh.verts:
    print v.co

[0.707107, 0.707107, 0.000000](vector)
[0.831470, 0.555570, 0.000000](vector)
[0.923880, 0.382683, 0.000000](vector)
[0.980785, 0.195090, 0.000000](vector)
[1.000000, 0.000000, 0.000000](vector)
[0.980785, -0.195090, 0.000000](vector)
[0.923880, -0.382683, 0.000000](vector)
[0.831470, -0.555570, 0.000000](vector)
[0.707107, -0.707107, 0.000000](vector)
[0.555570, -0.831470, 0.000000](vector)
[0.382683, -0.923880, 0.000000](vector)
[0.195090, -0.980785, 0.000000](vector)
[-0.000000, -1.000000, 0.000000](vector)
[-0.195091, -0.980785, 0.000000](vector)
[-0.382684, -0.923879, 0.000000](vector)
[-0.555571, -0.831469, 0.000000](vector)
[-0.707107, -0.707106, 0.000000](vector)
[-0.831470, -0.555570, 0.000000](vector)
[-0.923880, -0.382683, 0.000000](vector)
[-0.980785, -0.195089, 0.000000](vector)
[-1.000000, 0.000001, 0.000000](vector)
[-0.980785, 0.195091, 0.000000](vector)
[-0.923879, 0.382684, 0.000000](vector)
[-0.831469, 0.555571, 0.000000](vector)
[-0.707106, 0.707108, 0.000000](vector)
[-0.555569, 0.831470, 0.000000](vector)
[-0.382682, 0.923880, 0.000000](vector)
[-0.195089, 0.980786, 0.000000](vector)
[0.000002, 1.000000, 0.000000](vector)
[0.195092, 0.980785, 0.000000](vector)
[0.382685, 0.923879, 0.000000](vector)
[0.555572, 0.831469, 0.000000](vector)

```

### Quick Wrap Up

So, you've seen some of the different types of objects in Blender and the differences in exporting them. Fortunately, for our purposes, this is where we get an advantage over people who "just use" a pre-made format like LWO, 3ds etc. You see, it is very unlikely that you would want all three different styles that we've seen in a single file.



For example, the game that I'm writing at the moment, I only need the last style for several of my models; objects without faces because I am extruding them out at run time for different purposes. In other cases, I need to bring out pre-made quads for other objects which are **unrelated** to the simple co-ordinate types which I use different code for. That's they key.

*Even if I used the same file format, I would still need different code for loading and rendering the two different object types even within the same program.*

That's true even if I used a commercial file format such as 3ds or LWO.

By keeping to code short, to the point, and efficient, I get a distinct advantage over the heavyweight file formats.

### That's It for This Tutorial

iWeb has obviously got enough for one blog as it's slowing down again as I type. So time to bring this one to a close and to write the next entry.

Don't worry, it's uploaded now as I'm uploading them both at the same time. So for now, no need to wait, let's load something in part 2...

See you then. Hooroo!  
Simon Maurice

Copyright 2009 Simon Maurice. All Rights Reserved.

The code provided in these pages are for educational purposes only and may not be used for commercial purposes without Simon Maurice's expressed permission in writing. Information contained within this site cannot be duplicated in any form without Simon Maurice's expressed permission in writing; this includes, but is not limited to, publishing in printed format, reproduced on web pages, or other forms of electronic distribution.

Linking to these pages on other websites is permitted.

[← previous](#)

[next →](#)

