Learn OpenGL ES

Learn how to develop mobile graphics using OpenGL ES 2

Tag: perspective divide

Understanding OpenGL's Matrices

I often get questions related to OpenGL's matrices: how do they work, how do they get built, and so forth. This is a topic that I have been frequently confused by, myself, and I feel that it warrants further explanation.

To better understand OpenGL's matrices, and how and why we use them, we first need to understand the OpenGL coordinate space.

Normalized device coordinates

At the heart of things, OpenGL 2.0 doesn't really know anything about your coordinate space or about the matrices that you're using. OpenGL only requires that when all of your transformations are done, things should be in *normalized device coordinates*.

These coordinates range from -1 to +1 on each axis, regardless of the shape or size of the actual screen. The bottom left corner will be at (-1, -1), and the top right corner will be at (1, 1). OpenGL will then map these coordinates onto the viewport that was configured with *glViewport*. The underlying operating system's window manager will then map that viewport to the appropriate place

on the screen.

Adjusting to the screen's aspect ratio

While OpenGL wants things to be in normalized device coordinates, it's hard to work with these directly. The first problem is that they always range from -1 to +1, so if you use these coordinates directly, your image might be stretched when switching from portrait mode to landscape mode.

The first thing you can do to get around this problem is to define an *orthographic projection*. Android has the orthoM method; other platforms will have something similar. Let's take a closer look at Android's method:

```
orthoM(float[] m, int mOffset, float left, float right, float bottom, float top, float near, float far)
```

To define a simple matrix that adjusts things for the screen's aspect ratio, we might call *orthoM* as follows:

```
float aspectRatio = (float) width / (float) height;
orthoM(projectionMatrix, 0, -aspectRatio, aspectRatio, -1, 1, -1, 1);
```

Let's say that the screen dimensions are 800×600. The call would proceed as follows:

```
orthoM(projectionMatrix, 0, -1.333, 1.333, -1, 1, -1, 1);
```

Although the screen is wider than it's tall, we automatically adjust the coordinate space to match by mapping -(800/600) to the left side and (800/600) to the right side.

This also works when we switch to portrait mode:

```
orthoM(projectionMatrix, 0, -0.75, 0.75, -1, 1, -1, 1);
```

We shrink the width in order to compensate for the smaller screen.

At the heart of things, the orthographic projection matrix will still convert things to the [-1, 1] range, since that's what OpenGL expects. It just provides a way to adjust our coordinate space, so that we can see more of our scene if the screen is wider, and less if the screen is narrower.

3D projections

What about 3D projections? For those, we can use *frustumM*:

```
frustumM(float[] m, int offset, float left, float right, float bottom, float top, float near, float far)
```

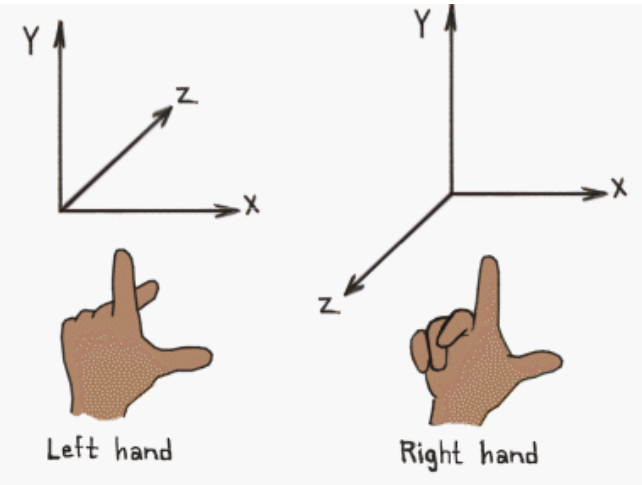
We could define a simple 3D projection as follows:

```
frustumM(projectionMatrix, 0, -aspectRatio, aspectRatio, -1, 1, 1, 100);
```

The near & far range are handled differently: both have to be positive, and far has to be greater than near. We also have to watch out for the Z axis: frustumM will actually *invert* it, so that the negative Z points into the distance!

This has to do with convention: normalized device coordinates are in a left-handed coordinate system, while by convention, when we use a projection matrix, we work in a right-handed coordinate system.

Below is a good image illustrating the situation:



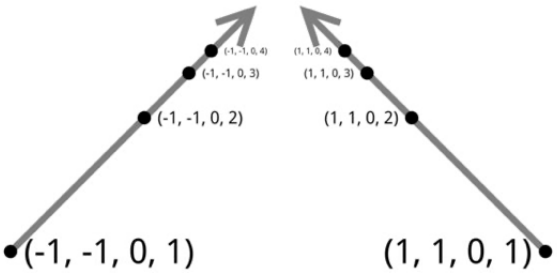
Left-handed and right-handed coordinate systems. Source: http://viz.aset.psu.edu/gho/sem_notes/3d_fundamentals/html/3d_coordinates.html

The perspective divide

The perspective projection doesn't actually create the 3D effect; for that, we need to do something called the *perspective divide*. Each coordinate in OpenGL actually has four components, X, Y, Z, and W. The projection matrix sets things up so that after multiplying with the projection matrix, each coordinate's W will increase the further away the object is. OpenGL will then divide by w: X, Y, Z will be divided by W. The further away something is, the more it will be pulled towards the center of the screen.

This PDF goes into more detail: http://www.terathon.com/gdc07_lengyel.pdf

This image shows how the same coordinate gets closer to the center of the screen as the W value increases:



Perspective divide by W

Say you have three XYZ source positions of the following:

```
(3, 3,    -3)
(3, 3,    -6)
(3, 3, -1000)
```

The second point is a little bit further, or more "into the screen" than the first, and the third point is much further away than the second point. An infinite projection matrix would convert the coordinates as follows:

```
(3, 3,   1,    3)
(3, 3,   4,    6)
(3, 3, 998, 1000)
```

That last component is W. Now, OpenGL will divide everything by W, so you get something like this:

```
(1,     1,     0.33...)
(0.5,   0.5,   0.66...)
(0.003, 0.003, 0.998  )
```

There are two side effects of this division by W:

1. The depth buffer becomes non-linear. There is a lot of Z precision up close, but less further away.
2. If you try to do translations, etc... on your vertices *after* the perspective projection, then you won't get the results you expect. This is because many of these operations depend on the W being 1, while after perspective projection it can be something else.

Here is an example of the perspective divide: imagine that we have a perspective projection matrix that looks as follows:

```
1.5, 0,  0,    0,
0,   1,  0,    0,
0,   0, -1.2, -2.2,
0,   0, -1,    0
```

This will transform coordinates as follows:

```
(1, 1, -1, 1) --> (1.5, 1, -1,   1)
(1, 1, -2, 1) --> (1.5, 1,  0.2, 2)
(2, 2, -2, 1) --> (3,   2,  0.2, 2)
```

After division by W, we get this:

```
(1.5, 1, -1,   1) --> (1.5,  1,   -1  )
(1.5, 1,  0.2, 2) --> (0.75, 0.5,  0.1)
(3,   2,  0.2, 2) --> (1.5,  1,    0.1)
```

Notice that the projection matrix just sets up the W, and it's the actual divide by OpenGL that does the perspective effect. To verify this, try it out with a matrix calculator:

http://www.math.ubc.ca/~israel/applet/mcalc/matcalc.html

The view and model matrices

In OpenGL, we commonly use two additional matrices: the view and model matrices:

THE MODEL MATRIX

This matrix is used to move a model somewhere in the world. For example, let's say we have a car model, and it's defined such that it is centered around (0, 0, 0). We can place one car at (5, 5, 5) by setting up a model matrix that translates everything by (5, 5, 5), and drawing the car model with this matrix. We can then easily add a second car to the scene by just adjusting the translation. The model matrix helps us to push stuff out into the world.

THE VIEW MATRIX

The view matrix is functionally equivalent to a camera. It does the same thing as a model matrix, but it applies the same transformations equally to every object in the scene. Moving the whole world 5 units towards us is the same as if

we had walked 5 units forwards.

Order of operations

These matrices all have to be multiplied in a specific way, if we want our results to be correct. Let's start with some basic definitions:

$vertex_{model}$
This is an original vertex, as defined inside one of our object models.

$vertex_{world}$
This is a vertex in world coordinates. We get to here by using a model matrix to push the model out into the world.

$vertex_{eye}$
This is a vertex in eye coordinates. We get here by using a view matrix to move the entire scene around.

$vertex_{clip}$
This is a vertex in clip coordinates (also known as homogeneous coordinates): this is the coordinate space after projection, but before the perspective divide.

$vertex_{ndc}$
This is a vertex in normalized device coordinates, and this is what we end up with after the perspective divide.

As we can see, getting to $vertex_{ndc}$ is just a matter of applying each transformation in order. Let's try to formulate this as an expression:

$vertex_{ndc}$ = PerspectiveDivide(ProjectionMatrix * $vertex_{eye}$)
$vertex_{ndc}$ = PerspectiveDivide(ProjectionMatrix * ViewMatrix * $vertex_{world}$)
$vertex_{ndc}$ = PerspectiveDivide(ProjectionMatrix * ViewMatrix * ModelMatrix *

$vertex_{model}$)

OpenGL takes care of the perspective division for us, so we don't actually need to worry about that. All we need to worry about is the order of operations; since matrix multiplication is non commutative, we'll get a different result depending on the order.

Column-major versus row-major order

A final point of confusion is often the layout of matrices in memory. OpenGL follows *column-major order*, meaning that the array offsets are specified like this:

```
0  4   8  12
1  5   9  13
2  6  10  14
3  7  11  15
```

m[0] ... m[3] refer to the first *column* of the matrix.

Please let me know your questions, comments, and feedback!

October 3, 2012 / Android, Articles / aspect ratio, Cartesian coordinate system, infinite projection matrix, left-handed coordinate system, model matrix, order of operations, orthographic projection, perspective divide, perspective projection, projection matrix, right-handed coordinate system, Transformation matrix, view matrix / 22 Comments