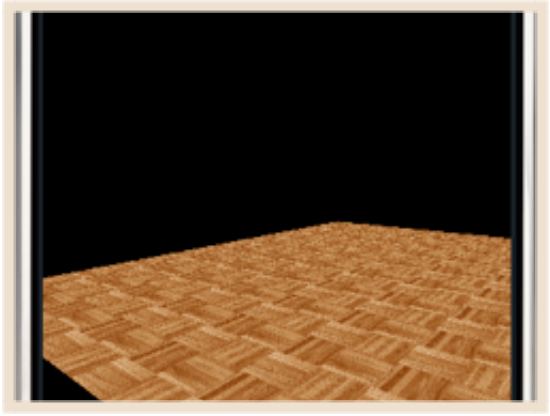
SIMON MAURICE - IPHONE - OPENGL ES



Monday, 18 May 2009

OpenGL ES 13 - Moving in 3D

Winter is almost upon us here in Sydney and I've been knocked down with the flu pretty bad. Nope, not the swine flu, just your standard every winter flu. The last time I had the flu this bad was way back in college when I was off for a week or so. I remember that time pretty well because I was reading the Stephen King book "The Stand" and that book, of course, deals with the super-flu taking over the world.

Lucky I'm not a Reg Barclay, or a Rodney McKay otherwise I may have been experiencing many sensations of anxiety reading that novel and feeling that way...

So the flu, busyness, mother's day, and a few other things have meant that I was absent from this blog for a while. So now it's time to get things going again.

Moving in 3D

I'm a bit pressed for time so this is going to be a real quick one here today. What we're going to do is to start to build a "genuine" 3D world, building everything up from the floor up. However, before I do that, I do want to introduce moving in a 3D world.

Today, what we're going to do is to add in new code to handle touches to walk around a "floor". By using touches, we can turn left, turn right, walk forwards and walk backwards. No running, dodging or head bobbing as we walk, they're easily added. I left them out because I wanted to keep it easy on me and still allow for people who don't have an iPod Touch or iPhone to be able to use this in the simulator.

So to get started, download the base for this project which is right here:

OpenGL ES 13 - Starter

There's not much coding going on, it's more about what and why we're going to do what we do.

The Mythical Camera
I think I've mentioned before that even though most of us refer to the 3D world as looking through a camera, there really is no camera in OpenGL per se. So if you want to make it appear that you are moving through a scene, you translate all the objects; the feeling of movement is created not by moving a camera like in the movies, but by moving the world relative to the (0, 0, 0) origin.

Whilst that sounds like a lot of work, it's not really. Depending on your application, there's a number of ways to do this and many, many more ways of optimising this for use on really large worlds. I'll make mention of that later on.

To make things a bit easier, I've brought with this tutorial's project a handy toy from OpenGL ES' big brother's GLU library: the `gluLookAt()` function.

While I normally don't refer to OpenGL in this series, I think that most of you should know what is the GLU library. Unfortunately it is not as part of the OpenGL ES specification but that doesn't mean that there are useful functions in there that we can't use. Porting these functions is easy and you don't need to port the entire library, just the select few that you need.

I've lifted the `gluLookAt()` function from the SGI's Open Source release, simply because that is what I had at hand and I know it works. There are other alternatives if you like. It's under a non-restrictive licence so it is OK to use here. Note that the code is not my copyright. The licence is there in the source code and, if you find it offensive, then there are other Open Source alternatives you can use.

If you want to use a different code base, or port over some other functions, the main thing that you have to do is change any `GLdouble's` to `GLfloat's`, change any associated `gl` calls to the

float version, and I'd say generally avoid anything which is user interface orientated (any windowing or input functions). There are many other things you need to watch out for (such has hardware related) but I find them to be fairly obvious.

I chose the SGI Open Source release as I had that on hand. For production use go for a free version which is up to date. Don't use the Mesa version, even the Mesa guys don't recommend it. Mesa isn't up to date and no longer being actively developed. I know there's code out there on the internet for the iPhone based on the Mesa GLU but it cannot be considered for production use (read: contains bugs).

Refer to the link on the Mesa home page titled SGI's GLU for more information on why they recommend the SGI or other library than their own.

Using `gluLookAt()`

This function is so simple to use and makes perfect sense once you get to know it. Let's have a look at the prototype:

```
void gluLookAt( GLfloat eyex,
                GLfloat eyey,
                GLfloat eyez,
                GLfloat centerx,
                GLfloat centery,
                GLfloat centerz,
                GLfloat upx,
                GLfloat upy,
                GLfloat upz)
```

I know 9 parameters can seem a little daunting at times but you just need to break it down a bit. The first three refer to the eye position, where you're looking from. Simply just an X, Y, Z co-ordinate.

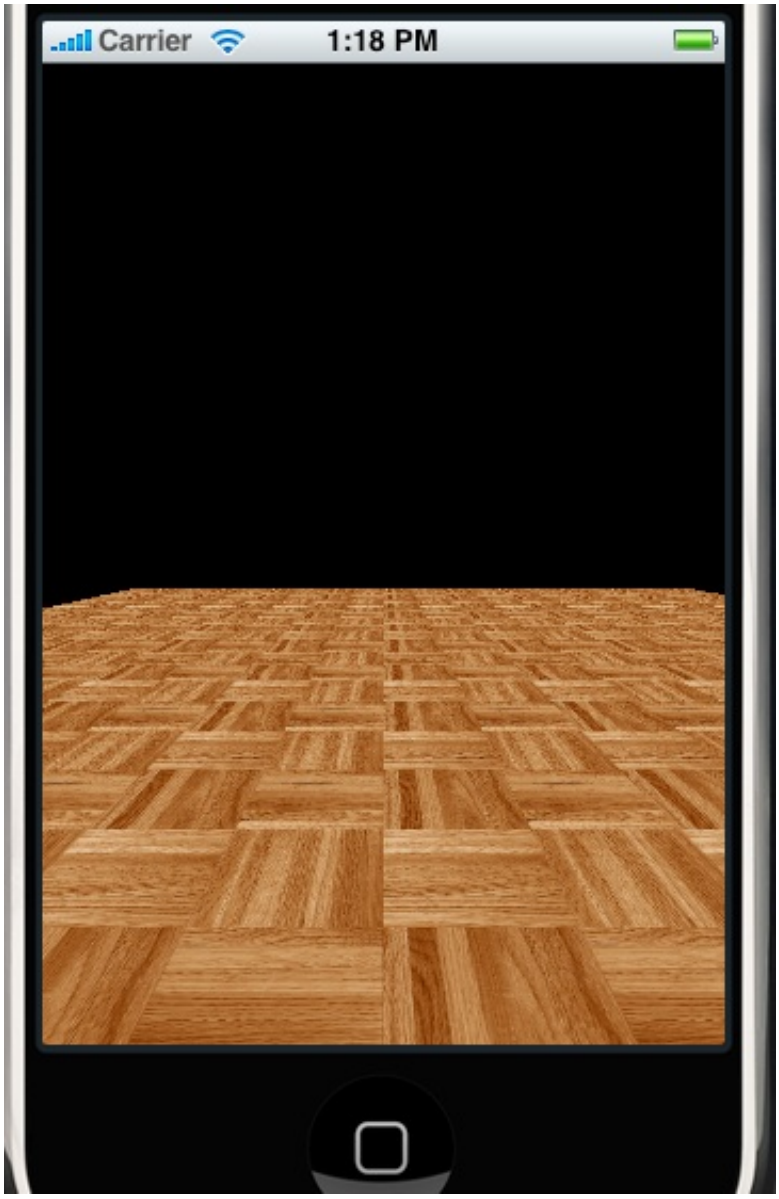
The second three refer to where you want to look at, again an X, Y, Z co-ordinate trio.

Finally, we can group the last three into the "`up`" vector. We'll not go into this trio now as it's really the first two co-ordinates which really gets us the effect we want.

So, the eye co-ordinates is that mythical camera position. They are in reference to your world co-ordinates of course. It's where you are looking from in your world. The "`center`" co-ordinates are where you are facing, your target for want of a better term. If the Y co-ordinate of the `center` is higher than your eye Y co-ordinate, you're looking up. If the Y is less than the eye Y co-ordinate, you're looking down and so on.

So, in our base project, we have it already set up but we do no

translations. All that happens is that we just draw a floor and look out into nothing-ness like so:



If you hit "Build and Go", that's what you'll get.

First of all let's use the `glLookAt()` function for the first time. Go to the `drawView:` method and add the following line right after the call to `glLoadIdentity()`:

```
    glLoadIdentity();
    gluLookAt(5.0, 1.5, 2.0,            // Eye location, look
"from"
              -5.0, 1.5, -10.0,         // Target location, look
"to"
              0.0, 1.0, 0.0);           // Ignore for now
```

Hit "Build and Go" again to satisfy yourself that all is working. You should get the following in the simulator:

With a single function call, we've shifted our looking from location to one corner and looking towards the opposite corner. Go ahead and have a play around with the parameters to `glLookAt()`. See what happens.

Movement in 3D

Now that you've got a grasp on `gluLookAt()`, let's use it to simulate walking on this floor. We're only going to move really along two axis (X and Z, in other words no height changes) and use turning in order to change direction.

So, thinking back to the `gluLookAt()` function, what pieces of information do you think you'll need in order to walk in a 3D world?

You need:

Your viewing location or "eye"
Your facing direction (target) or "centre"

Once you know those two, then you're ready to handle user input to allow the user to control where you are.

Let's assume that we want to start from the two values for eye and centre which we used earlier. That's that part sorted but of course hard coding them doesn't allow for movement so let's first head straight to the interface and add the following variables:

```
GLfloat eye[3];// Where we are viewing from
GLfloat center[3];// Where we are looking towards
```

I've just named them eye and `center` but you can call the "position" and "facing" if you like. It doesn't really matter; I just chose these because they match the `gluLookAt()` function.

These two variables hold the X, Y, and Z co-ordinates. I could have hard-coded the Y value as it's not changing but hey, why bother.

Next hit up the `initWithCoder:` method. Somewhere in there, initialise these two variables with the values we used previously for the call to `gluLookAt()`:

```
        eye[0] = 5.0;
        eye[1] = 1.5;
        eye[2] = 2.0;

        center[0] = -5.0;
        center[1] = 1.5;
        center[2] = -10.0;
```

Now, let's go back to the `drawView:` method. Change the call to `gluLookAt()` to:

```
  gluLookAt(eye[0], eye[1], eye[2], center[0], center[1],
center[2],
                0.0, 1.0, 0.0);
```

Hit "Build & Go" just to satisfy yourself. that everything has worked.

Getting Ready for Movement

Before we can start handling touch events to move around the world, we need to set a few things up for us in the header file. Switch to the header file so we can set some defaults and create a new enum type.

First, make some settings for how fast we walk and how fast we turn:

```
#define WALK_SPEED 0.005
#define TURN_SPEED 0.01
```

I did find these a little slow but you can change them to suit your own preferences once you've seen how they operate.

Next, we want to create an enumerated type so we can store exactly what we are doing. Add the following:

```
typedef enum __MOVMENT_TYPE {
MTNone = 0,
MTWalkForward,
MTWAlkBackward,
MTTurnLeft,
MTTurnRight
} MovementType;
```

So, at any point while we're running the app, we can be standing still (`MTNone`), walking forward, backward, turning left, or turning right. That's it I'm afraid for this tutorial, nothing too speccy yet.

Finally, we define a variable which will hold the current movement:

```
MovementType currentMovement;
```

Don't forget to go to the `initWithCoder`: method and set the default value for the `currentMovement` variable:

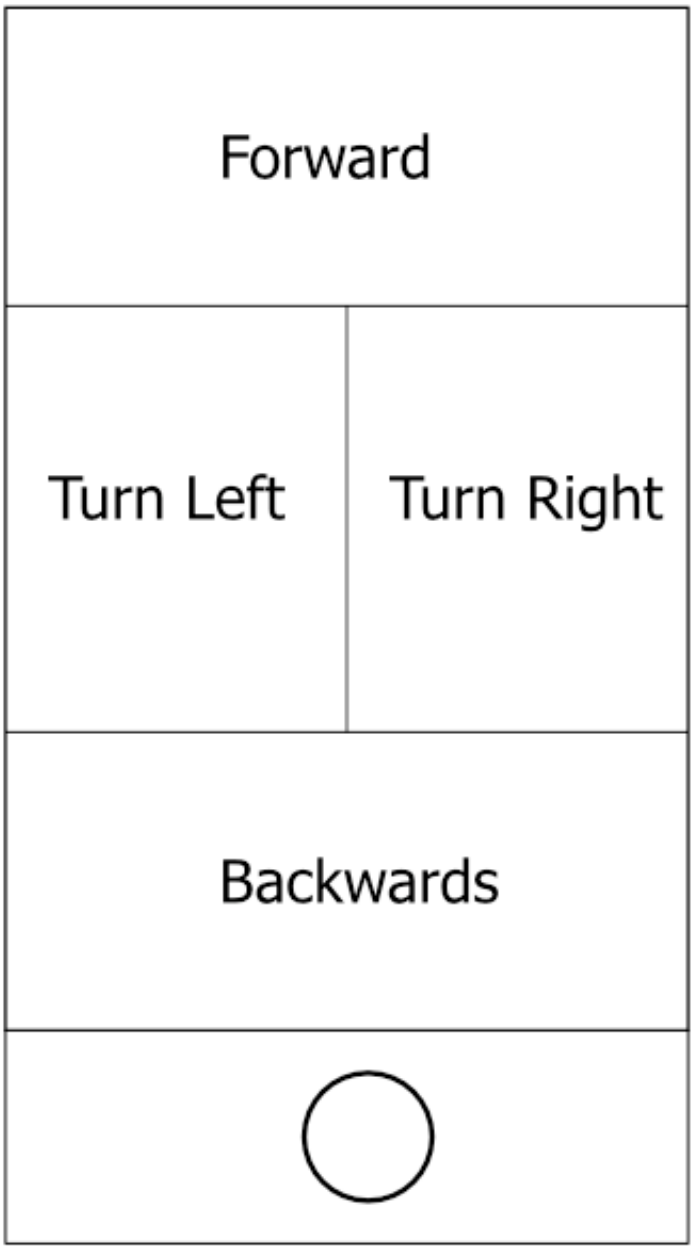
```
currentMovement = MTNone;
```

It should default to that being a variable but it's good practice to do.

Get Touchy Feely

Right, now we've got the basics out of the way, let's start working on the actual handling of the touches. If you remember back to the last tutorial, I introduced all four methods for handling touches. In this tutorial, we're only going to use two: `touchesBegan` and `touchesEnded`. Just going to keep it simple.

In order to determine what action to take, I've divided the iPhone's screen up into four segments:



Sorry for my poor drawing skills. Basically, the screen is 480 pixels high. Therefore, I can split it up into 3 even segments of 160 pixels. Pixels 0~160 is for going forward, 320~480 is for going backwards, and the middle 160 have been halved for the left and the right turning operations.

So, with that in mind, here's the first of the touches methods:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{

    UITouch *t = [[touches allObjects] objectAtIndex:0];
    CGPoint touchPos = [t locationInView:t.view];

    // Determine the location on the screen. We are interested in
iPhone
    // Screen co-ordinates only, not the world co-ordinates
```

```
    //  because we are just trying to handle movement.
    //
    // (0, 0)
    //  +-----------+
    //  |           |
    //  |    160    |
    //  |-----------| 160
    //  |     |     |
    //  |     |     |
    //  |-----------| 320
    //  |           |
    //  |           |
    //  +-----------+ (320, 480)
    //

    if (touchPos.y < 160) {
        // We are moving forward
        currentMovement = MTWalkForward;

    } else if (touchPos.y > 320) {
        // We are moving backward
        currentMovement = MTWAlkBackward;

    } else if (touchPos.x < 160) {
        // Turn left
        currentMovement = MTTurnLeft;
    } else {
        // Turn Right
        currentMovement = MTTurnRight;
    }
}
```

So, when the user starts touching, all we do is to record the segment and then set the variable so that when it comes time to calculating our new location, we know what we're doing. Also remember that we don't have to put a method definition in our interface; these methods are inherited.

Now for the touchesEnded method:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{

    currentMovement = MTNone;
}
```

Which is self explanatory. Finally we want a method to process these touch events. This time we need to add a method declaration in our interface. Switch to the header file, and add the following method definition:

```
- (void)handleTouches;
```

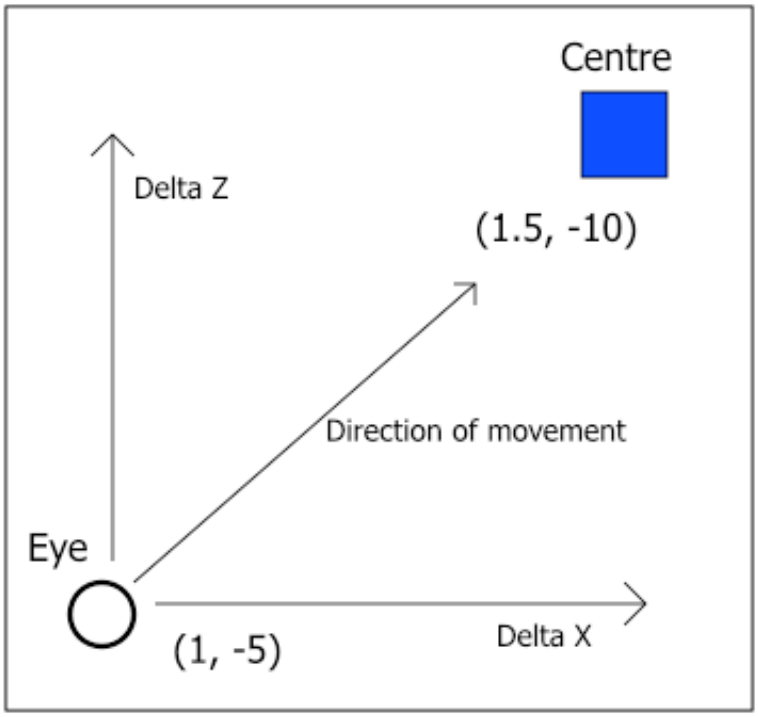
Then, switch back and start implementing this method. It's in this method that we're going to calculate our movements through the 3D world.

The Theory of Moving in 3D

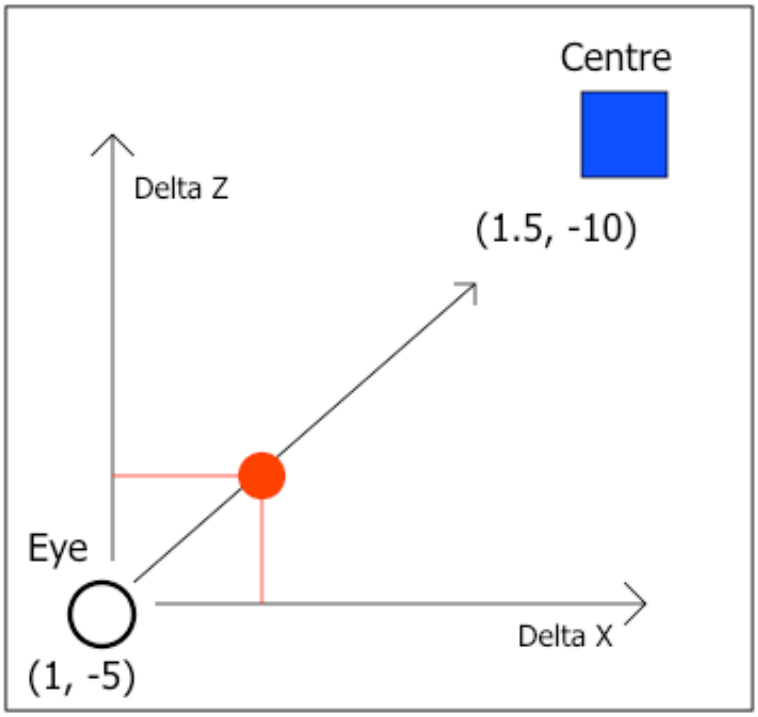
I've just got to cover some basics first of all. I'm sure you're not going to be surprised when I say that this is only one of the ways in which to calculate new locations in a 3D world given n units of movement along any vector v. I tried to remember who originally wrote this but can't locate the source. I thought it was Arvo but now I don't think so. I'll remember; it was a long time ago that's for sure and well before Wolf 3D even made people aware that you can do this in real time.

Let's look at walking first. If the user tells us to walk forward, we need to be aware of not only our looking from location, but also our target location. The looking from location tells us where we are currently, and our looking to target gives is the direction in which we are walking.

Since a picture tells a thousand words, have a look at the following picture showing our look from point and our look to target.



With this method of movement, we know the distance between the two points as an delta of the X co-ordinates and a delta of the Z co-ordinates. All we need to do to get the new X and Z values is the multiply the current co-ordinates by "speed" value. Like this:



We can easily solve the new co-ordinates for the red point.

We start with the deltaX and deltaZ:

```
deltaX = 1.5 - 1.0 = 0.5
deltaZ = -10 - (- 5.0) = -5.0
```

So now we multiply by our walk speed:

```
xDisplacement = deltaX * WALK_SPEED
              = 0.5 * 0.01
              = 0.005

zDisplacement = deltaZ * WALK_SPEED
              = -5.0 * 0.01
              = 0.05
```

Therefore the new co-ordinate indicated by the red dot in the image above is: eyeC + CDisplacement

```
    (eyex + xDisplacement, eyey, eyez +
zDisplacement)
= (0.005+1.0, eyey,(-10)+ 0.05)
= (1.005, eyey, -9.95)
```

Now, this method is not without it's drawbacks. The main issue being the bigger the distance between the look from and look to locations the faster the "walk speed" will be. It can be managed though and the cost

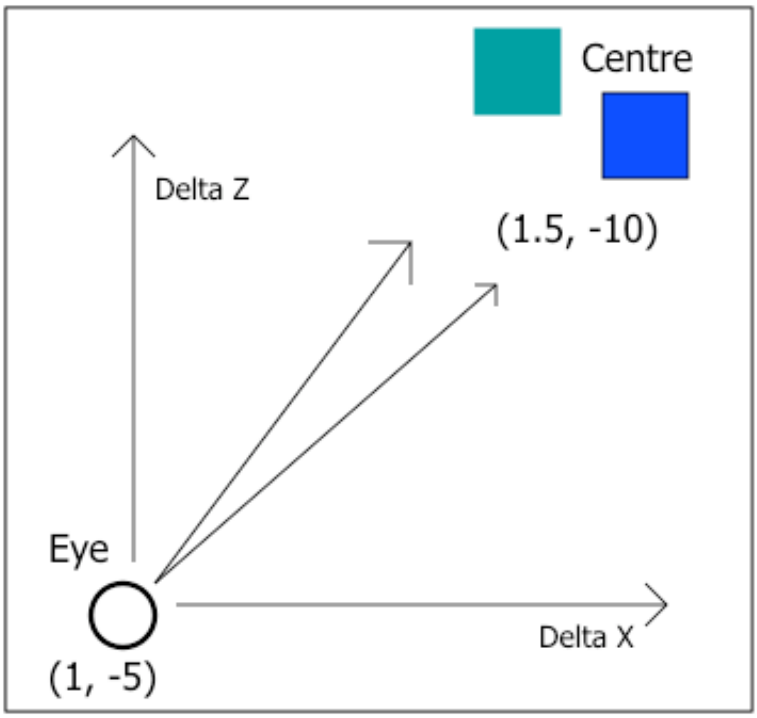
of managing this is less CPU intensive than more many other movement algorithms.

For a world this small, the relative size of the difference between our look from location and our look at location is in reality too large so do experiment with this when you're done. You'll find the relationship between the distance between the two points, and our defined `WALK_SPEED` does make a difference in the apparent walk speed as we move around.

Now, let's look at turning left and right.

It's not uncommon for me to see code where the programmer will dutifully record the angle at which the scene is being rendered from. Not today. We know the angle that we are working with because we know the two points (remember Pythagorus, we have a right angle triangle).

Have a look at the following:



As we want to simulate turning around. We don't actually need to do anything except to move our looking at or target location ***around in a circle***. Our definition for `TURN_SPEED` really is the angle at which we are turning.

Here's the key to understanding what I have just said. You don't make changes to your eye co-ordinates; you change what you're looking at. By plotting a new location (ie the next increment of our angle as

specified by `TURN_SPEED`) on a virtual circle around our eye, you get a new "angle of rotation".

So, if we are just going to define turning as essentially drawing a circle around our centre being our eye or looking from location, then we just need to remember how to draw a circle.

In other words, it's just:

```
newX = eyeX + radius * cos(TURN_SPEED)*deltaX -
                sin(TURN_SPEED)*deltaZ

newZ = eyeZ + radius * sin(TURN_SPEED)* deltaX +
                cos(TURN_SPEED)*deltaZ
```

Handling Touches and Converting these into Movements
OK, now let 's put this into practice.

Back in the implementation, we need to start working out from the touches what to do as far as getting new parameters for **`gluLookAt()`**. First, start off with just the method implementation and some basics:

```
- (void)handleTouches {

    if (currentMovement == MTNone) {
        // We're going nowhere, nothing to do here
        return;
    }
```

First, just check to see if we're moving. If not, there's nothing to do.

Now, no matter if we're moving or turning, we need to know the **`deltaX`** and **`deltaZ`** values. I'm just storing these in a variable called vector:

```
    GLfloat vector[3];

    vector[0] = center[0] - eye[0];
    vector[1] = center[1] - eye[1];
    vector[2] = center[2] - eye[2];
```

I did calculate the Y delta but it's not needed.

Now, we need to work out which movement action we need to do. That's just in a switch statement:

```
    switch (currentMovement) {
        case MTWalkForward:
            eye[0] += vector[0] * WALK_SPEED;
            eye[2] += vector[2] * WALK_SPEED;
            center[0] += vector[0] * WALK_SPEED;
            center[2] += vector[2] * WALK_SPEED;
            break;
```

```
        case MTWAlkBackward:
            eye[0] -= vector[0] * WALK_SPEED;
            eye[2] -= vector[2] * WALK_SPEED;
            center[0] -= vector[0] * WALK_SPEED;
            center[2] -= vector[2] * WALK_SPEED;
            break;

        case MTTurnLeft:
            center[0] = eye[0] + cos(-TURN_SPEED)*vector[0] -
sin(-TURN_SPEED)*vector[2];
            center[2] = eye[2] + sin(-TURN_SPEED)*vector[0] +
cos(-TURN_SPEED)*vector[2];
            break;

        case MTTurnRight:
            center[0] = eye[0] + cos(TURN_SPEED)*vector[0] -
sin(TURN_SPEED)*vector[2];
            center[2] = eye[2] + sin(TURN_SPEED)*vector[0] +
cos(TURN_SPEED)*vector[2];
            break;
    }
}
```

And that's it for the handle touches method. The implementation is just the algorithm we went through earlier.

Bringing it All Together
Head back to the drawView method and add the following line before the call to **`gluLookAt()`**:

```
    [self handleTouches];
```

And that's it, we're done!

Hit "Build and Go" now! Right now!!

Not exactly 6 degrees of freedom but given the above code, you can do it. It's not that much more work. The how to is all here in this tutorial.

OK That's all for Today
That's all for today. As per usual here's the completed code. As always, you can get me on the email if you have any questions. If you've emailed me and I haven't responded, it's been because I've been too sick and will get back to you eventually.

As usual, here's the source code for today's tutorial:

OpenGLES13 Project.zip

I'll ensure the gap between today's tutorial and the next one won't be as long as this last one. Coming up next, we're going to load up walls and floors from a map rather than statically, and move around in a world loaded like a real game level. If you're really lucky, I'll show you

object collision detection as well!

Until next time, take care. Hooroo!
Simon Maurice

previous next