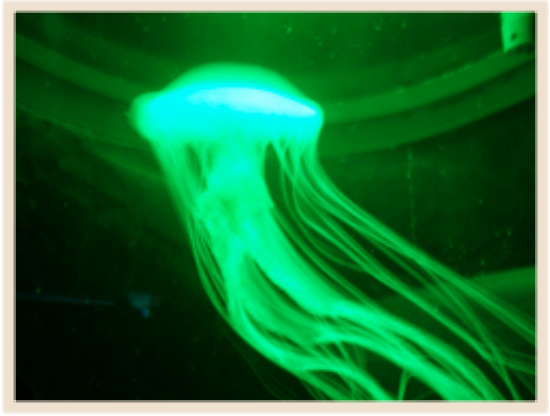
SIMON MAURICE - IPHONE - OPENGL ES



Monday, 1 June 2009

OpenGL ES 13.5 - Moving in 3D Part 2: Some Theory that I Should Have Explained

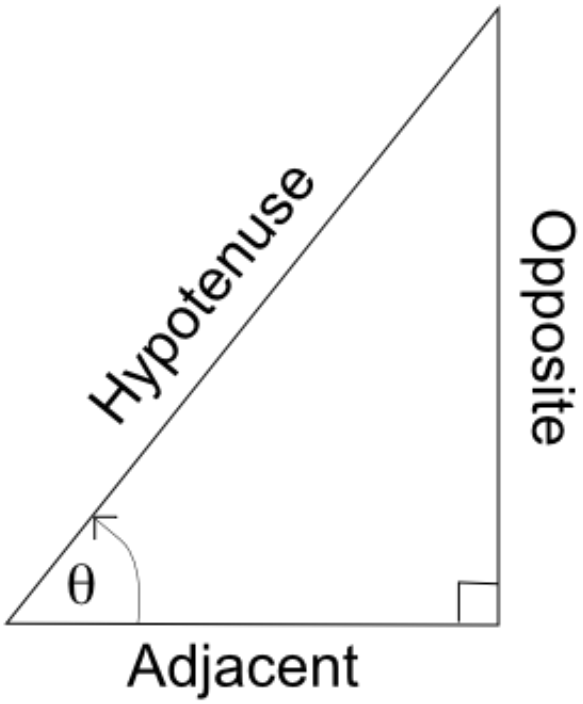
I meant to get this tutorial up yesterday but last night was the first State of Origin match for 2009. For those of you who don't know, State of Origin is Rugby League's superbowl, except that it's three matches a year instead of one and is played between the Australian states of New South Wales (where I live) and Queensland every year. There's no better spectacle in the rugby league world and it really defines the world champions. There's no other side in the world, club or national team, that can better the State of Origin winners. So last night was a matter of a few too many pints after work in the pub and me not writing this...

This is just a quick part 2 to the last tutorial. It's a bit of a back to maths basics as some people have emailed me with questions on just what I meant by drawing a circle and how that relates to turning or rotating in 3D graphics.

I'm starting from scratch, talking right-angled triangles and leading right through to how that circle and triangle thing make turning possible with the `gluLookAt()` function.

The Right-Angled Triangle

No, I'm not going to explain this to you. I need you to know what a right angled triangle is, what the hypotenuse is, and what an opposite and adjacent sides are. Just in case in your part of the world, you use different terminology, I am going to put the picture here just to be sure that when I say adjacent, you know exactly what I am referring to:



I believe in North America, you guys refer to this as a Right Triangle. That's just an example of the differences in terminology that I want to avoid before moving on.

I'm not going to explain these, I'm just reminding you of the stuff you learned in maths in high school but have forgotten because it wasn't really applicable.

The length of any side can be worked out as easily as:

Hypotenuse: $\text{hypotenuse}^2 = \text{adjacent}^2 + \text{opposite}^2$
in C: `hypoenuse = sqrt(adjacent*adjacent + opposite*opposite);`

Adjacent length: cosine (Θ) * hypotenuse.
in C: `adjacent = cos(angleInRadians) * hypotenuse;`

Opposite Length: sine(Θ) * hypotenuse
in C: `oppositeLength = sin(angleInRadians) * hypotenuse;`

The first thing to remember is that the C maths functions work in radians, not degrees. Don't worry too much about radians, just remember how to covert between the two with these two macros:

```
#define DEGREES_TO_RADIANS(__ANGLE) ((__ANGLE) / 180.0 * M_PI)
#define RADIANS_TO_DEGREES(__RADIANS) ((__RADIANS) * 180 / M_PI)
```

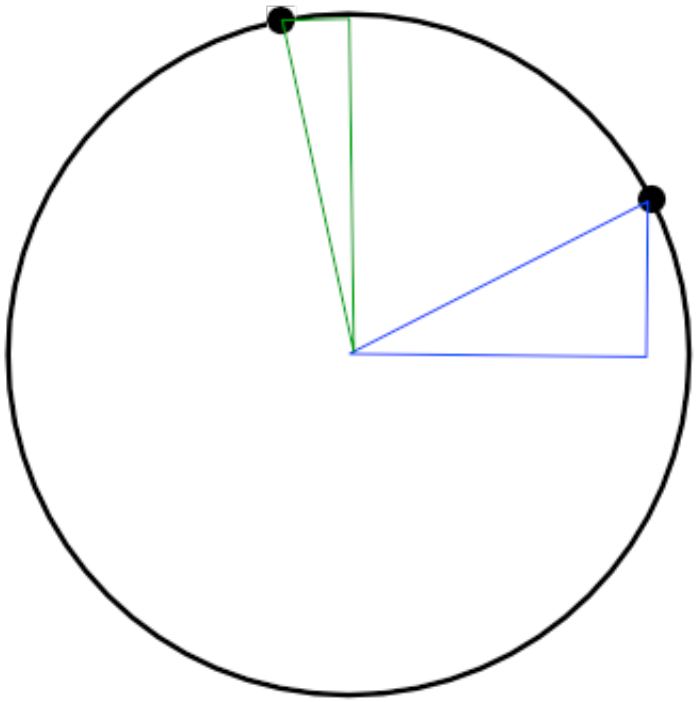
The easy way to remember the relationship between radians and degrees, you already know there are 360º in a circle so just remember there are 2 x Pi radians in a circle. Or Pi radians in a semi-circle.

With the GCC maths library, M_PI is defined in the math.h file as being 3.14-whatever. It's not actually to the C standard. The C standard doesn't allow for a definition of Pi. You're technically supposed to compute it for yourself so you can have it as accurate as possible for your hardware implementation (it's just acos(-1) from memory). I've never found a time when I've needed to use anything other than the defined value when coding for many years now.

If you can't remember what is sin(), cos(), & tan() do, don't worry about the theory behind them. It's just like driving a car. You don't need to know how an internal combustion engine works to drive a car, you only need to know how (and when) to use it.

Linking this Back to Graphics Programming
Remember in the last tutorial, I discussed moving in 3D and turning. I said it's just like drawing a circle. Let me explain this in more detail first by showing you how you can draw a circle.

Here is a picture of a circle with 2 points of the circle plotted.



Did you notice something? Have another look if you can't see it yet.

Yes, they are right-angled triangles!

So, we can now proceed to plot any point on the circle by knowing only the radius and angle. Therefore, we can now actually ***draw*** a circle.

To draw a circle, we know the length of the hypotenuse because that's our circle's radius. All we need to solve is the X and the Y. Given the functions above and referring to the blue triangle, we can work out the X co-ordinate as simply the adjacent side and the Y as the opposite side.

Of course, the adjacent and opposite sides change in depending on each quadrant but we don't need to concern ourselves with that. The code to draw the circle is just:

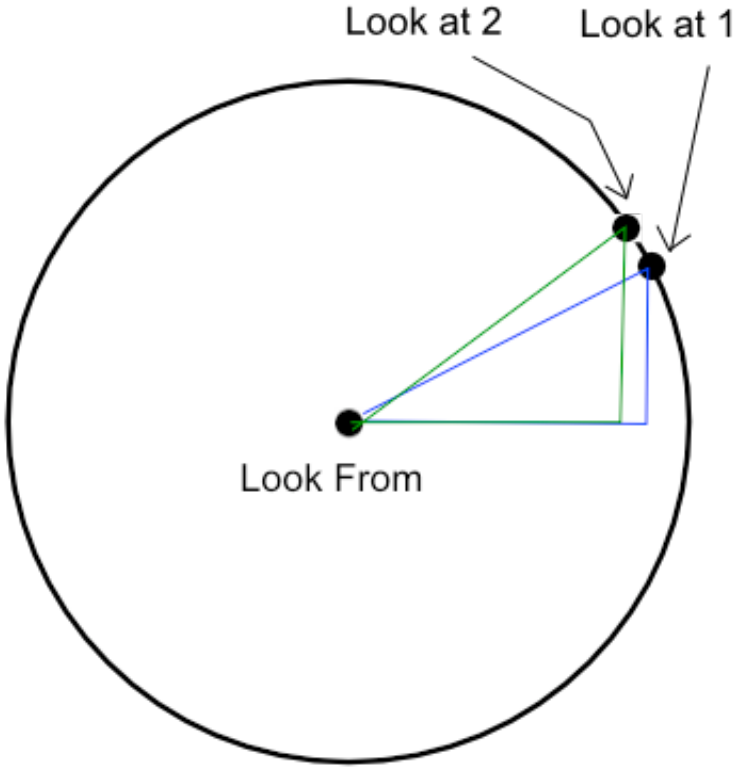
```
for (angle = 0; angle < 2*M_PI; angle += 0.01) {
    point[0] = RADIUS * cos(angle);
    point[1] = RADIUS * sin(angle);
    glDrawArrays(GL_POINTS, 0, 1);
}
```

Just remember that we need to pass sin() & cos() radians, not degrees. There are 2 * pi radians in 360°. The finer grade to the angle increment in the for loop, the neater the circle will be. In most cases, you are likely

to use very short lines instead of points to get a nice neat circle.

Linking Circles Back to Turning in the Last Tutorial
Right. Remember that we used `glLookAt()` in the last tutorial to decide where we were going to be be looking at from any point in our 3D world? I said it was just like drawing a circle.

Now, if the user decides to turn in an anti-clockwise direction, the view would be like this:



I'm sure you know where I'm going with this. Right-angled triangles describing a circle. :)

So, if you recall, we had our look from position and our look at position. Our look from position doesn't change, we just move our look at position around the circle's circumference as we rotate.

The code was:

```
facing[0] = position[0] + cos(-TURN_SPEED)*vector[0] -

sin(-TURN_SPEED)*vector[2];
facing[2] = position[2] + sin(-TURN_SPEED)*vector[0] +

cos(-TURN_SPEED)*vector[2];
```

The variable facing is where we are looking at, our look at point on the circle. The position variable is where we are looking from. It just looks more complicated because it's in a 3D world rather than a 2D paper

drawing. Even though I didn't calculate Y co-ordinates because I fixed it, you still need to calculate bot the X and Z co-ordinates.

That's it for Today

I just wanted to cover this topic in a bit of detail for you. I'll be putting up the next tutorial in a few days or so which is back on the main track as I mentioned before. The only issue is not the rendering but actually loading a file. I tried to get something under a free licence to use but nothing seemed suitable. Once I've got that done, I'll get it posted.

No code for this tutorial as it is more theory based. Hope that rounds off the last tutorial now.

Until next time, hooroo!
Simon Maurice

previous *next*