Extracted from:

OpenGL ES 2 for Android

A Quick-Start Guide

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

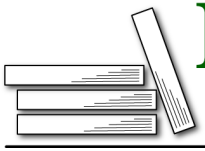
OpenGL ES 2 for Android

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

Now that we have a nice backdrop with some stormy clouds on the horizon, it's time for us to start adding in some substance to our world. In this chapter, we'll learn how to use a height map to add some terrain to the scene. As we do this, we'll touch many new areas of OpenGL and learn how to use the depth buffer for preventing overdraw, and we'll also learn how to store vertex and index data directly on the GPU for better performance.

Height maps are an easy way to add terrain to a scene, and they can easily be generated or edited using a regular paint program. The depth buffer itself is a fundamental part of OpenGL, and it helps us easily render more complex scenes without worrying too much about how that scene is put together.

Here's our game plan:

- First we'll look at how to create a height map and load it into our application using vertex buffer objects and index buffer objects.
- We'll then take our first look at culling and the depth buffer, two techniques for occluding hidden objects.

Let's continue the project from last chapter by copying the code over into a new project called 'Heightmap'.

12.1 Creating a Height Map

A height map is simply a two-dimensional map of heights, much like a topography map that you can find in an atlas. A simple way to create a height map is to use a grayscale image, with the light areas representing high ground and the dark areas representing low ground. Since it's just an image, we can draw our own height map using any regular paint program and end up with something like the next figure. You can even download height maps of real terrain off the Internet from places like the National Elevation Dataset.[1]

1. http://ned.usgs.gov/

Figure 59—A height map example

We'll use the height map from the preceding figure for this project, which can be downloaded from this book's website.[2] Place this image in your project's /res/drawable-nodpi/ folder; in the next step, we'll load this height map data in.

12.2 Creating Vertex and Index Buffer Objects

To load in the height map, we're going to use two new OpenGL objects: a vertex buffer object and an index buffer object. These two objects are analogous to the vertex arrays and index arrays that we've been using in previous chapters, except that the graphics driver can choose to place them directly in the GPU's memory. This can lead to better performance for objects that we don't change often once they've been created, such as a height map. These buffer objects aren't always faster, though, so it does pay to compare both options.

Creating a Vertex Buffer Object

To load in these buffer objects, we'll need to create some supporting code. Let's create a new class called VertexBuffer in the com.particles.android.data, with the following member variable and constructor:

Heightmap/src/com/particles/android/data/VertexBuffer.java

```
private final int bufferId;
```

2. http://pragprog.com/book/kbogla/

```
public VertexBuffer(float[] vertexData) {
    // Allocate a buffer.
    final int buffers[] = new int[1];
    glGenBuffers(buffers.length, buffers, 0);
    if (buffers[0] == 0) {
        throw new RuntimeException("Could not create a new vertex buffer object.");
    }
    bufferId = buffers[0];

    // Bind to the buffer.
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);

    // Transfer data to native memory.
    FloatBuffer vertexArray = ByteBuffer
        .allocateDirect(vertexData.length * BYTES_PER_FLOAT)
        .order(ByteOrder.nativeOrder())
        .asFloatBuffer()
        .put(vertexData);
    vertexArray.position(0);

    // Transfer data from native memory to the GPU buffer.
    glBufferData(GL_ARRAY_BUFFER, vertexArray.capacity() * BYTES_PER_FLOAT,
        vertexArray, GL_STATIC_DRAW);

    // IMPORTANT: Unbind from the buffer when we're done with it.
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

To send vertex data into a vertex buffer object, we first create a new buffer object using glGenBuffers(). This method takes in an array, so we create a new one-element array to store the new buffer ID. We then bind to the buffer with a call to glBindBuffer(), passing in GL_ARRAY_BUFFER to tell OpenGL that this is a vertex buffer.

To copy the data into the buffer object, we have to first transfer it into native memory just like we used to do with VertexArray. Once it's there, we can transfer the data into the buffer object with a call to glBufferData(). Let's take a look at this method's parameters in more detail (see Table 7, *glBufferData parameters*, on page 10).

When we're done loading data into the buffer, we need to make sure that we unbind from the buffer by calling glBindBuffer() with 0 as the buffer ID; otherwise calls to functions like glVertexAttribPointer() elsewhere in our code will not work properly.

We'll also need a wrapper to glVertexAttribPointer() like we had with our old VertexArray class, so let's add a new method called setVertexAttribPointer():

glBufferData(int target, int size, Buffer data, int usage)

int target	This should be GL_ARRAY_BUFFER for a vertex buffer object, or GL_ELEMENT_ARRAY_BUFFER for an index buffer object.
int size	This is the size of the data in bytes.
Buffer data	This should be a Buffer object that was created with allocateDirect().
int usage	This tells OpenGL the expected usage pattern for this buffer object. Here are the options: *GL_STREAM_DRAW* This object will only be modified once and only used a few times. *GL_STATIC_DRAW* This object will be modified once, but it will be used many times. *GL_DYNAMIC_DRAW* This object will be both modified and used many times. These are hints rather than constraints so that OpenGL can do optimizations on its end. We'll want to use GL_STATIC_DRAW most of the time.

Table 7—glBufferData parameters

Heightmap/src/com/particles/android/data/VertexBuffer.java

```
public void setVertexAttribPointer(int dataOffset, int attributeLocation,
    int componentCount, int stride) {
    glBindBuffer(GL_ARRAY_BUFFER, bufferId);
    glVertexAttribPointer(attributeLocation, componentCount, GL_FLOAT,
        false, stride, dataOffset);
    glEnableVertexAttribArray(attributeLocation);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

The main differences here are that we now need to bind to the buffer before calling glVertexAttribPointer(), and we use a slightly different glVertexAttribPointer() that takes in an int instead of a Buffer as the last parameter. This integer tells OpenGL the offset in bytes for the current attribute; this could be 0 for the first attribute or a specific byte offset for subsequent attributes.

As before, we make sure to unbind from the buffer before returning from the method.

Creating an Index Buffer Object

We'll also need a wrapper class for the index buffer, so go ahead and create a new class called IndexBuffer in the same package as VertexBuffer. You can copy/paste over the member variable and constructor from VertexBuffer and make the following changes:

- Use short[] and ShortBuffer as the types.
- Use GL_ELEMENT_ARRAY_BUFFER instead of GL_ARRAY_BUFFER.
- To get the size in bytes, add the new constant BYTES_PER_SHORT to Constants with the value 2, and use that instead of BYTES_PER_FLOAT when you call glBufferData().

We'll need to use the buffer ID when we use it to draw, so let's add an accessor for it:

Heightmap/src/com/particles/android/data/IndexBuffer.java

```
public int getBufferId() {
    return bufferId;
}
```

Now that we have our supporting code in place, let's get that height map loaded in.

12.3 Loading in the Height Map

To load the height map into OpenGL, we need to load in the image data and convert it into a set of vertices, one for each pixel. Each vertex will have a position based on its position in the image and a height based on the brightness of the pixel. Once we have all of the vertices loaded in, we'll use the index buffer to group them into triangles that we can draw with OpenGL.

Generating the Vertex Data

Let's create a new class called Heightmap in the com.particles.android.objects package, adding the following code inside the class to start out:

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private static final int POSITION_COMPONENT_COUNT = 3;

private final int width;
private final int height;
private final int numElements;
private final VertexBuffer vertexBuffer;
private final IndexBuffer indexBuffer;

public Heightmap(Bitmap bitmap) {
```

```
        width = bitmap.getWidth();
        height = bitmap.getHeight();

        if (width * height > 65536) {
            throw new RuntimeException("Heightmap is too large for the index buffer.");
        }
        numElements = calculateNumElements();
        vertexBuffer = new VertexBuffer(loadBitmapData(bitmap));
        indexBuffer = new IndexBuffer(createIndexData());
    }
```

We pass in an Android bitmap, load the data into a vertex buffer, and create an index buffer for those vertices. Let's start adding the definition for loadBitmapData():

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private float[] loadBitmapData(Bitmap bitmap) {
    final int[] pixels = new int[width * height];
    bitmap.getPixels(pixels, 0, width, 0, 0, width, height);
    bitmap.recycle();

    final float[] heightmapVertices =
        new float[width * height * POSITION_COMPONENT_COUNT];
    int offset = 0;
```

To efficiently read in all of the bitmap data, we first extract all of the pixels with a call to getPixels(), and then we recycle the bitmap since we won't need to keep it around. Since there will be one vertex per pixel, we create a new array for the vertices with the same width and height as the bitmap.

Let's add some code to convert the bitmap pixels into height map data:

Heightmap/src/com/particles/android/objects/Heightmap.java

```
    for (int row = 0; row < height; row++) {
        for (int col = 0; col < width; col++) {
            final float xPosition = ((float)col / (float)(width - 1)) - 0.5f;
            final float yPosition =
                (float)Color.red(pixels[(row * height) + col]) / (float)255;
            final float zPosition = ((float)row / (float)(height - 1)) - 0.5f;

            heightmapVertices[offset++] = xPosition;
            heightmapVertices[offset++] = yPosition;
            heightmapVertices[offset++] = zPosition;
        }
    }
    return heightmapVertices;
}
```

To generate each vertex of the height map, we first calculate the vertex's position; the height map will be 1 unit wide in each direction and centered

at an *x*-*z* of (0, 0), so with these loops, the upper left corner of the bitmap will map to (-0.5, -0.5), and the lower right corner will map to (0.5, 0.5).

We assume that the image is grayscale, so we read the red component of the pixel and divide that by 255 to get the height. A pixel value of 0 will correspond to a height of 0, and a pixel value of 255 will correspond to a height of 1. Once we've calculated the position and the height, we can write out the new vertex to the array.

Before we move on, let's take a closer look at this loop. Why do we read the bitmap row by row, scanning each column from left to right? Why not read the bitmap column by column instead? The reason we read the data row by row is because that's how the bitmap is laid out sequentially in memory, and CPUs are much better at caching and moving data around when they can do it in sequence.

It's also important to note the way we are accessing the pixels. When we extracted the pixels using getPixels(), Android gave us a one-dimensional array. How then do we know where to read in the pixels? We can calculate the right place with the following formula:

```
pixelOffset = currentRow * height + currentColumn
```

Using this formula, we can use two loops to read in a one-dimensional array as if it were a two-dimensional bitmap.

Generating the Index Data

Back in the constructor, we figured out how many index elements we needed by calling calculateNumElements(), and we saved the result in numElements. Let's go ahead and create that method now:

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private int calculateNumElements() {
    return (width - 1) * (height - 1) * 2 * 3;
}
```

The way this works is that for every group of 4 vertices in the height map, we'll generate 2 triangles, 3 indices for each triangle, for a total of 6 indices. We can calculate how many groups we need by multiplying (width - 1) by (height - 1), and then we just multiply that by 2 triangles per group and 3 elements per triangle to get the total element count. For example, a height map of 3 x 3 will have (3 - 1) x (3 - 1) = 2 x 2 = 4 groups. With two triangles per group and 3 elements per triangle, that's a total of 24 elements.

Let's generate the indices with the following code:

Heightmap/src/com/particles/android/objects/Heightmap.java

```
private short[] createIndexData() {
    final short[] indexData = new short[numElements];
    int offset = 0;

    for (int row = 0; row < height - 1; row++) {
        for (int col = 0; col < width - 1; col++) {
            short topLeftIndexNum = (short) (row * width + col);
            short topRightIndexNum = (short) (row * width + col + 1);
            short bottomLeftIndexNum = (short) ((row + 1) * width + col);
            short bottomRightIndexNum = (short) ((row + 1) * width + col + 1);

            // Write out two triangles.
            indexData[offset++] = topLeftIndexNum;
            indexData[offset++] = bottomLeftIndexNum;
            indexData[offset++] = topRightIndexNum;

            indexData[offset++] = topRightIndexNum;
            indexData[offset++] = bottomLeftIndexNum;
            indexData[offset++] = bottomRightIndexNum;
        }
    }

    return indexData;
}
```

This method creates an array of shorts with the required size, and then it loops through the rows and columns, creating triangles for each group of four vertices. We don't even need the actual pixel data to do this; all we need is the width and the height. We first learned about indices back in Section 11.3, *Creating a Cube*, on page ?, and this code follows the same pattern.

Something interesting happens if you try to store index values greater than 32,767: the cast to short will cause the number to *wrap around into a negative value*. However, due to two's complement, these negative numbers will have the right value when OpenGL reads them in as unsigned values (see *Converting Between Signed and Unsigned Data Types*, on page ?). As long as we don't have more than 65,536 elements to index, we'll be fine.

Tips & Gotchas

There are a few things to watch out for when using buffer objects. Technically, Android supports OpenGL ES 2 starting from Android 2.2 (Froyo), but unfortunately these bindings are broken, and vertex and index buffers are unusable from Java without writing a custom Java Native Interface (JNI) binding.

The good news is that these bindings were fixed in Android's Gingerbread release, and as of the time of this writing, only 9 percent of the market is still on Froyo, so this problem isn't as big of a deal as it used to be.

Just like with Java's ByteBuffers, using OpenGL buffer objects improperly can lead to native crashes, which can be difficult to debug. If your application suddenly disappears and you see something like "Segmentation fault" in the Android log, it's a good idea to double-check all of your calls involving the buffers, especially calls to glVertexAttribPointer().