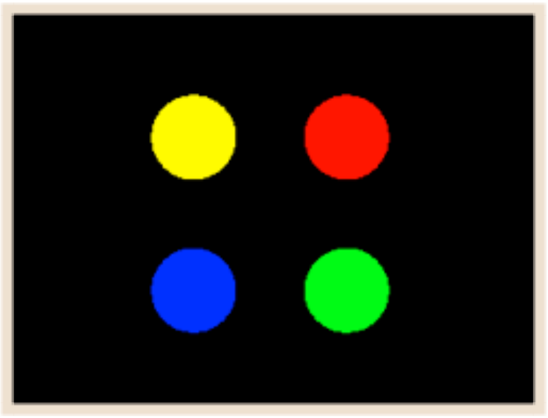
SIMON MAURICE - IPHONE - OPENGL ES



Monday, 20 April 2009

OpenGL ES 08 - The Final Primitives: Points and Lines in a Stride

OK, let's get a real quick tutorial out of the way. The final primitives which we have not covered is points and lines. They're so easy but, after answering a few questions on them already, I thought I'd better cover them.

I am going to go through these two at velocity because I think once they're introduced, you'll pretty much think: "that makes sense" and be ready to move on. That's pretty much the goal.

However, don't skim this too quickly. I'm going to use that cryptic "stride" parameter in the `glVertexPointer()` call. I'm going to cover this now as I am writing a tutorial to use Blender objects in our iPhone code so we can get a stupidly large quantity of objects on the screen without too much work.

Stride is worth covering as most people simply skim over it (as I have done so far) or state things like "it's a really advanced topic and you're not ready yet" (probably because they haven't got it to work yet...). It's got a real gotcha but once you're aware of it, it's easy.

Points

In OpenGL ES, a point is a dot in 3D space. It will be rendered as a circle if you have enabled `GL_POINT_SMOOTH`, otherwise will be rendered as a square.

Now you could actually render a quick and dirty point using two triangles rather than using a point but I don't recommend it.

In fact, I've just finished a discussion with someone who stated that he ***always*** drew his points using two triangles, after all, "that is the way the hardware implements it". While I won't say that this approach is specifically wrong, but really isn't right either.

The argument was that the underlying hardware is implementing the point drawing using 2 triangles to form a square (true in many cases) so by sending the hardware GL point data, it's adding to the hardware's workload. However that ignores such basic questions such as: What is more efficient on the hardware: `GL_TRIANGLE_STRIP` or `GL_TRIANGLE_FAN`?

In short, don't second guess the hardware. In our case on the iPhone, we could write our code to maximise the throughput of the OpenGL ES chip, but we probably don't need to.

Anyway, let's get on with drawing a point. We're going to render four points on the display, in four different colours. To define a point, all you need is an X, Y, and Z co-ordinate, and a size.

First, download our basic project: AppleCoder-OpenGLES-00.zip

To draw four points, we just specify the point co-ordinates in a vertex array, and our colours in a colour array. So, in the `drawView[]` method, we need to add the following structure:

```
    const GLfloat points[] = {
        1.0, 1.0, -6.0,         1.0, 0.0, 0.0, 1.0, // First
point, Red
        1.0, -1.0, -6.0,        0.0, 1.0, 0.0, 1.0, // Second
point, Green
        -1.0, -1.0, -6.0,       0.0, 0.0, 1.0, 1.0, // Third
point, Blue
        -1.0, 1.0, -6.0,        1.0, 1.0, 0.0, 1.0  // Fourth
point, Yellow
    };
```

I hope you can see straight away that all I've done is to combine the vertex array (our co-ordinates) and our colour array (RGBA values) into a single array. No real great dramas?

The drawing code will look almost the same and there's only one line I will go into detail about. There are a couple of steps which we want to follow in order to draw our points but all will be familiar except for setting the point size.

Here's the drawing code for these four points:

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPointSize(50.0);
    glVertexPointer(3, GL_FLOAT, 28, points);
    glColorPointer(4, GL_FLOAT, 28, &points[3]);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, 4);
```

`glPointSize()` sets the size of the point (obviously). However, we don't definitely need this as OpenGL ES has a default point size which it will render if you don't set it. I've just chosen a nice large value.

The next two lines are a departure from what we normally do:

```
    glVertexPointer(3, GL_FLOAT, 28, points);
    glColorPointer(4, GL_FLOAT, 28, &points[3]);
```

Now, I have covered these before but the third parameter, stride, has always been zero, and the pointer to the data has always just referred to the `arrayName[0]` element. Now, since we've stored our colour data in our vertex array, we need to tell OpenGL to ignore the colours for locating each point, and also ignore each point's location when looking up the colour.

That's where stride comes into play (parameter 3 is stride). From the man page, the value for stride is defined as:

```
Specifies the byte offset between consecutive vertices.
If stride is 0, the vertices are understood to be tightly
packed in the array. The initial value is 0.
```

This is the gotcha I mentioned earlier and the single most problem caused by people using stride. Read this carefully:

The stride is the offset between consecutive vertices, not the padding between each co-ordinate!

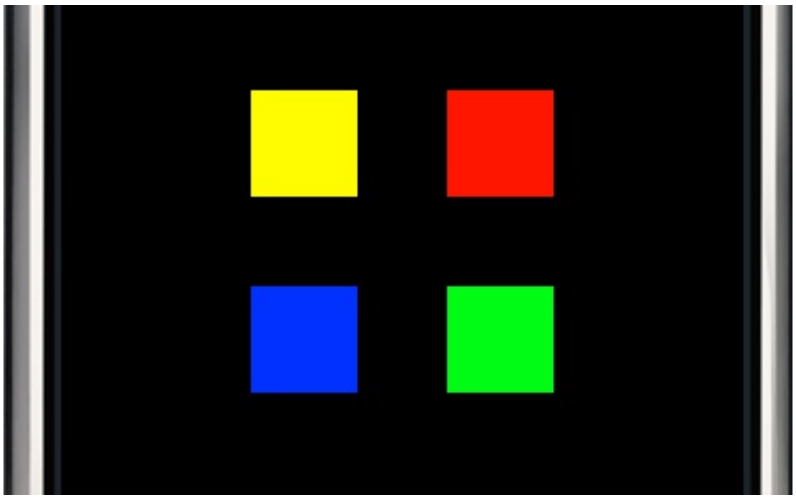
So, if you were to look at the result of: `sizeof(GLfloat)`, you'd see that it's 4 bytes. Therefore, with 3 `GLfloat's` for each vertex, and 4 `GLfloat's` for each colour, each point's definition (location and colour) occupies 28 bytes. That is our stride value!

So, it's the difference between `points[0]` and `points[7]`. Can you see that `points[7]` represents the start of the next vertex co-ordinates? See also that it is stored 28 bytes further into the array from the previous vertex start?

That's stride, in a nutshell. You're going to feel like a guru answering a million questions of people who can't seem to get this working.

Finally, we need to tell OpenGL that the colour array begins at: `points[3]` which is the first value for the RGBA colour (OpenGL colour's processor cannot second guess your structure so you can't just point it at points[0] expect it to work things out with the vertex processor to know where to start). Stride remains the same.

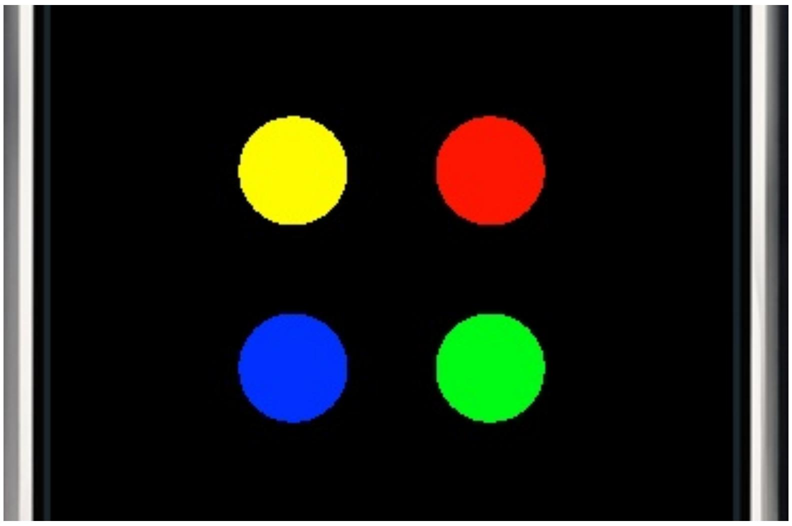
OK, so hit Build and Go and you'll get the following in the simulator:



There we have it, four 50.0 sized points which haven't been smoothed, so they're squares. What to make them round? It's easy; just one line of code:

```
    glEnable(GL_POINT_SMOOTH);
```

Add this line right before `glPointSize()`. Run it again and you'll have smoothed (round) points:



Have a play around with the point sizes, change the colours, and generally experiment with these. To start you off, put the following line:

```
    glRotatef(0.5, 0.0, 0.0, 1.0);
```

right before the `glEnable(GL_POINT_SMOOTH)` we just added. Then start to use `glTranslatef()` to change their Z location. Keep experimenting until you know this cold.

Lines
Lines are just as easy as anything else we've covered: it's only mysterious until you've been shown the way. Let's start by adding the following code:

```
    const GLfloat lines[] = {
        2.0, 2.0, -6.0,     1.0, 1.0, 0.0, 1.0, // Starting point
in yellow
        2.0, -2.0, -6.0,    0.0, 1.0, 1.0, 1.0, // second point
in aqua
        -2.0, -2.0, -6.0,   1.0, 0.0, 0.0, 1.0, // third point in
red
        -2.0, 2.0, -6.0,    0.0, 0.0, 1.0, 1.0  // fourth point
in blue
    };
```

All straight forward after the rendering the points above.

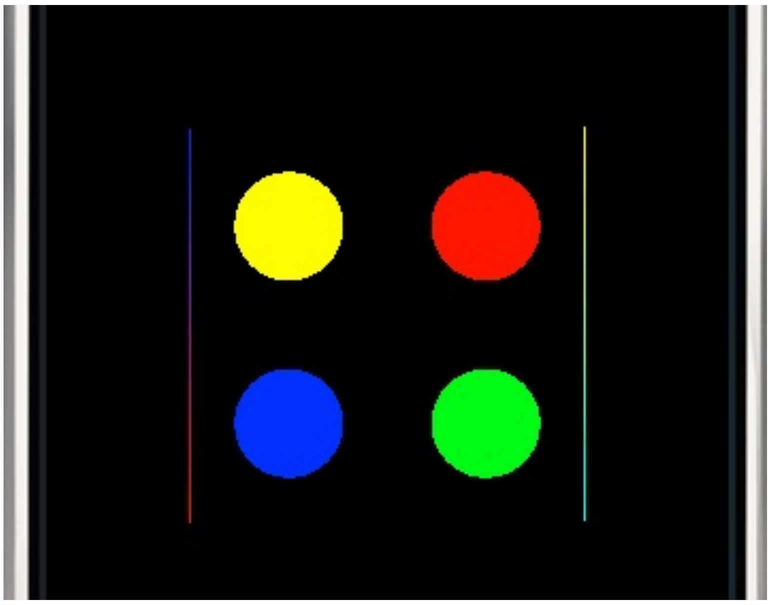
Now, to draw the lines, let's have a look at the code (from `drawView[]`):

```
    // Setup and render the points
    glEnable(GL_POINT_SMOOTH);
    glPointSize(50.0);
    glVertexPointer(3, GL_FLOAT, 28, points);
    glColorPointer(4, GL_FLOAT, 28, &points[3]);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, 4);

    // Setup and render the lines
    glVertexPointer(3, GL_FLOAT, 28, lines);
    glColorPointer(4, GL_FLOAT, 28, &lines[3]);
    glDrawArrays(GL_LINES, 0, 4);
```

After setting up and drawing the points, you can see that all we need to do is to tell OpenGL about the new array for the lines' vertices and colours, then tell OpenGL to draw it with a call to `glDrawArrays()`.

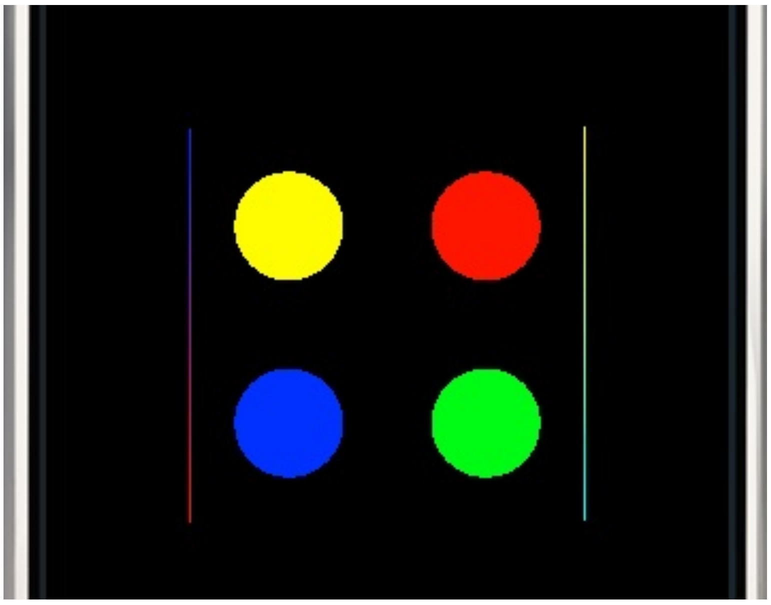
Hit Build and Go and you'll get the following:



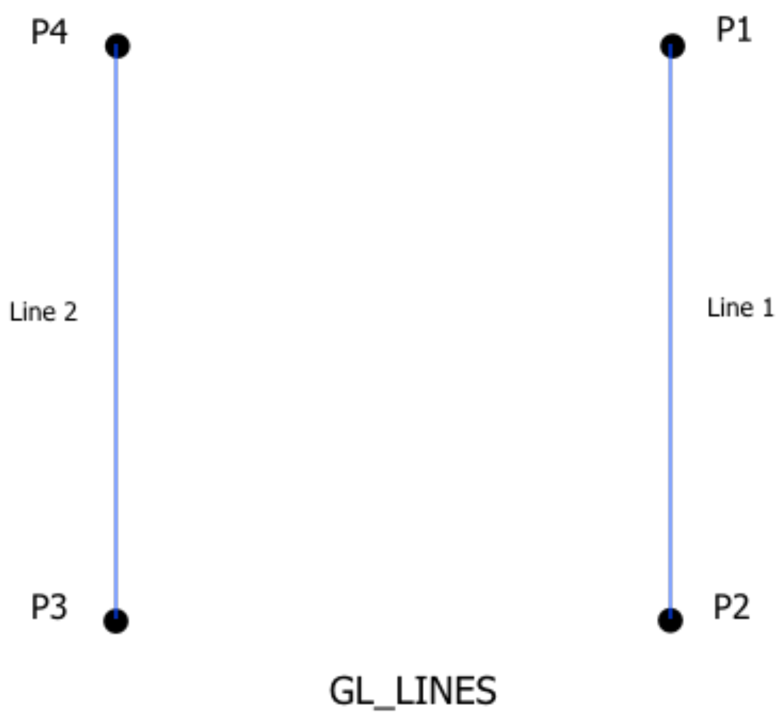
Seems pretty straight forward doesn't it? 4 points in the array so two lines, each with a starting and ending location. Yes, well, that's exactly what we've got. However, I can't get away with leaving it there. Have a look at the `glDrawArrays()` first parameter. It's simply `GL_LINES`. Like drawing triangles, there are a couple of methodologies for drawing lines.

Here they are described.

`GL_LINES` - Each vertex pair is used to make a line.



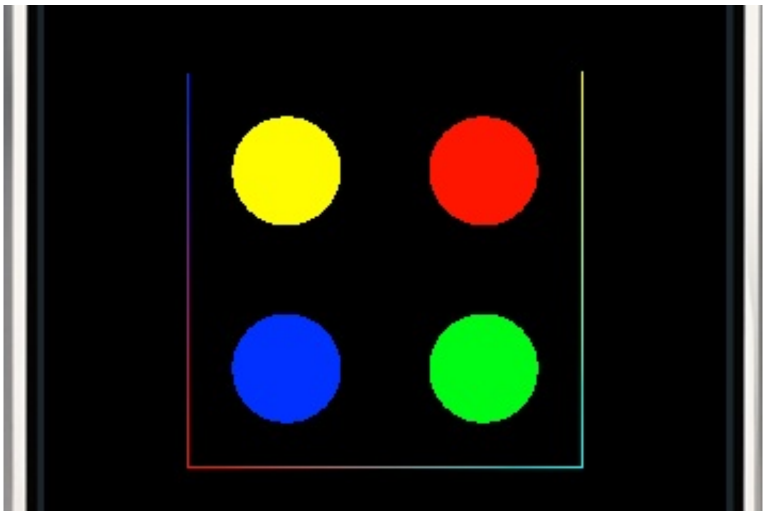
This is how OpenGL looks at it:



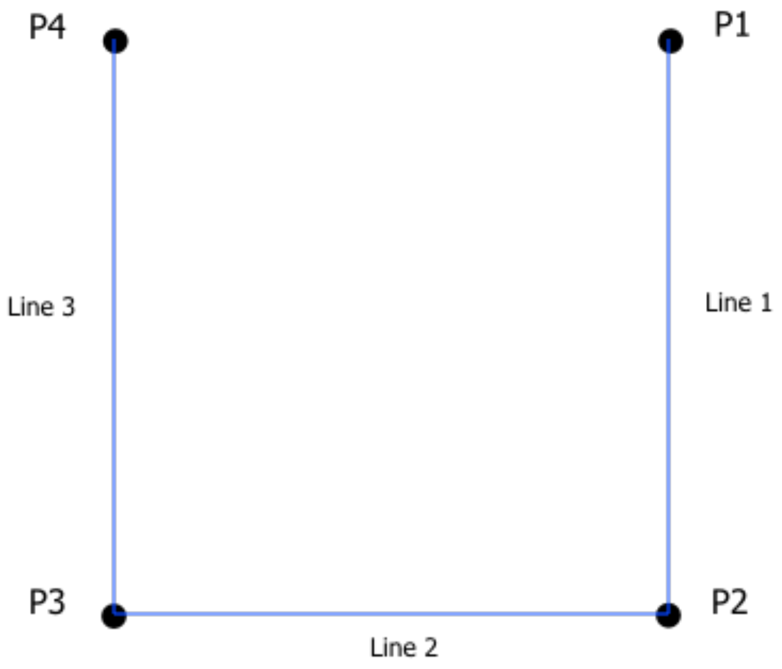
GL_LINE_STRIP - The first vertex specifies the starting point, then each successive vertex is the source point for a line which is rendered between each and the previous vertex.

```
    glDrawArrays(GL_LINE_STRIP, 0, 4);
```

Here’s how our project looks:



and this is how OpenGL looks at it:



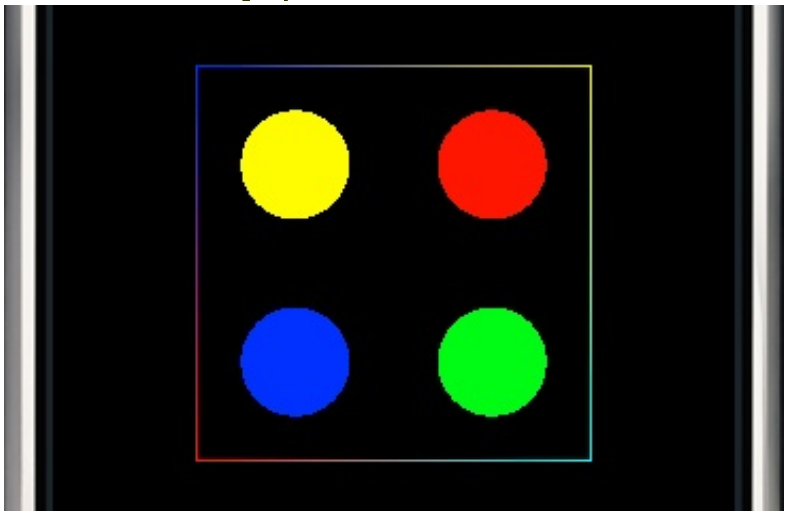
GL_LINE_STRIP

`GL_LINE_LOOP` - The first vertex specifies the starting point, then each successive vertex is the source point for a line which is rendered between each and the previous vertex. The final vertex is then connected to the starting point.

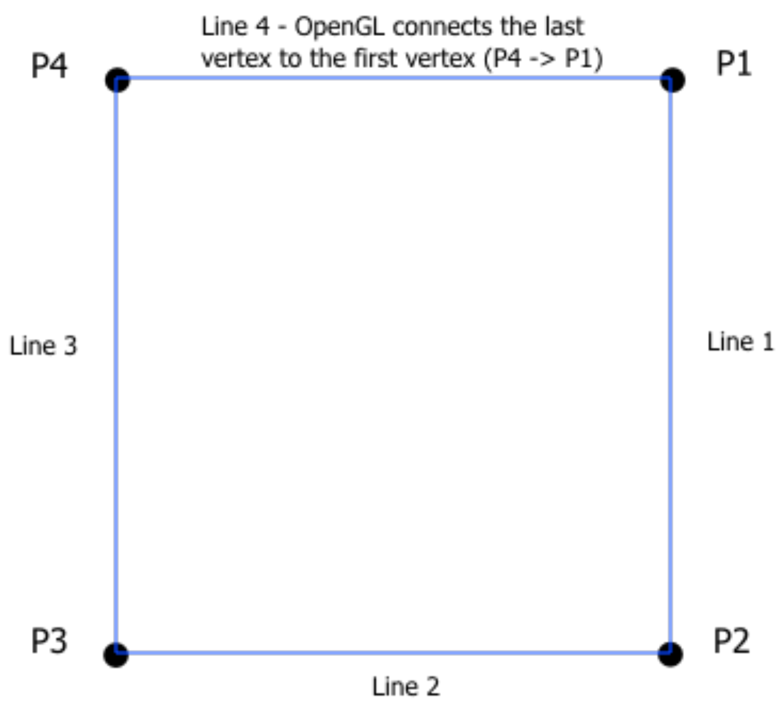
So our code looks like:

```
    glDrawArrays(GL_LINE_LOOP, 0, 4);
```

Our simulator displays:



And from OpenGL's perspective.


Line 4 - OpenGL connects the last vertex to the first vertex (P4 -> P1)
P4
P1
Line 3
Line 1
P3
P2
Line 2

GL_LINE_LOOP

Note that we don't need to stop (or start) at 4 lines, there can be any number of lines but obvious we need at least 2 co-ordinates to form a line.

And that's lines in a nutshell. To make thicker lines, you can experiment with `glLineWidth(GLfloat lineWidth)` calls to make the lines thicker if you like. Note that you may need to adjust start and end co-ordinates to make a "true" square with changes in line thickness.

Also, you can try turning on `GL_LINE_SMOOTH` to do some basic antialiasing of the lines with this call:

```
glEnable(GL_LINE_SMOOTH);
```

It won't make a big difference for a variety of reasons. I'll treat antialiasing as a separate topic in it's own right later on in this series.

Lines and points don't just need to be coloured, they can be texture mapped as well which can make some really good special effects.

Conclusion of another tutorial...
Well, that was a lot longer than I expected but, once again, I hope you've learnt something. Even if it's just the information on stride value for `glVertexArray();` trust me it won't be long before someone comes asking how to make it work!! I'll be getting into some more complex tutorials again next so keep an eye out.

Here's the code for this tutorial:

AppleCoder-OpenGLES-08.zip

Thanks to all who've pointed out the couple of errors that crept in to the web version of the tutorials, the source code, of course, has been fine, just a few errors on the web pages.

Also, all those who've sent me emails of encouragement and support of this series. It's those emails that keeps everything rocking here.

The home of the tutorials is in the "Tutorials" section of the iphonedevsdk.com forums. Check out the thread there.

Until next time, hooroo!
Simon Maurice

previous next