Extracted from:

OpenGL ES 2 for Android

A Quick-Start Guide

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

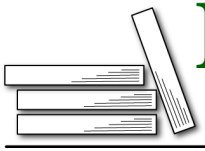
OpenGL ES 2 for Android

A Quick-Start Guide

Kevin Brothaler

Foreword by Mario Zechner, creator of libgdx, author of *Beginning Android Games*

Edited by Susannah Davidson Pfalzer



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The unit circle image in Figure 43, from *http://en.wikipedia.org/wiki/File:Unit_circle.svg*, is used according to the terms described in the Creative Commons Attribution-ShareAlike license, located at *http://creativecommons.org/licenses/by-sa/3.0/legalcode*.

Day skybox and night skybox courtesy of Jockum Skoglund, also known as hipshot, hipshot@zfight.com,*http://www.zfight.com*.

The image of the trace capture button is created and shared by the Android Open Source Project and is used according to terms described in the Creative Commons 2.5 Attribution License.

Printed in the United States of America.
ISBN-13: 978-1-937785-34-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2013

Imagine that you're at an arcade, standing in front of an air hockey table and looking over the table toward your opponent. What would the table look like from your perspective? Your end of the table would appear larger, and you would be looking down at the table from an angle, rather than from directly overhead. After all, nobody plays a game of air hockey by standing on top of the table and looking straight down.

OpenGL is great at rendering things in 2D, but it really begins to shine when we add a third dimension to the mix. In this chapter, we'll learn how to enter the third dimension so we can get that visual feel of staring down our opponent from across the table.

Here's our game plan for the chapter:

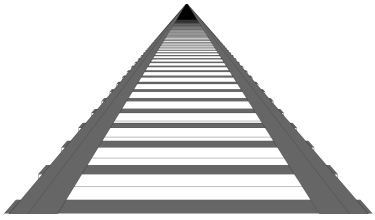
- First we'll learn about OpenGL's *perspective division* and how to use the w component to create the illusion of 3D on a 2D screen.
- Once we understand the w component, we'll learn how to set up a perspective projection so that we can see the table in 3D.

Let's start off by copying the project from last chapter over into a new project, called 'AirHockey3D'.

6.1 The Art of 3D

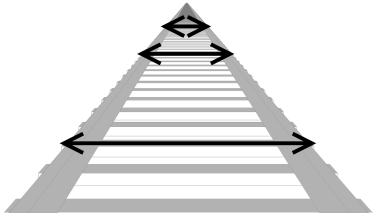
For centuries, artists have been fooling people into perceiving a flat two-dimensional painting as a complete three-dimensional scene. One of the tricks they use is called *linear projection*, and it works by joining together parallel lines at an imaginary vanishing point to create the illusion of perspective.

We can see a classic example of this effect when standing on a straight pair of railway tracks; as we look at the rails receding into the distance, they appear to get closer together, until they seem to vanish at a single point on the horizon:



The railroad ties also appear to get smaller as they get further away from us. If we measured the apparent size of each railroad tie, their measured size would decrease in proportion to their distance from our eyes.

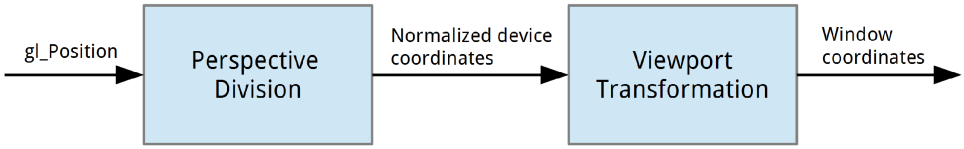
In the following image, observe how the measured width of each railroad tie decreases with distance:



This trick is all that is needed to create a realistic 3D projection; let's go ahead and learn how OpenGL actually does it.

6.2 Transforming a Coordinate from the Shader to the Screen

We are now familiar with normalized device coordinates, and we know that in order for a vertex to display on the screen, its x, y, and z components all need to be in the range of [-1, 1]. Let's take a look at the following flow chart to review how a coordinate gets transformed from the original gl_Position written by the vertex shader to the final coordinate onscreen:



There are two transformation steps and three different coordinate spaces.

Clip Space

When the vertex shader writes a value out to gl_Position, OpenGL expects this position to be in *clip space*. The logic behind clip space is very simple: for any given position, the x, y, and z components all need to be between $-w$ and w for that position. For example, if a position's w is 1, then the x, y, and z components all need to be between -1 and 1. Anything outside this range will not be visible on the screen.

The reason why it depends on the position's w will be apparent once we learn about perspective division.

Perspective Division

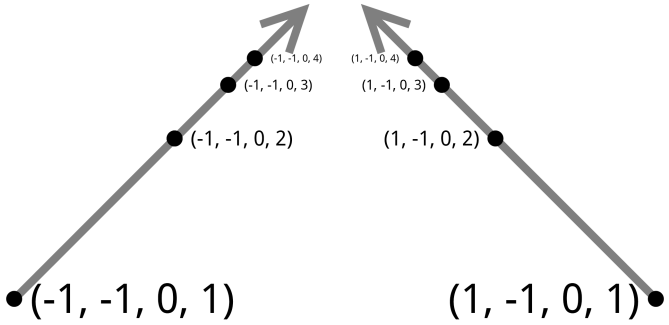
Before a vertex position becomes a normalized device coordinate, OpenGL actually performs an extra step known as *perspective division*. After perspective

division, positions will be in normalized device coordinates, in which every visible coordinate will lie in the range of [-1, 1] for the *x*, *y*, and *z* components, regardless of the size or shape of the rendering area.

To create the illusion of 3D on the screen, OpenGL will take each gl_Position and divide the *x*, *y*, and *z* components by the *w* component. When the *w* component is used to represent distance, this causes objects that are further away to be moved closer to the center of the rendering area, which then acts like a vanishing point. This is how OpenGL fools us into seeing a scene in 3D, using the same trick that artists have been using for centuries.

For example, let's say that we have an object with two vertices, each at the same location in 3D space, with the same *x*, *y*, and *z* components, but with different *w* components. Let's say these two coordinates are (1, 1, 1, 1) and (1, 1, 1, 2). Before OpenGL uses these as normalized device coordinates, it will do a perspective divide and divide the first three components by *w*; each coordinate will be divided as follows: (1/1, 1/1, 1/1) and (1/2, 1/2, 1/2). After this division, the normalized device coordinates will be (1, 1, 1) and (0.5, 0.5, 0.5). The coordinate with the larger *w* was moved closer to (0, 0, 0), the center of the rendering area in normalized device coordinates.

In the following image, we can see an example of this effect in action, as a coordinate with the same *x*, *y*, and *z* will be brought ever closer to the center as the *w* value increases:



In OpenGL, the 3D effect is linear and done along straight lines. In real life, things are more complicated (imagine a fish-eye lens), but this sort of linear projection is a reasonable approximation.

Homogenous Coordinates

Because of the perspective division, coordinates in clip space are often referred to as *homogenous coordinates*,[1] introduced by August Ferdinand Möbius in 1827. The reason they are called *homogenous* is because several coordinates in clip space can map to the same point. For example, take the following points:

(1, 1, 1, 1), (2, 2, 2, 2), (3, 3, 3, 3), (4, 4, 4, 4), (5, 5, 5, 5)

After perspective division, all of these points will map to (1, 1, 1) in normalized device coordinates.

The Advantages of Dividing by *W*

You might be wondering why we don't simply divide by *z* instead. After all, if we interpret *z* as the distance and had two coordinates, (1, 1, 1) and (1, 1, 2), we could then divide by *z* to get two normalized coordinates of (1, 1) and (0.5, 0.5).

While this can work, there are additional advantages to adding *w* as a fourth component. We can decouple the perspective effect from the actual *z* coordinate, so we can switch between an orthographic and a perspective projection. There's also a benefit to preserving the *z* component as a depth buffer, which we'll cover in *Removing Hidden Surfaces with the Depth Buffer*, on page ?.

Viewport Transformation

Before we can see the final result, OpenGL needs to map the *x* and *y* components of the normalized device coordinates to an area on the screen that the operating system has set aside for display, called the *viewport*; these mapped coordinates are known as *window coordinates*. We don't really need to be too concerned about these coordinates beyond telling OpenGL how to do the mapping. We're currently doing this in our code with a call to `glViewport()` in `onSurfaceChanged()`.

When OpenGL does this mapping, it will map the range (-1, -1, -1) to (1, 1, 1) to the window that has been set aside for display. Normalized device coordinates outside of this range will be clipped. As we learned in Chapter 5, *Adjusting to the Screen's Aspect Ratio*, on page ?, this range is always the same, regardless of the width or height of the viewport.

1. http://en.wikipedia.org/wiki/Homogeneous_coordinates

6.3 Adding the *W* Component to Create Perspective

It will be easier to understand the effects of the w component if we actually see it in action, so let's add it to our table vertex data and see what happens. Since we'll now be specifying the x, y, z, and w components of a position, let's begin by updating `POSITION_COMPONENT_COUNT` as follows:

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java
```
private static final int POSITION_COMPONENT_COUNT = 4;
```

We must always make sure that we're giving OpenGL the right component count for everything that we use; otherwise we'll either have a corrupt screen, nothing will display at all, or we might even crash our application.

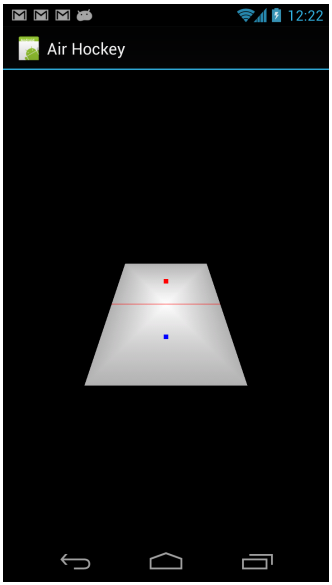
The next step is to update all of our vertices:

AirHockey3D/src/com/airhockey/android/AirHockeyRenderer.java
```
float[] tableVerticesWithTriangles = {
    // Order of coordinates: X, Y, Z, W, R, G, B

    // Triangle Fan
       0f,    0f, 0f, 1.5f,   1f,   1f,   1f,
    -0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,
     0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,
     0.5f,  0.8f, 0f,   2f, 0.7f, 0.7f, 0.7f,
    -0.5f,  0.8f, 0f,   2f, 0.7f, 0.7f, 0.7f,
    -0.5f, -0.8f, 0f,   1f, 0.7f, 0.7f, 0.7f,

    // Line 1
    -0.5f, 0f, 0f, 1.5f, 1f, 0f, 0f,
     0.5f, 0f, 0f, 1.5f, 1f, 0f, 0f,

    // Mallets
    0f, -0.4f, 0f, 1.25f, 0f, 0f, 1f,
    0f,  0.4f, 0f, 1.75f, 1f, 0f, 0f
};
```

We added a z and a w component to our vertex data. We've updated all of the vertices so that the ones near the bottom of the screen have a w of 1 and the ones near the top of the screen have a w of 2; we also updated the line and the mallets to have a fractional w that's in between. This should have the effect of making the top part of the table appear smaller than the bottom, as if we were looking into the distance. We set all of our z components to zero, since we don't need to actually have anything in z to get the perspective effect.

OpenGL will automatically do the perspective divide for us using the w values that we've specified, and our current orthographic projection will just copy

these *w* values over; so let's go ahead and run our project to see what it looks like. It should look similar to the next figure:



Things are starting to look more 3D! We were able to do this just by putting in our own *w*. However, what if we wanted to make things more dynamic, like changing the angle of the table or zooming in and out? Instead of hard-coding the *w* values, we'll use matrices to generate the values for us. Go ahead and revert the changes that we've made; in the next section, we'll learn how to use a perspective projection matrix to generate the *w* values automatically.