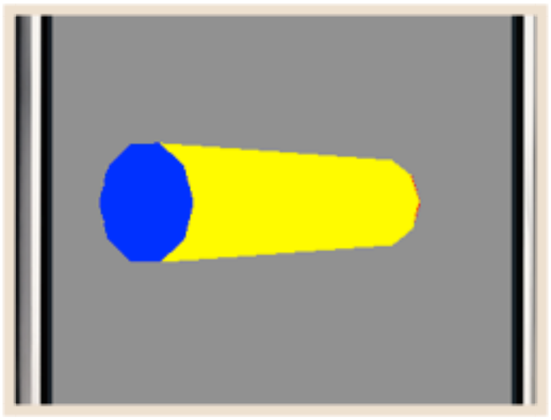
SIMON MAURICE - IPHONE - OPENGL ES



Tuesday, 28 July 2009

OpenGL ES 18 - Monkeys on Your Back and Geometric Shapes

The past week....

I've made a sudden change of direction. You see, I'd been working on a game for the iPhone, sort of like a flying game where you could do such god-like things as deform landscape in real time etc. Well, I got it up and running on the device with most of the engine complete and it hummed along at 60fps (how I got there is another story, one I'll certainly be sharing) with many frames needing to deal with over 100k textured triangles.

The only trouble is, it just didn't work as a game.

The whole lack of keyboard issue reared it's ugly head. Whilst I brought in an on screen keyboard (icon board actually) as needed, it was just too awkward without the physical keyboard. The small screen really didn't bring out the best in the rendering engine either. Ditch that game idea for the iPhone at least. I'll press that code into service on the desktop though some time soon, it will work there.

Never mind. You need to bounce back from these things and that very night I did. With the next day off work and my wife already fast asleep in bed (it was around 11:30PM), I realised the kind of game for the iPhone that I *should* be writing were the kinds of games that *I like to play* on the device.

So it was like: 2D scrolling shoot ‘em up, here I come!

By the time I went to bed at 4AM (chased into bed actually as wifey woke up wondering where I was), I had most of the engine going. I was loading a depth sorted tile based map, parallax scrolling was working nicely, sprites were taking form nicely; you get the idea. It was coming together.

Now I just need to make a game out of it. I already know what I’m doing for the game design, I just need to make all the graphics for it which I’m doing at the moment. Right now, I think I’ll have it completed in around 4 weeks and submitted into the App Store. Once that’s done, I’ll release the code under the GNU GPL and post it here somewhere.

I know I’m on the right track because, despite the fact I haven’t even finished this game, I’ve already got the next one going in my head so once that’s done, the next one will be under development straight away. Actually, it already is. Again, watch this space for the code release.

In the meantime, I’ve got to cut down the volume of emails that I’m getting. I get way too many at the moment so I’ve started a FAQ page on this website and I’ve got to get a few common questions out of the way. And here’s one monkey that I want to get off my back:

Those Damn Geometric Shapes Questions
I do get quite a volume of emails from this tutorial series as you can probably imagine. I don’t like the idea of putting myself out there on a pedestal, pontificating to all but “don’t call me if you don’t understand”. That doesn’t work for me so that’s why I have no problem putting an email me link on every tutorial.

The only thing has been this constant damn request for “how do I do a cylinder”, or “how about a sphere”, or “how about a penta-poly-hecto-hedrogram”?

So far I’ve been fairly resolute. You see, drawing these kind of objects is not a graphics problem, it’s a maths problem. Also, you tend not to draw a lot of these from code in the “real world”; most would come from say Tutorial #15 where you would load up a sphere in Blender and export that bad boy. To hell with computing that on the fly unless you really need to.

I can’t remember the last time I drew a sphere programatically. I know I have done it but I think it was several years ago. It’s more classroom stuff rather than something useful. It’s like going through a tutorial and drawing a polyhedron. Yeah, nice, but whoopeee. What are you going to do with it?

Like I said, the requests keep coming so that’s what’s happening here. White flag is raised and today I’m covering:

Circles
Well, circles are actually really handy to know. That’s all a cylinder is: just a circle which has been extruded along it’s length. One of the challenging things for the beginner in OpenGL ES is that you can’t just call up a circle like you can a triangle or quad. You’ve got to create it. For that, you do need to know primarily what you want in the circle (eg

filled, unfilled, smoother etc) and a little bit of maths principles and you can get what you want.

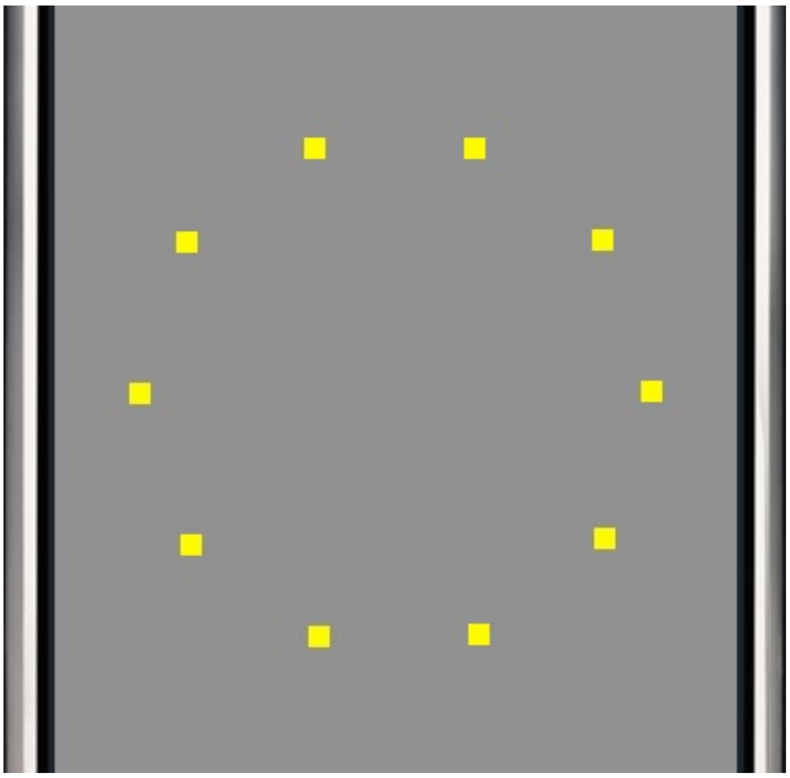
I know I've mentioned this before, but this is how you can describe a circle with origin (centre) at point (Cx, xy), with radius r:

```
x = Cx + r * cos(angle)
y = Cy + r * sin(angle)
```

So, given only the origin and the radius, you can draw a circle. It's at this point that usually people stop me and say "yes but you also need to know the angle which is a final unknown in the equation!". No, it's not unknown. You simply start at the angle of a 0º (actually radians) and continue to trace out the circumference.

So we can start by filling out a vertex array with the individual points on the circumfrence of a circle that we want to plot.

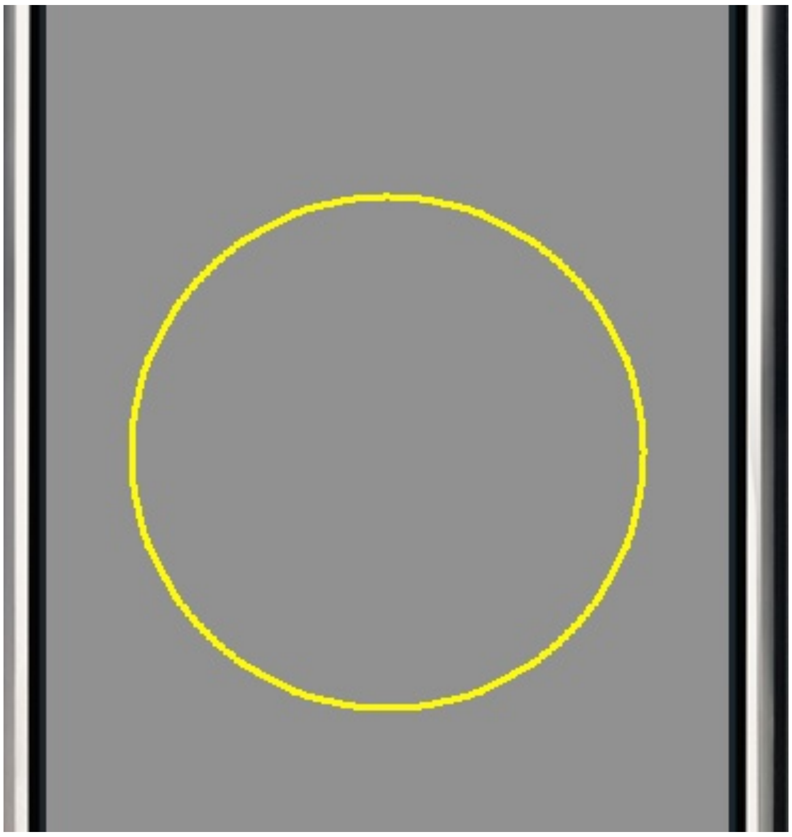
```
    GLfloat points[20];
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, points);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glPointSize(10.0);
    int i = 0;
    float radius = 0.75;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        points[i++] = radius * cos(angle);
        points[i++] = radius * sin(angle);
    }
    glDrawArrays(GL_POINTS, 0, 10);
```



Obviously, by shrinking the increment on the angle variable will bring the points closer together.

```
    GLfloat points[722];
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, points);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glPointSize(3.0);
    int i = 0;
    float radius = 0.75;
    for (float angle = 0; angle < 2*M_PI; angle += (2*M_PI)/360)
{
        points[i++] = radius * cos(angle);
        points[i++] = radius * sin(angle);
    }
    glDrawArrays(GL_POINTS, 0, 360);
```

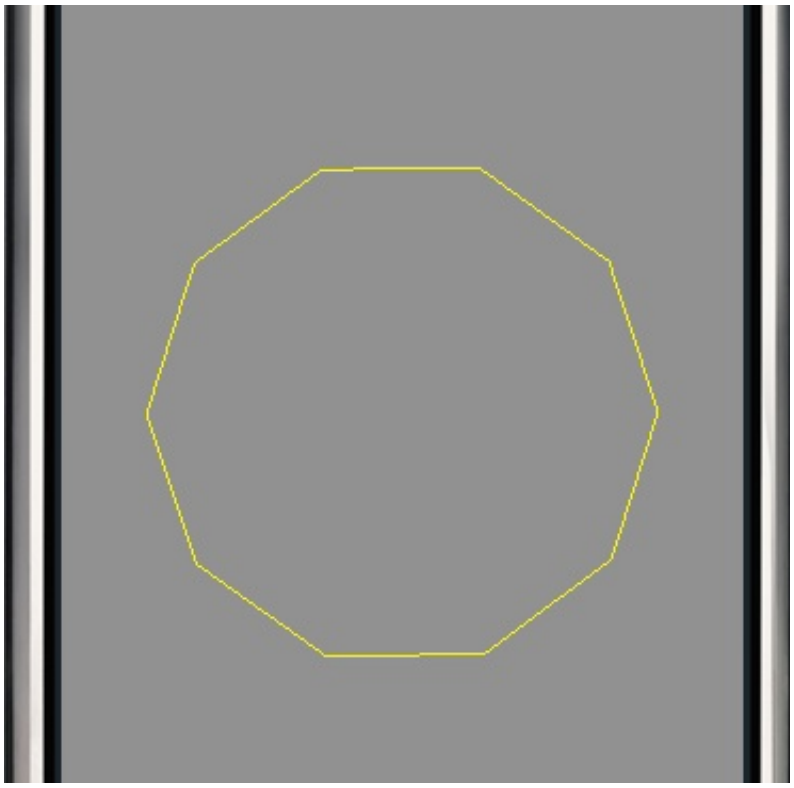
By rendering 360 2.0 sized points, you'd get this:



I actually made the array 722 because I was concerned about a rounding situation which would give an extra point so I just made the array large enough just in case. In reality, 720 was fine.

Switching the code back to the original circle and changing `GL_POINTS` to `GL_LINE_LOOP`, you get this:

```
    GLfloat points[20];
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, points);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    int i = 0;
    float radius = 0.75;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        points[i++] = radius * cos(angle);
        points[i++] = radius * sin(angle);
    }
    glDrawArrays(GL_LINE_LOOP, 0, 10);
```



Filled Circles
So far, nothing really too hard. Just say we want to fill the circles in above? This is where our old rendering friend GL_TRIANGLE_FAN comes into play.

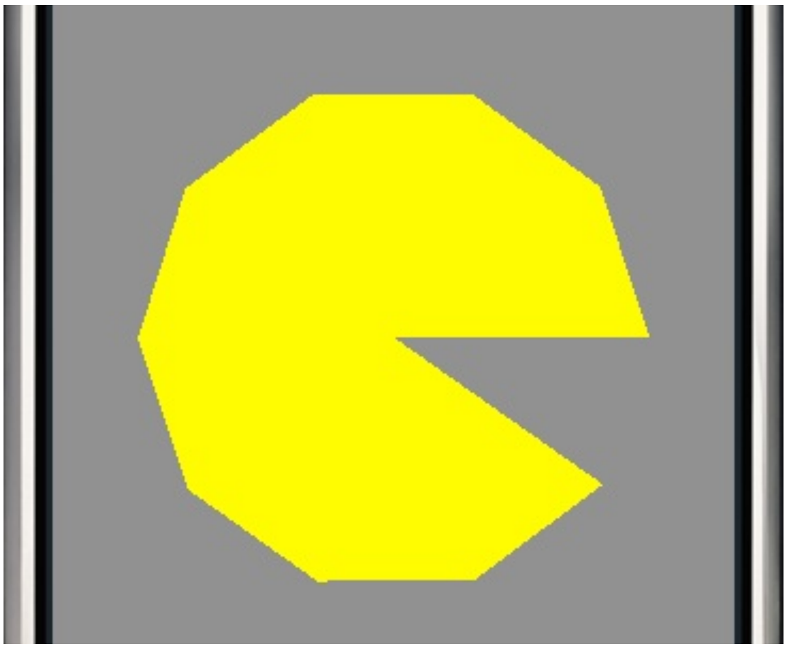
Now, remember passing the parameter GL_TRIANGLE_FAN will render the first three points of the vertex array as a triangle, then render a triangle formed by *each subsequent vertex and the first vertex*.

So, to make the circle above filled, the first point in the vertex array becomes the centre. The first two points form the first triangle. Then each indvidual vertex after will form a triangle with the centre and the previous vertex.

So this code:

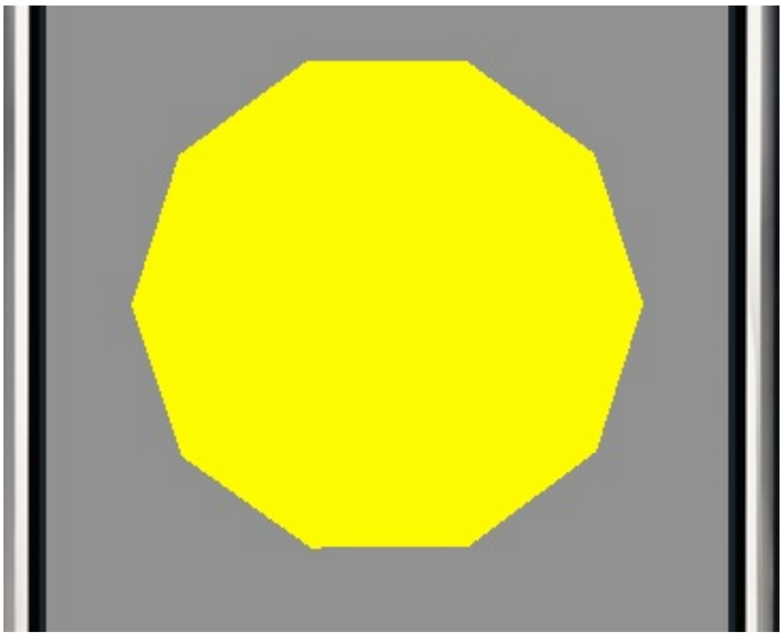
```
    GLfloat points[22];
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, points);
    glColor4f(1.0, 1.0, 0.0, 1.0);
points[0] = 0.0;// Circle centre
points[1] = 0.0;
    int i = 2;
    float radius = 0.75;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        points[i++] = radius * cos(angle);
        points[i++] = radius * sin(angle);
    }
    glDrawArrays(GL_TRIANGLE_FAN, 0, 11);
```

generates this:



Errrrrr.... opps! Looks like Pacman! In reality, if you think about it, we need another vertex co-ordinate in order to get GL_TRIANGLE_FAN to fill the circle completely. We need to link the last vertex back with the first vertex to make that last triangle. So what we need to do is:

```
    GLfloat points[24];
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, points);
    glColor4f(1.0, 1.0, 0.0, 1.0);
points[0] = 0.0;// Circle centre
points[1] = 0.0;
    int i = 2;
    float radius = 0.75;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        points[i++] = radius * cos(angle);
        points[i++] = radius * sin(angle);
    }
points[22] = points[2];
points[23] = points[3];
    glDrawArrays(GL_TRIANGLE_FAN, 0, 11);
```



Whilst that's not the only way to fill a circle, it's probably the easiest and best on most OpenGL platforms. Other methods such as using lines don't really work that well on the iPhone as the line drawing engine is fairly poor (might be improved on the 3GS though).

Circles into Cylinders
Now, we can take a circle and turn it into a cylinder fairly easily. I'm going to switch to a 3D view though just to make the depth work for me.

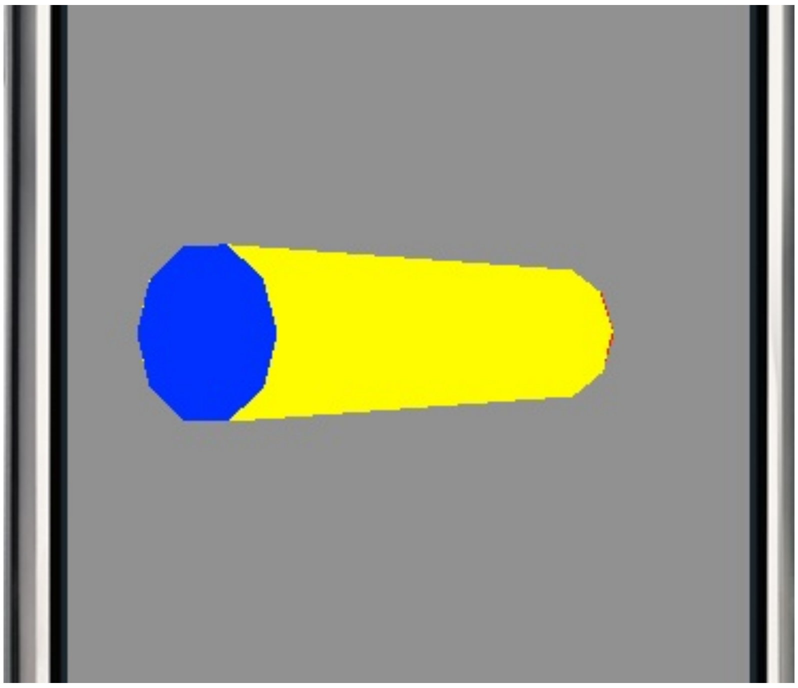
So, use this project base:

```
    GLfloat frontCircle[30];             // Now have X, y, Z
    GLfloat rearCircle[30];
    float radius = 0.5;
    GLfloat origin[2] = {
        0.0, 0.0
    };
    int i = 0;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        frontCircle[i] = origin[0] + radius * cos(angle);
// X
        frontCircle[i+1] = origin[1] + radius * sin(angle);
// Y
        frontCircle[i+2] = 4.0;      // Z, somewhere off behind
the viewer

        rearCircle[i] = frontCircle[i];
        rearCircle[i+1] = frontCircle[i+1];
        rearCircle[i+2] = -4.0;            // Off in front of us
        i += 3;
    }
    rota += 0.05;
    if (rota > 360) rota = 0;
    glRotatef(1, 0.0, 1.0, 0.0);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, frontCircle);
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_LINE_LOOP, 0, 10);

    glVertexPointer(3, GL_FLOAT, 0, rearCircle);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_LINE_LOOP, 0, 10);

    GLfloat line[6];
    glVertexPointer(3, GL_FLOAT, 0, line);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    for (int i = 0; i < 10; i++) {
        line[0] = frontCircle[i*3];
        line[1] = frontCircle[i*3+1];
        line[2] = frontCircle[i*3+2];
        line[3] = rearCircle[i*3];
        line[4] = rearCircle[i*3+1];
        line[5] = rearCircle[i*3+2];
        glDrawArrays(GL_LINES, 0, 2);
    }
```

OK, so that's a wireframe cylinder, you can fill it in by drawing the circles like we did last time and the sides can be filled by converting them into quads like so:

```
    GLfloat frontCircle[33];            // Now have X, y, Z
    GLfloat rearCircle[33];
    float radius = 0.5;
    GLfloat origin[2] = {
        0.0, 0.0
    };
    int i = 0;
    for (float angle = 0; angle < 2*M_PI; angle += 0.630) {
        frontCircle[i] = origin[0] + radius * cos(angle);
// X
        frontCircle[i+1] = origin[1] + radius * sin(angle);
// Y
        frontCircle[i+2] = 2.0;           // Z, somewhere off
behind the viewer

        rearCircle[i] = frontCircle[i];
        rearCircle[i+1] = frontCircle[i+1];
        rearCircle[i+2] = -2.0;           // Off in front of us
        i += 3;
    }

    frontCircle[30] = frontCircle[0];
    frontCircle[31] = frontCircle[1];
    frontCircle[32] = frontCircle[2];
    rearCircle[30] = rearCircle[0];
    rearCircle[31] = rearCircle[1];
    rearCircle[32] = rearCircle[2];

    glPushMatrix();
    rota += 0.5;

    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rota, 0.0, 1.0, 0.0);
    glVertexPointer(3, GL_FLOAT, 0, frontCircle);
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 10);

    glVertexPointer(3, GL_FLOAT, 0, rearCircle);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 10);

    GLfloat side[12];
    glVertexPointer(3, GL_FLOAT, 0, side);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    // For each section we need to form a quad with the two front
and two rear sets of vertices
    for (int i = 0; i < 10; i++) {
        side[0] = frontCircle[i*3];
        side[1] = frontCircle[i*3+1];
        side[2] = frontCircle[i*3+2];

        side[3] = frontCircle[i*3+3];
        side[4] = frontCircle[i*3+4];
        side[5] = frontCircle[i*3+5];

        side[6] = rearCircle[i*3];
        side[7] = rearCircle[i*3+1];
        side[8] = rearCircle[i*3+2];

        side[9] = rearCircle[i*3+3];
        side[10] = rearCircle[i*3+4];
        side[11] = rearCircle[i*3+5];
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    }
    glPopMatrix();
```

Add in a translate and a rotate to spin that bad boy and you'll get:



Obviously, you wouldn't re-generate all the geometry every frame in a "real world" situation but I just decided to make it all a bit easier and drop it in there. Again, also all you need to do to make it smoother is just to increase the quantity of "sections" which make up the circle(s).

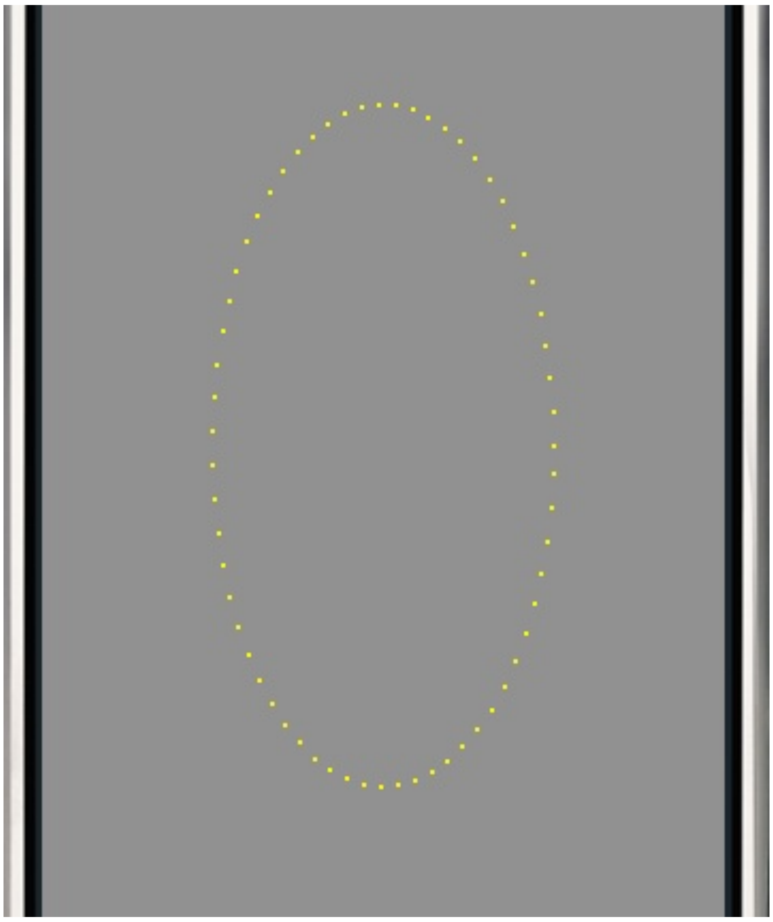
Notice the errors in the above image? You can see part of the red circle which should be fully occluded and the yellow on the left of the blue circle. I am assuming they are rounding errors which aren't too much of a problem. Personally, I wouldn't bother fixing them because in a real world situation, you're going to have more than just a flipping cylinder being rendered. If someone playing a game has the time to notice little faults like that, then you've really done something wrong in your game-play design.

Ellipses
These fellas are just as easy to draw. All you need to realise about an ellipse is that it's a circle with two radii. From memory, they're called the major axis and the minor axis rather than radius. I tend not to think of them as a major or minor, typically just as an X radius and a Y radius. So have a look at this:

To draw an ellipse, you do:

```
GLfloat xradius = 0.25;
GLfloat yradius = 0.5;

GLfloat point[2];

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(2, GL_FLOAT, 0, point);
glColor4f(1.0, 1.0, 0.0, 1.0);
glPointSize(2.0);

for (float angle = 0; angle < 2*M_PI; angle += 0.1) {
    glPushMatrix();
    point[0] = cos(angle)*xradius;
    point[1] = sin(angle)*yradius;
    glTranslatef(point[0], point[1], 0.0);
    glDrawArrays(GL_POINTS, 0, 1);
    glPopMatrix();
}
```

Which gives you:



So in other words, it’s just like drawing a circle except you have to deal with two radii.

Everything else applies just like we did for a circle.

That’s It for Today
I’ll leave it here for now. I’ll be back again in a couple of days with something else. I didn’t bother with a final tutorial project here because all you need to do is insert the code in the draw method in the project above.

Until next time, hooroo
Simon Maurice

previous next