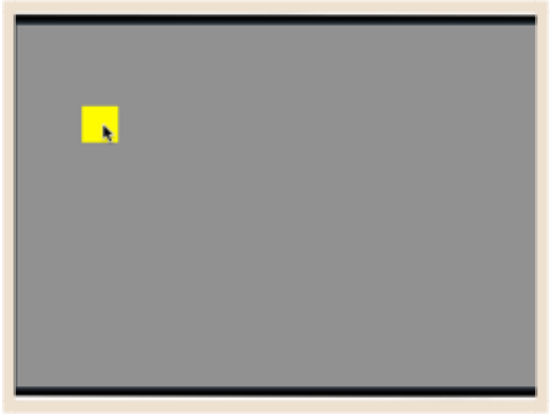
SIMON MAURICE - IPHONE - OPENGL ES



Tuesday, 28 April 2009

OpenGL ES 12 - Landscape View & Handling Touches Part 1 - 2D World

Had a funny conversation with my boss the other day. He's a much more learned OpenGL and general graphics gentleman than I am despite the fact we've both been working professionally in OpenGL for the same length of time roughly. He'd seen these tutorials and joked that "If that's really what I thought about OpenGL, I'd never have hired you!".

The reason for him saying that was the fact that I use terms like "OpenGL will render this like so..." or "send your vertex array to OpenGL". I know, and some others who read these tutorials will also know, that when I refer to OpenGL existing as an entity, its like me telling you that Santa Claus is real!! Whilst I'm not trying to mislead you, I'm just trying to make things simpler and, in reality, teaching the way that I think I would have liked to be taught.

Just as a side note to my conversation with my boss, he said he would find some good modern books to recommend for beginners since he thought the book era was not really dead, and I just think that search is the answer to the world's problems including world hunger, global peace etc etc etc. Anyway he's going to dig something up for me. If they're any good, I'll let you know.

My first exposure to OpenGL was at my first job in Government research. We'd just got budget approval for some frighteningly expensive SGI hardware and

myself and another colleague were sent off for a week of intensive OpenGL introduction before these boxes arrived.

For the first three and a half days of this week, we didn't see a single line of code. It was nothing more than hours and hours talking about the specification, the state machine and all that core information that, while at some stages were very interesting, was fair dinkum boring! I was often just nodding and pretending to understand, and I know my colleague was too. But it wasn't until we were introduced to the code that things started to come together in a small way.

When a brand new SGI Indy arrived on my desk some months later (yes, that was ***my*** computer, not shared!) I really started to learn OpenGL probably in much the same way I now show here. Put things on the screen and then experiment with them. Didn't care if the perspective is out, or things are looking squashed. Didn't worry about being optimising for speed. Just get things happening.

So when you read me saying things that you know are not true but show concept as an oversimplification, it's because for many people out there, I still need them to believe Santa Claus exists rather than explain the long detail about how those presents appeared under the Christmas tree.

Tangent Time...
Anyway, today I'm off on a tangent. I wanted to continue on with the tunnel from the last tutorial where I was going to break that down into a 3D world where we would draw the floor from a map, move around and then build the walls and rooms to explore. I'm going to put that on hold for a minute.

I've started to get quite a few requests for handling touches which, on the surface is quite easy, but there's lots of potential frustration in the detail. Some guys who have been really good to me through this tutorial series with comments and notifying me of errors in my text etc want to know this. So as a general nod and thank you to them (you know who you are), here it comes.

I do need to cover this in two stages though. Working in a 2D world for something like a slide scrolling arcade game like Defender, or a orthographic perspective game like Syndicate, is really different to a 3D game. Whilst touches may seem less important than the accelerometer in a 3D game, you still need a fire button, pause game button etc for user input so the principles still do apply.

I'll deal with 3D next time. In OpenGL (ie not OpenGL ES) it's quite easy to do because we have handy utility functions to take the grunt work out of it which aren't available to us on the iPhone. So that's for next time.

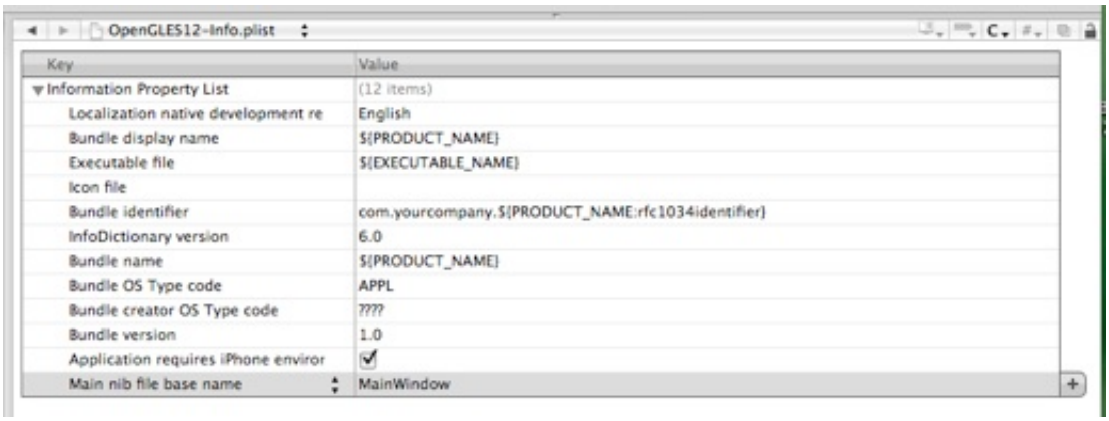
The Starting Point - Going Horizontal
What we're going to do very simply first up is to bring up a 2D display in landscape mode. I'm going to use landscape mode because I think most games

run in this mode and, if you want a portrait interface, I'm sure you'll see the things we need to leave out. So to begin with, we'll go from square one again to show you the steps you need to do to make an app appear in landscape mode.

There's no starting template, so fire up Xcode and create a new project using Apple's OpenGL ES template.

First off, we'll set the device to go straight into landscape mode and get rid of the status bar. Under the "Groups & Files" navigation area, expand the "Resources" folder, and open the app's .plist file (ie mine was called OpenGLES12-info.plist because my project name is OpenGLES12).

You'll get this on the screen:



Click on the last item so that the "+" icon appears on the right hand side and click on the "+" icon to add a new item. In the drop down which appears under the "Key" heading, select "Initial Interface Orientation". TAB across to the "Value" heading and select "Landscape (right home button)".

This will bring our app open in landscape mode with the home button on the right hand side.

Now to get rid of the status bar so we have a full screen, press the "+" button again and select the key: "Status bar is initially hidden", and click the check box under the value field so that a tick appears.

Drawing Something to Fondle

We can't really do a touch event without something on the screen to move around with our finger. So, g=head straight to the `EAGLView.m` file and the `drawView[]` method. We'll delete the square drawing code so make the draw view method look like this:

```
- (void)drawView {

    [EAGLContext setCurrentContext:context];

    glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
    glViewport(0, 0, backingWidth, backingHeight);
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrthof(-1.0f, 1.0f, -1.5f, 1.5f, -1.0f, 1.0f);
    glMatrixMode(GL_MODELVIEW);

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];
}
```

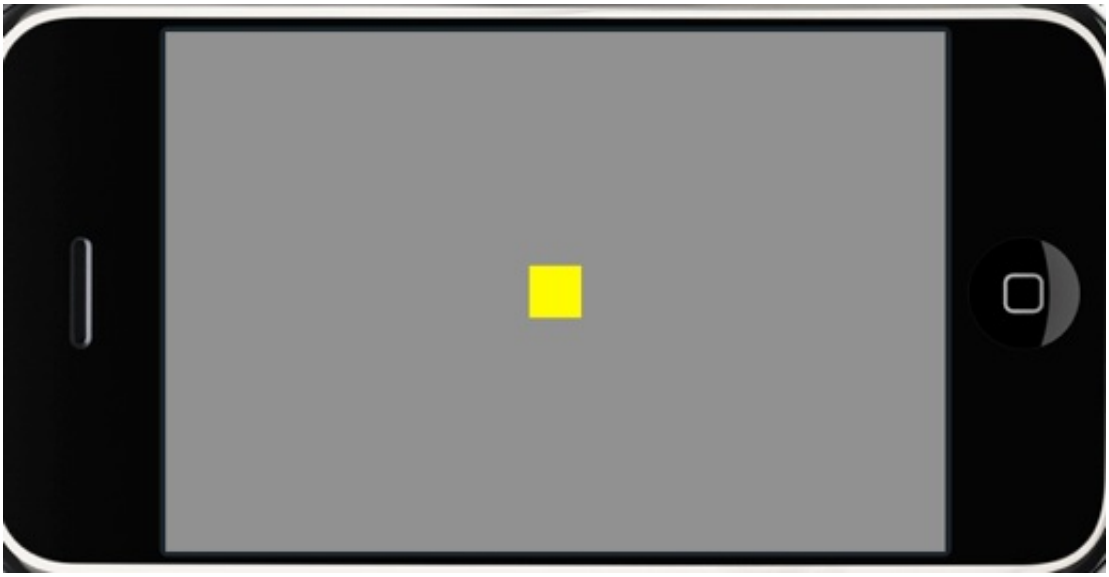
There are two steps to drawing something on screen in landscape mode. First of all, I'll just do the drawing code and then we need to make some adjustments to the view to make things look right in landscape mode. I'll do the drawing code first so you can do a "Build and Go" to make sure everything is working right before we get to the tricky stuff.

We'll draw a point, so first add the following after the call to `glClear()` and before the buffer swap:

```
    const GLfloat pointLocation[] = {
        0.0, 0.0
    };

    glPointSize(32.0);
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glVertexPointer(2, GL_FLOAT, 0, pointLocation);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, 1);
```

Hit "Build and Go" and the simulator should start in landscape mode with the following on screen:



So far so good. All we've done is to draw a point.

Make Sure You Have Some Caffeine...
Now to deal with the biggest source of confusion of working in landscape mode.

Despite the fact that the iPhone knows we are in landscape mode, it won't tell OpenGL this. So right now, if you try and translate this point to a higher Y value (ie towards the top of the display), it will actually move towards the top of the iPhone, towards the earpiece.

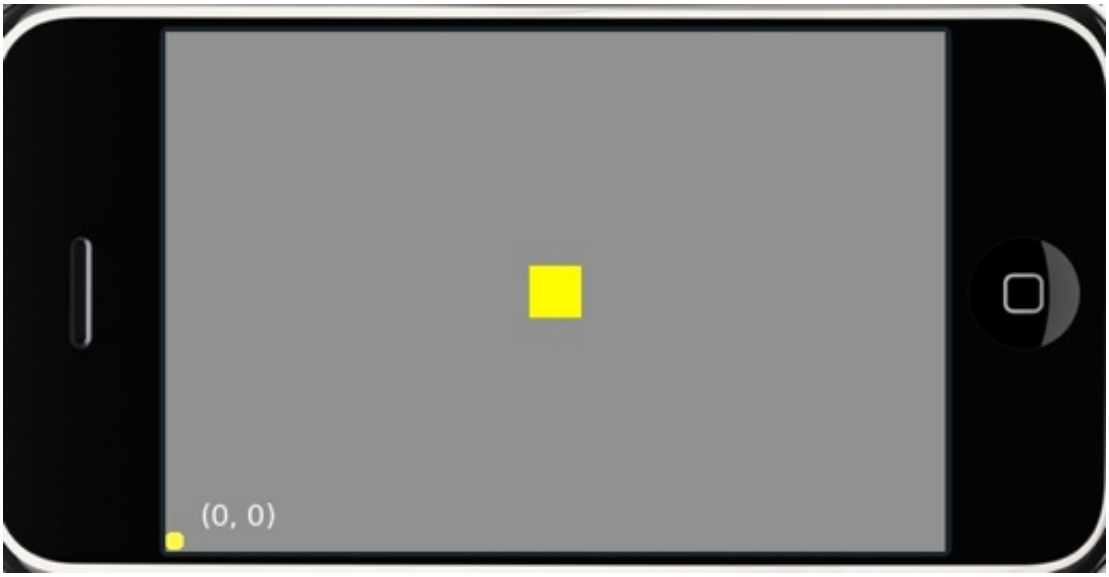
Try it!

There are two ways I can think of to get around this but the easiest way is just to rotate the projection matrix through 90º. The other way is changing everything from below the surface of the `EAGLView` class but I haven't tried going that deep yet and I'm pretty sure that there's a roadblock in there somewhere.

So, in order to make this work, we need rotate our projection matrix. This is achieved as follows:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glRotatef(-90.0, 0.0, 0.0, 1.0);
glOrthof(-1.0, 1.0, 1.5, -1.5, -1.0, 1.0);
glTranslatef(-1.0, -1.5, 0.0);
```

The first line of code after the `glLoadIdentity()` just rotates our projection so that when we specify a change in the X co-ordinate, it does that rather than change the Y-co-ordinate. The second line is just a connivence for us whilst working on a 2D view. Instead of making the centre of the screen (0, 0), we have moved the co-ordinates of (0, 0) (or the origin if you think in those terms) to the bottom left of the screen, ie:



If you wanted the home button on the left, you would rotate through +90º instead of negative.

We're not done yet. We need to ensure that our aspect ratio is right. Currently it's not because we're originally set up for a display that is taller than it is wider (portrait mode). So what we need to do look a bit more closely at our call to

`glOrthof()`.

```
glOrthof(-1.0f, 1.0f, -1.5f, 1.5f, -1.0f, 1.0f);
```

It's the first four parameters that we're interested in. This does two things for us. It tells OpenGL where our objects get clipped from our display and the aspect ratio.

The clipping part is defined by each individual parameter passed to the function. It's the left, right, bottom, top (near and far are the last two, they don't matter in 2D). So, an object with an X co-ordinate of less than -1.0 (ie more negative) is not on the screen. Greater than 1.0 is also not displayed (it's off to the right). Same applies for the Y co-ordinates.

Now, the aspect ratio is the combination of the X pair and the Y pair. You can see that the overall width of our display is 2.0 units (ie abs(-1.0) + abs(1.0) = 2.0) and the overall height is 3.0 units. But since we're in landscape mode we need to swap these around otherwise rendered objects will look "squashed".

So, we can change the `glOrthof()` and the subsequent `glTranslatef()` function to these two to get these two issues fixed:

```
glOrthof(-1.5, 1.5, 1.0, -1.0, -1.0, 1.0);
glTranslatef(-1.5, -1.0, 0.0);
```

That's better. Now you can see that we've also changed the translation to move the origin point of (0, 0) to the bottom left.

Now, I've shown it to you in this way to break it down a bit so as to introduce this concept a bit more slowly. However, we can actually do away with the `glTranslatef()` function simply by changing he parameters to `glOrthof()`. Here's our final set up code:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glRotatef(-90.0, 0.0, 0.0, 1.0);
glOrthof(0.0, 3.0, 0.0, 2.0, -1.0, 1.0);
```

Our width is still 3 units and our height is still 2 units, but we've just specified new clipping boundaries. Note the order of the functions? You cannot do the rotation after the call to `glOrthof()` because you always rotate around (0, 0).

OK, so there we are, in landscape mode and all of our transformations will now work as we expect. Time to move on. But before we can move to touches, let's look at...

World to Screen Co-ordinates

First up, a quick demonstration. Switch to the definition and add the new variable as follows:

```
GLfloat newLocation[2];
```

Now, switch back to the implementation and in the `initWithCoder[]` method, define the variable's initial location as:

```
newLocation[0] = 1.5;
newLocation[1] = 1.0;
```

New `newLocation[0]` is our X co-ordinate and `newLocation[1]` is our Y co-ordinate. Where do you think the point will appear on the screen?

Hold that thought! Let's add some code and find out. In the `drawView[]` method, we'll change the drawing code to:

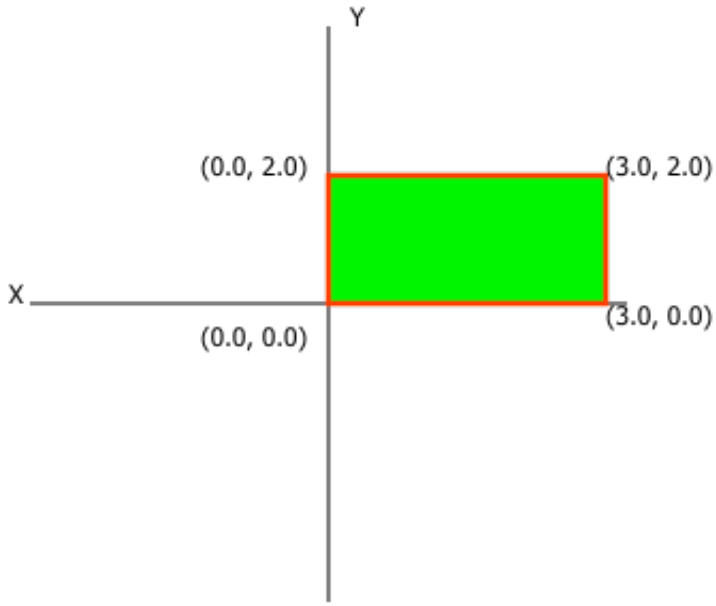
```
glPushMatrix();
glPointSize(32.0);
glColor4f(1.0, 1.0, 0.0, 1.0);
glTranslatef(newLocation[0], newLocation[1], 0.0);
glVertexPointer(2, GL_FLOAT, 0, pointLocation);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS, 0, 1);
glPopMatrix();
```

Apart from pushing and popping the current (object) matrix, we've added a line to move the point to the X and Y co-ordinates held by our variable `newLocation`.

Hit "Build and Go" to see if you were right on the point's final location.

Were you right? The point is now back in the centre of our screen. So what does that tell us about world to screen co-ordinates?

We now know that OpenGL will only display objects which are currently located in our world bound by the rectangle which has an origin of (0, 0) and extends 3.0 world units in the positive X direction and 2.0 world units in the positive Y direction. Thus:



From the diagram above, that's a representation of what we see on our iPhone with these clipping co-ordinates. The green area represents the viewable section of our world.

Therefore we can now map these to screen co-ordinates. I'll show you how to do this below but I'm sure you're already starting to work it out.

Handling Touches
This is not hard to do thankfully. The `EAGLView` class is a subclass of `UIView`, which, in turn, is a subclass of `UIResponder`. The `UIResponder` class defines but does not implement the following 4 methods to handle touches:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
```

`touchesBegan` is fired when the user's finger first touches the screen.

`touchesMoved` occurs with dragging the finger after a touchesBegan[]

`touchesEnded` is fired when the user's finger is lifted

`touchesCancelled` is an even that happens after `touchesBegan` but is interrupted by the system events such as memory warnings or those blue "network lost" messages I frequently seem to get

We do not have access to the `UIControlEvents` such as `TouchUpInside` etc

because we are not using `UIControls`.

Those are the four methods you have access to to receive touches. For starters, and to show how we can translate the screen co-ordinates into world co-ordinates, let's look first at `touchesBegan`.

Add a new method to the `EAGLView` implementation. You don't need to put it in the header as it's already defined in the `UIResponder` super class.

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

}
```

It's in this method that we're adding the new code. First of all, we need to get the co-ordinates of where the touch event occurred. These are obtained as easily as:

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

Please read the API reference if you need to know how I got this information. I'm going to convert these into a "percentage" of the screen. That is, for half way along either axis, instead of getting a co-ordinate, I want 0.5 or 50%. That's as easy as:

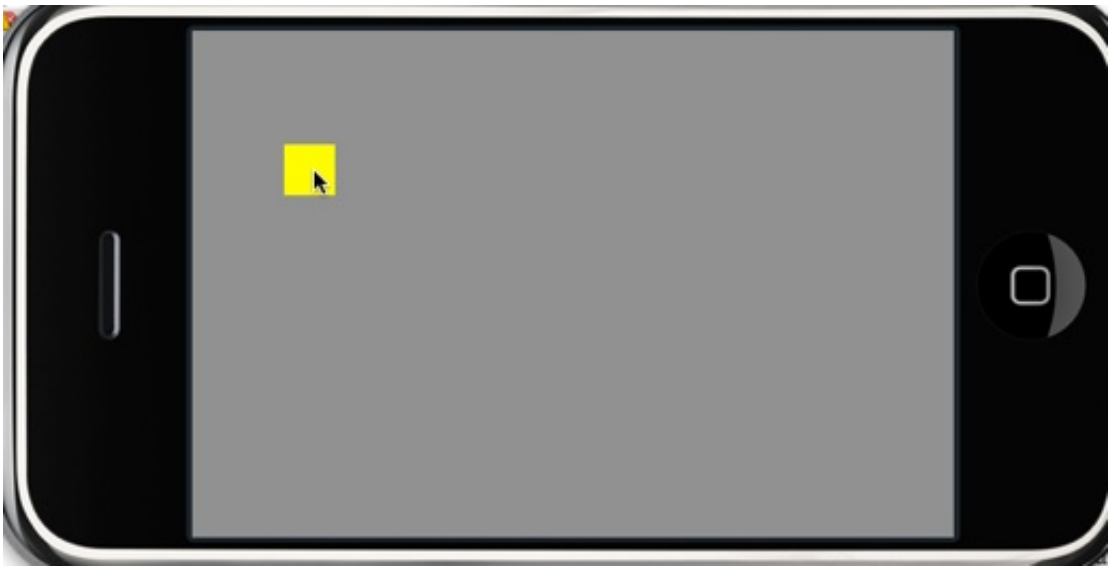
```
CGRect bounds = [self bounds];
CGPoint p = CGPointMake((touchPos.x - bounds.origin.x) /
bounds.size.width,
                       (touchPos.y - bounds.origin.y) /
bounds.size.height);
```

Finally. convert the percentage value to the viewable world co-ordinates. In order to do this, we need to switch the UITouch supplied X and Y co-ordinates. Even though the iPhone knows it's in landscape format, it still supplies the X and Y co-ordinates as though it is in portrait view. So the point made above p.x contains our Y value and vice versa.

```
    newLocation[0] = 3.0 * p.y;
    newLocation[1] = 2.0 * p.x;
```

Remember above that the X width of the viewable area was 3.0 units? So if a touch is 50% of the X value (ie w.x = 0.5), then our world co-ordinate is 3.0 * 0.5 = 240.0. Same applies for the Y value.

Once you've added the above code, hit "Build and Go". Click anywhere on the simulator's screen area and the point will follow the mouse click.



Just so long as you remember, anything supplied to you as an X co-ordinate by UI-anything will need to be treated as a Y co-ordinate. Y co-ordinate from UI-anything is really an X co-ordinate.

OK, so that's fine but we can also make it follow our touches by putting the same code into `touchesMoved[]` like this:

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *t = [[touches allObjects] objectAtIndex:0];
    CGPoint touchPos = [t locationInView:t.view];

    CGRect bounds = [self bounds];

    // This takes our point and makes it into a "percentage" of the
screen
    //   That is 0.85 = 85%
    CGPoint p = CGPointMake((touchPos.x - bounds.origin.x) /
                        bounds.size.width,
                           (touchPos.y - bounds.origin.y) /
                        bounds.size.height);

    newLocation[0] = 3.0 * p.y;
    newLocation[1] = 2.0 * p.x;
}
```

That is exactly the same as the touches began function. If you hit "Build and Go" now, click and drag in the simulator, the point will follow your mouse clicks.

NOTE: At the time of writing this, I've noticed that when you drag to the left, it stops just shy of the screen's edge and won't allow you to drag the point all the way to the edge. Also the simulator stops sending touch events. This is limited to the simulator only based on an Apple technical note. I'll check this when I get a chance and post confirmation here.

[Edit: Updated: Seems to work ok on the device but the problem appears to continue to exist in simulator 3.0. No biggie, just so long as the device works ok]

Finger Co-ordinate to Object Co-ordinate Detection

I know you guys won't let me end this without discussing whether or not the user's finger has actually touched an object or missed. In the 2D world, it's fairly straight forward and we'll make the changes to the `touchesBegan` method.

First of all, we need to have some basic understanding from the user's point of view. A mouse is very easy to click with but a lumpy finger is a bit different. Mobile Safari often thinks I've clicked on one link instead of the one I've clicked on because once your finger gets over the general screen area, you can't see the exact target anymore.

Further, our objects are defined very specifically. (1.445, 1.444) is not (1.444, 1.444) so we have to allow some common sense in the touch detection otherwise you'll be clicking for hours on end trying to get the exact co-ordinate of our point.

So, let's start modifying this so that the `touchesMoved` method only responds if we touched the point itself, and not anywhere else on the screen. First, add a new variable in the definition being:

```
BOOL fingerOnObject;
```

and in the `initWithCoder` method, set this to a value of `NO`:

```
fingerOnObject = NO;
```

Now, in the `touchesBegan` method, delete the last two lines where we assign values to the `newLocation` variable, and replace them with the following:

```
CGRect touchArea = CGRectMake((3.0 * p.y) - 0.1, (2.0 * p.x) - 0.1, 0.2,
0.2);

if ((newLocation[0] > touchArea.origin.x) &&
          (newLocation[0] < (touchArea.origin.x + touchArea.size.width)))
{
        if ((newLocation[1] > touchArea.origin.y) &&
                (newLocation[1] < (touchArea.origin.y +
touchArea.size.height))) {
            fingerOnObject = YES;
        }
    }
```

So what I've done here was to create a rectangle which is larger than the touch area by offsetting both the X and Y co-ordinates by 0.1 units and making it 0.2 units wide and high. Note that you need to make the width and height double the size of the X and Y offset value. Then, all the if statements do is to check that our point is within that touch area.

Then at the start of the `touchesMoved` method, add the following if statement at the very start of the method:

```
    if (!fingerOnObject) {
        return;
```

```
    }
```

So if the `fingerOnObject` variable is not set to `YES`, then we don't care if the user is dragging their finger. Finally, implement the `touchesEnded` method to reset the `fingerOnObject` variable:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    fingerOnObject = NO;
}
```

That's it. Hit "Build and Go" and click on the point to make it move only when you've got within the touch area.

On first drag, it does jump a bit depending on where you clicked but partly, that's ok because the user's finger is in the way and that would be partly lost. However, depending on your touch area target, you can make it smaller or, what I would do would be to move half way in the `touchesBegan` method and let the first call to `touchesMoved` make the second half which, given their finger is in the way would be enough to cover that initial jump.

Detecting Touches Onto A Square
Touching a square is easier than touching a point. You know the origin of the square by the variable `newLocation`. You know what offset to use from the square vertex array, as well as the width and the height. So all you need to do for a square is the comparison.

I've not done the code as I'm typing this on my lunch break and it's nearly time to get back to work!! Given that, I'll call it quits for this tutorial.

Thanks to All Those Who Commented and Helped in this Series
So, to the guys who gave me lots of feedback and helped me fix errors in the text and code from this series and who requested this topic, its a thank you to you all. Hope this answered at least some of your questions and got you going in the right direction.

As usual, the download link for the code:

OpenGLES12.zip

This was a bit of fun for me actually and I'll continue this again adding some more of the features you'd want for a 2D game as I know that's what a lot of you guys are writing.

Until the next 2D or 3D instalment, hooroo!
Simon Maurice

previous next

Made on a Mac

Email Me