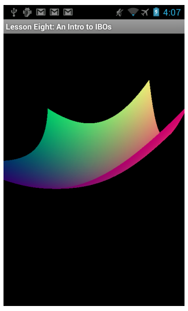
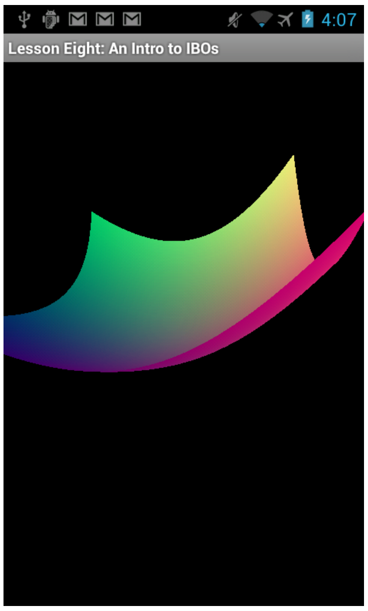
Learn OpenGL ES

Learn how to develop mobile graphics using OpenGL ES 2

Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs)





In our last lesson, we learned how to use vertex buffer objects on Android. We learned about the difference between client-side memory and GPU-dedicated memory, and the difference between storing texture, position and normal data in separate buffers, or altogether in one buffer. We also learned how to work around Froyo's broken OpenGL ES 2.0 bindings.

In this lesson, we'll learn about index buffer objects, and go over a practical example of how to use them. Here's what we're going to cover:

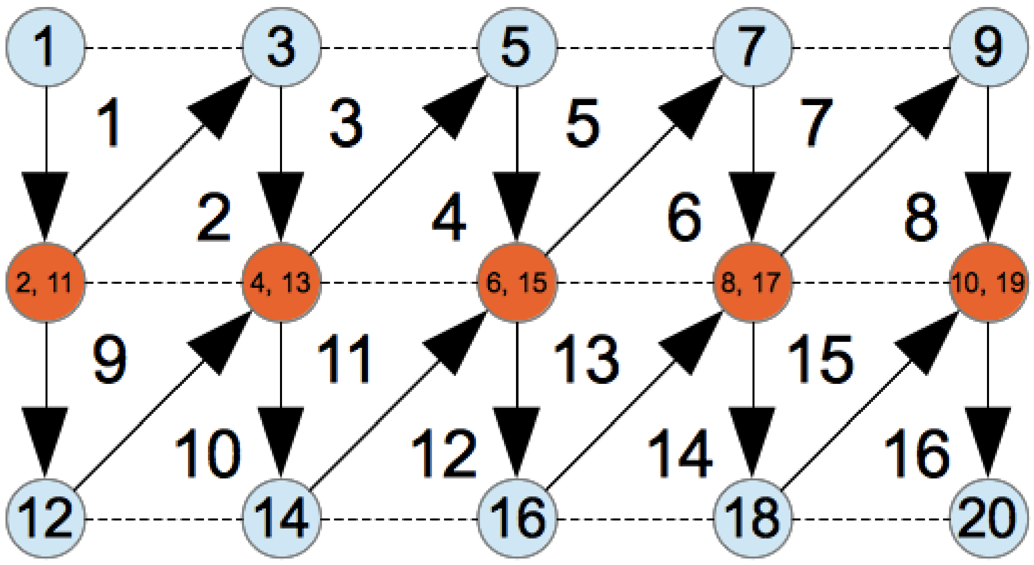
- The difference between using vertex buffer objects only, and using vertex buffer objects together with index buffer objects.
- How to join together triangle strips using degenerate triangles, and render an entire height map in a single rendering call.

Let's get started with the fundamental difference between vertex buffer objects and index buffer objects:

VERTEX BUFFER OBJECTS AND INDEX BUFFER OBJECTS

In the previous lesson, we learned that a vertex buffer object is simply an array of vertex data which is directly rendered by OpenGL. We can use separate buffers for each attribute, such as positions and colors, or we can use a single buffer and interleave all of the data together. Contemporary articles suggest interleaving the data and making sure it's aligned to 4-byte boundaries for better performance.

The downside to vertex buffers come when we use many of the same vertices over and over again. For example, a heightmap can be broken down into a series of triangle strips. Since each neighbour strip shares one row of vertices, we will end up repeating a lot of vertices with a vertex buffer.

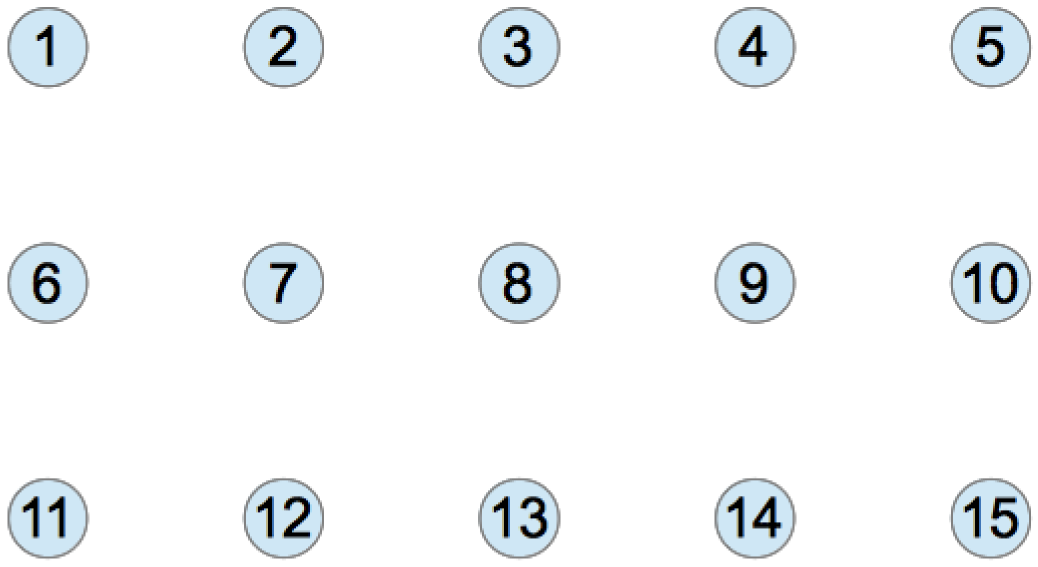


You can see a vertex buffer object containing the vertices for two triangle strip rows. The order of vertices is shown, and the triangles defined by these vertices is also shown, when drawn using *glDrawArrays(GL_TRIANGLE_STRIP, …)*. In this example, we're assuming that each row of triangles gets sent as a separate call to *glDrawArrays()*. Each vertex contains data as follows:

```
vertexBuffer = {
    // Position - Vertex 1
    0, 0, 0,
    // Color
    1,1,1
    // Normal
    0,0,1,
    // Position - Vertex 2
    1, 0, 0,
    ...
}
```

As you can see, the middle row of vertices needs to be sent twice, and this will also happen for each additional row of the height map. As our height map gets larger, our vertex buffer could end up having to repeat a lot of position, color, and normal data and consume a lot of additional memory.

How can we improve on this state of affairs? We can use an *index buffer object*. Instead of repeating vertices in the vertex buffer, we'll define each vertex once, and only once. We'll refer to these vertices using offsets into this vertex buffer, and when we need to reuse a vertex, we'll repeat the offset instead of repeating the entire vertex. Here's a visual illustration of the new vertex buffer:



Notice that our vertices are no longer linked together into triangles. We'll no longer pass our vertex buffer object directly; instead, we'll use an index buffer to tie the vertices together. The index buffer will contain *only* the offsets to our vertex buffer object. If we wanted to draw a triangle strip using the above buffer, our index buffer would contain data as follows:

```
    indexBuffer = {
        1, 6, 2, 7, 3, 8, ...
    }
```

That's how we'll link everything together. When we repeat the middle row of vertices, we only repeat the number, instead of repeating the entire block of data. We'll draw the index buffer with a call to *glDrawElements(GL_TRIANGLE_STRIP, …).*

LINKING TOGETHER TRIANGLE STRIPS WITH DEGENERATE TRIANGLES

The examples above assumed that we would render each row of the heightmap with a separate call to *glDrawArrays()* or *glDrawElements()*. How can we link up each row to the next? After all, the end of the first row is all the way on the right, and the beginning of the second row on the left. How do we link the two?

We don't need to get fancy and start drawing right to left or anything like that. We can instead use what's known as a *degenerate triangle*. A degenerate triangle is a triangle that has no area, and when the GPU encounters such triangles, it will simply skip over them.

Let's look at our index buffer again:

```
indexBuffer = {
    1, 6, 2, 7, 3, 8, 4, 9, ...
}
```

When drawing with GL_TRIANGLE_STRIP, OpenGL will build triangles by taking each set of three vertices, advancing by one vertex for each triangle. Every subsequent triangle shares two vertices with the previous triangle. For example, here are the sets of vertices that would be grouped into triangles:

- Triangle 1 = 1, 6, 2
- Triangle 2 = 6, 2, 7
- Triangle 3 = 2, 7, 3
- Triangle 4 = 7, 3, 8
- Triangle 5 = 3, 8, 4
- Triangle 6 = 8, 4, 9

- …

OpenGL also maintains a specific order, or *winding* when building the triangles. The order of the first 3 vertices determines the order for the rest. If the first triangle is counter-clockwise, the rest will be counter-clockwise, too. OpenGL does this by swapping the first two vertices of every even triangle (swapped vertices are in bold):

- Triangle 1 = 1, 6, 2
- Triangle 2 = **2, 6,** 7
- Triangle 3 = 2, 7, 3
- Triangle 4 = **3, 7,** 8
- Triangle 5 = 3, 8, 4
- Triangle 6 = **4, 8,** 9
- …

Let's show the entire index buffer, including the missing link between each row of the height map:

```
indexBuffer = {
    1, 6, 2, 7, 3, 8, 4, 9, 5, 10, ..., 6, 11, 7, 12, 8, 13, 9, 14, 10,
15
}
```

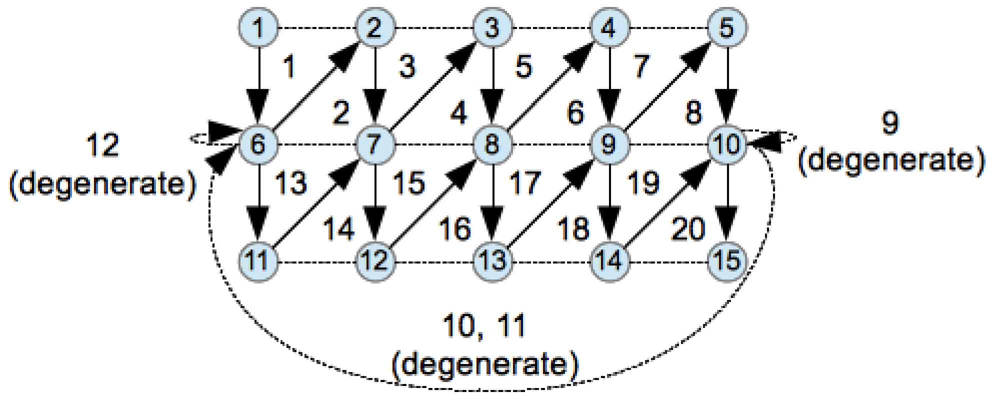
What do we need to put in between, in order to link up the triangles? We'll need an even number of new triangles in order to preserve the winding. We can do this by repeating the last vertex of the first row, and the first vertex of the second row. Here's the new index buffer below, with the duplicated vertices in bold:

```
indexBuffer = {
    1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 10, 6, 6, 11, 7, 12, 8, 13, 9, 14, 10,
15
}
```

Here's what the new sequence of triangles looks like:

- …
- Triangle 8 = 5, 9, 10
- Triangle 9 (degenerate) = 5, 10, 10
- Triangle 10 (degenerate) = 10, 10, 6
- Triangle 11 (degenerate) = 10, 6, 6
- Triangle 12 (degenerate) = 6, 6, 11
- Triangle 13 = 6, 11, 7
- …

By repeating the last vertex and the first vertex, we created four degenerate triangles that will be skipped, and linked the first row of the height map with the second. We could link an arbitrary number of rows this way and draw the entire heightmap with one call to *glDrawElements()*. Let's take a look visually:



The degenerate triangles link each row with the next row.

Degenerate triangles done the wrong way

We need to repeat both the last vertex of the first row and the first of the second row. What would happen if we didn't? Let's say we only repeated one vertex:

```
indexBuffer = {
    1, 6, 2, 7, 3, 8, 4, 9, 5, 10, **10,** 6, 11, 7, 12, 8, 13, 9, 14, 10, 15
}
```

Here's what the sequence of triangles would look like:

- …
- Triangle 8 = 5, 9, 10
- Triangle 9 (degenerate) = 5, 10, 10
- Triangle 10 (degenerate) = 10, 10, 6
- **Triangle 11 = 10, 6, 11**
- Triangle 12 = 11, 6, 7
- …

Triangle 11 starts at the right and cuts all the way across to the left, which isn't what we wanted to happen. The winding is now also incorrect for the next row of triangles, since 3 new triangles were inserted, swapping even and odd.

A PRACTICAL EXAMPLE

Let's walk through the code to make it happen. I highly recommend heading over and reading Android Lesson Seven: An Introduction to Vertex Buffer Objects (VBOs) before continuing.

Do you remember those graph calculators, that could draw parabolas and other stuff on the screen? In this example, we're going to draw a 3d parabola using a height map. We'll walk through all of the code to build and draw the height map. First, let's get started with the definitions:

```
class HeightMap {
    static final int SIZE_PER_SIDE = 32;
    static final float MIN_POSITION = -5f;
    static final float POSITION_RANGE = 10f;

    final int[] vbo = new int[1];
    final int[] ibo = new int[1];

    int indexCount;
```

- We'll set the height map to 32 units per side, for a total of 1,024 vertices and 1,922 triangles, not including degenerate triangles (The total number of triangles in a height map is equal to (2 * (units_per_side – 1)^2)). The height map positions will range from -5 to +5.

- The OpenGL reference to our vertex buffer object and index buffer object will go in vbo and ibo, respectively.
- *indexCount* will hold the total number of generated indices.

Building the vertex data

Let's look at the code that will build the vertex buffer object. Remember that we still need a place to hold all of our vertices, and that each vertex will be defined once, and only once.

```
HeightMap() {
    try {
        final int floatsPerVertex = POSITION_DATA_SIZE_IN_ELEMENTS + NORMA
                + COLOR_DATA_SIZE_IN_ELEMENTS;
        final int xLength = SIZE_PER_SIDE;
        final int yLength = SIZE_PER_SIDE;

        final float[] heightMapVertexData = new float[xLength * yLength *

        int offset = 0;

        // First, build the data for the vertex buffer
        for (int y = 0; y &lt; yLength; y++) {
            for (int x = 0; x &lt; xLength; x++) {
                final float xRatio = x / (float) (xLength - 1);

                // Build our heightmap from the top down, so that our tria
                // counter-clockwise.
                final float yRatio = 1f - (y / (float) (yLength - 1));

                final float xPosition = MIN_POSITION + (xRatio * POSITION_
                final float yPosition = MIN_POSITION + (yRatio * POSITION_

                ...
            }
        }
```

This bit of code sets up the loop to generate the vertices. Before we can send data to OpenGL, we need to build it in Java's memory, so we create a floating point array to hold the height map vertices. Each vertex will hold enough floats to contain all of the position, normal, and color information.

Inside the loop, we calculate a ratio between 0 and 1 for x and y. This xRatio and yRatio will then be used to calculate the current position for the next vertex.

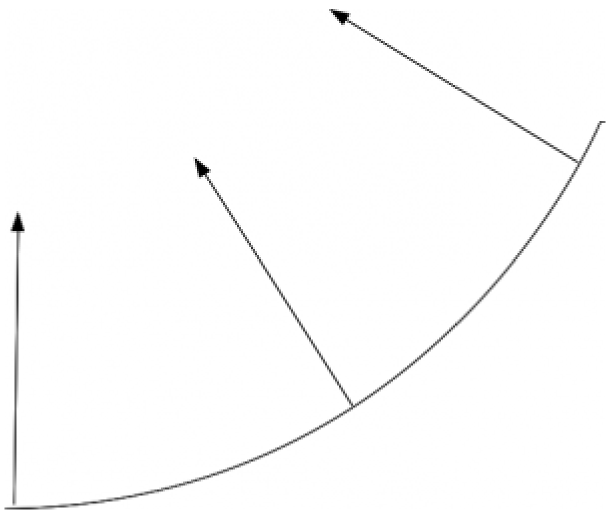
Let's take a look at the actual calculations within the loop:

```
// Position
heightMapVertexData[offset++] = xPosition;
heightMapVertexData[offset++] = yPosition;
heightMapVertexData[offset++] = ((xPosition * xPosition) + (yPosition * yP
```

First up is the position. Since this is a 3D parabola, we'll calculate the Z as $X^2 + Y^2$. We divide the result by 10 so the resulting parabola isn't so steep.

```
// Cheap normal using a derivative of the function.
// The slope for X will be 2X, for Y will be 2Y.
// Divide by 10 since the position's Z is also divided by 10.
final float xSlope = (2 * xPosition) / 10f;
final float ySlope = (2 * yPosition) / 10f;

// Calculate the normal using the cross product of the slopes.
final float[] planeVectorX = {1f, 0f, xSlope};
final float[] planeVectorY = {0f, 1f, ySlope};
final float[] normalVector = {
        (planeVectorX[1] * planeVectorY[2]) - (planeVectorX[2] * planeVect
        (planeVectorX[2] * planeVectorY[0]) - (planeVectorX[0] * planeVect
        (planeVectorX[0] * planeVectorY[1]) - (planeVectorX[1] * planeVect

// Normalize the normal
final float length = Matrix.length(normalVector[0], normalVector[1], norma

heightMapVertexData[offset++] = normalVector[0] / length;
heightMapVertexData[offset++] = normalVector[1] / length;
heightMapVertexData[offset++] = normalVector[2] / length;
```

Next up is the normal calculation. As you'll remember from our lesson on lighting, the normal will be used to calculate lighting. The normal of a surface is defined as a vector perpendicular to the tangent plane at that particular point. In other words, the normal should be an arrow pointing straight away from the surface. Here's an visual example for a parabola:



The first thing we need is the tangent of the surface. Using a bit of calculus, (don't worry, I didn't remember it either and went and searched for an online calculator ;)) we know that we can get the tangent, or the slope from the derivative of the function. Since our function is $X^2 + Y^2$, our slopes will therefore be 2X and 2Y. We scale the slopes down by 10, since for the position we had also scaled the result of the function down by 10.

To calculate the normal, we create two vectors for each slope to define a plane, and we calculate the cross product to get the normal, which is the perpendicular vector.

We then *normalize* the normal by calculating its length, and dividing each component by the length. This ensures that the overall length will be equal to 1.

```
// Add some fancy colors.
heightMapVertexData[offset++] = xRatio;
heightMapVertexData[offset++] = yRatio;
heightMapVertexData[offset++] = 0.5f;
heightMapVertexData[offset++] = 1f;
```

Finally, we'll set some fancy colors. Red will scale from 0 to 1 across the X axis, and green will scale from 0 to 1 across the Y axis. We add a bit of blue to brighten things up, and assign 1 to alpha.

Building the index data

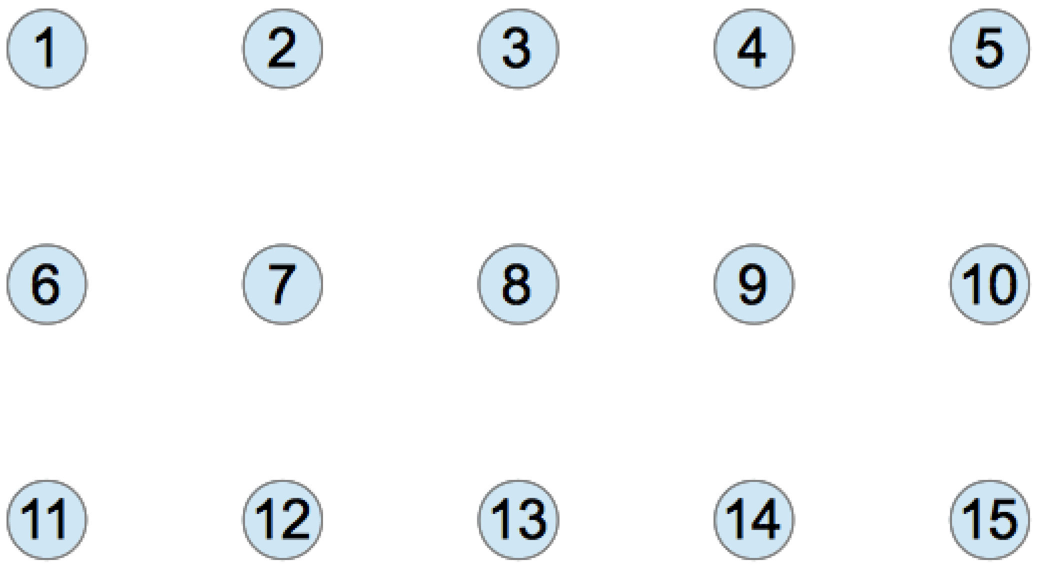
The next step is to link all of these vertices together, using the index buffer.

```
// Now build the index data
final int numStripsRequired = yLength - 1;
final int numDegensRequired = 2 * (numStripsRequired - 1);
final int verticesPerStrip = 2 * xLength;

final short[] heightMapIndexData = new short[(verticesPerStrip * numStrip
        + numDegensRequired];

offset = 0;

for (int y = 0; y &lt; yLength - 1; y++) {      if (y &gt; 0) {
        // Degenerate begin: repeat first vertex
        heightMapIndexData[offset++] = (short) (y * yLength);
    }

    for (int x = 0; x &lt; xLength; x++) {
        // One part of the strip
        heightMapIndexData[offset++] = (short) ((y * yLength) + x);
```

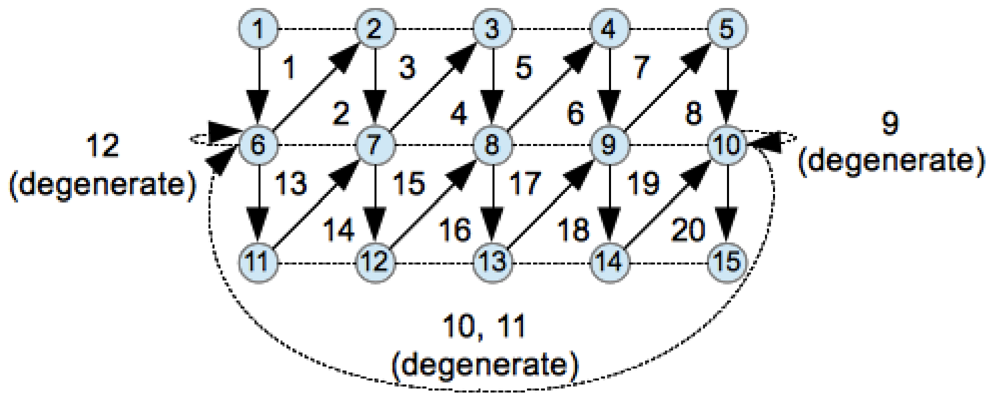
```
            heightMapIndexData[offset++] = (short) (((y + 1) * yLength) + x);
        }

        if (y &lt; yLength - 2) {
            // Degenerate end: repeat last vertex
            heightMapIndexData[offset++] = (short) (((y + 1) * yLength) + (xL
        }
    }

    indexCount = heightMapIndexData.length;
```

In OpenGL ES 2, an index buffer needs to be an array of unsigned bytes or shorts, so we use shorts here. We’ll read from two rows of vertices at a time, and build a triangle strip using those vertices. If we’re on the second or subsequent rows, we’ll duplicate the first vertex of that row, and if we’re on any row but the last, we’ll also duplicate the last vertex of that row. This is so we can link the rows together using degenerate triangles as described earlier.

We’re just assigning offsets here. Imagine we had the height map as shown earlier:



With the index buffer, we want to end up with something like this:



Our buffer will contain data like this:

```
heightMapIndexData = {1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 10, 6, 6, 11, 7, 12,
8, 13, 9, 14, 10, 15}
```

Just keep in mind that although our examples start with 1, and go on to 2, 3, etc… in the actual code, our arrays should be 0-based and start with 0.

Uploading data into vertex and index buffer objects

The next step is to copy the data from Dalvik's heap to a direct buffer on the native heap:

```
final FloatBuffer heightMapVertexDataBuffer = ByteBuffer
    .allocateDirect(heightMapVertexData.length * BYTES_PER_FLOAT).order(By
    .asFloatBuffer();
heightMapVertexDataBuffer.put(heightMapVertexData).position(0);

final ShortBuffer heightMapIndexDataBuffer = ByteBuffer
    .allocateDirect(heightMapIndexData.length * BYTES_PER_SHORT).order(Byt
    .asShortBuffer();
heightMapIndexDataBuffer.put(heightMapIndexData).position(0);
```

Remember, the index data needs to go in a short buffer or a byte buffer. Now we can create OpenGL buffers, and upload our data into the buffers:

```
GLES20.glGenBuffers(1, vbo, 0);
GLES20.glGenBuffers(1, ibo, 0);

if (vbo[0] &gt; 0 &amp;&amp; ibo[0] &gt; 0) {
    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vbo[0]);
    GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER, heightMapVertexDataBuffer.
                            * BYTES_PER_FLOAT, heightMapVertexDataBuffer,

    GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, ibo[0]);
    GLES20.glBufferData(GLES20.GL_ELEMENT_ARRAY_BUFFER, heightMapIndexData
                            * BYTES_PER_SHORT, heightMapIndexDataBuffer, (

    GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
    GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, 0);
} else {
    errorHandler.handleError(ErrorType.BUFFER_CREATION_ERROR, &quot;glGenE
}
```

We use *GL_ARRAY_BUFFER* to specify our vertex data, and *GL_ELEMENT_ARRAY_BUFFER* to specify our index data.

Drawing the height map

Much of the code to draw the height map will be similar as in previous lessons. I won't cover the matrix setup code here; instead, we'll just look at the calls to bind the array data and draw using the index buffer:

```
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, vbo[0]);

// Bind Attributes
glEs20.glVertexAttribPointer(positionAttribute, POSITION_DATA_SIZE_IN_ELEM
        false, STRIDE, 0);
GLES20.glEnableVertexAttribArray(positionAttribute);

glEs20.glVertexAttribPointer(normalAttribute, NORMAL_DATA_SIZE_IN_ELEMENTS
        false, STRIDE, POSITION_DATA_SIZE_IN_ELEMENTS * BYTES_PER_FLOAT);
GLES20.glEnableVertexAttribArray(normalAttribute);

glEs20.glVertexAttribPointer(colorAttribute, COLOR_DATA_SIZE_IN_ELEMENTS,
        false, STRIDE, (POSITION_DATA_SIZE_IN_ELEMENTS + NORMAL_DATA_SIZE_
        * BYTES_PER_FLOAT);
GLES20.glEnableVertexAttribArray(colorAttribute);

// Draw
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, ibo[0]);
glEs20.glDrawElements(GLES20.GL_TRIANGLE_STRIP, indexCount, GLES20.GL_UNSI

GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
GLES20.glBindBuffer(GLES20.GL_ELEMENT_ARRAY_BUFFER, 0);
```

In the previous lesson, we had to introduce a custom OpenGL binding to properly use VBOs, as they're broken in Froyo. We'll also need to use this binding for IBOs as well. Just like in the previous lesson, we bind our position, normal, and color data in our vertex buffer to the matching attribute in the shader, taking care to pass in the proper stride and start offset of each attribute. Both the stride and the start offset are defined in terms of *bytes*.

The main difference is when it's time to draw. We call *glDrawElements()* instead of *glDrawArrays()*, and we pass in the index count and data type. OpenGL ES 2.0 only accepts GL_UNSIGNED_SHORT and GL_UNSIGNED_BYTE, so we have to make sure that we define our index data using shorts or bytes.

Rendering and lighting two-sided triangles

For this lesson, we don't enable GL_CULL_FACE so that both the front and the back sides of triangles are visible. We need to make a slight change to our fragment shader code so that lighting works properly for our two-sided triangles:

```
float diffuse;

if (gl_FrontFacing) {
    diffuse = max(dot(v_Normal, lightVector), 0.0);
} else {
    diffuse = max(dot(-v_Normal, lightVector), 0.0);
}
```

We use the special variable *gl_FrontFacing* to find out if the current fragment is part of a front-facing or back-facing triangle. If it's front-facing, then no need to do anything special: the code is the same as before. If it's back-facing, then we simply invert the surface normal (since the back side of a triangle faces the opposite direction) and proceed with the calculations as before.

Further exercises

When would it be a downside to use index buffers? Remember, when we use index buffers the GPU has to do an additional fetch into the vertex buffer object, so we're introducing an additional step.

WRAPPING UP

The full source code for this lesson can be downloaded from the project site on GitHub. A compiled version of the lesson can also be downloaded directly from the Android Market:

Thanks for stopping by, and please feel free to check out the code and share your comments below.

Author: Admin

Kevin is the author of OpenGL ES 2 for Android: A Quick-Start Guide. He also has extensive experience in Android development. View all posts by Admin

Admin / May 10, 2012 / Android, Android Tutorials / Dalvik, degenerate triangles, Froyo, height maps, IBOs, Index Buffer Object, triangle strips, VBOs, Vertex Buffer Object

43 thoughts on "Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs)"

Martin

May 28, 2012 at 1:46 pm

Thank you for the tutorial, (which is quite good like allways). But you have avoided a problem by using a waved mesh. Otherwise you couldn't index the normals to light the sharp edge well.Would you store them in a seperate unindexed buffer in case?

Admin

May 29, 2012 at 6:13 am

Good point. For a height map you can probably get away with not using sharp edges, if the map is meant to represent terrain or something else that has mostly smooth edges.

For other objects, it depends. If you don't reuse data for the same points, maybe you're better off not using an index buffer? Index buffers help to reduce the data when you reuse a lot of the same data, but if your data keeps changing then you might be better off just feeding the GPU the vertices directly. If you really want to get detailed you can profile the different cases. 😉

Ray Ingles

May 30, 2012 at 4:20 pm

Thanks for taking the time to do these lessons. I've confirmed they run very well on even my original Droid, and – unlike many – you actually know how to *comment* code so it can be followed. Kudos!

Admin

June 6, 2012 at 2:18 am

Thanks Ray, I appreciate it! 🙂

Oscar

June 4, 2012 at 7:07 pm

Hey there, I’ve been following your tutorials.

This is a great one but requires some math study to really understand it (apart from the ibos usage).

I have a couple of questions, I would appreciate alot if you could help me.

Is it efficient to use several buffer objects for each UV mapping?
I.E. I create 2 buffers for a 2framed bitmap, mapped from 0.0 to 0.5 and 0.5 to 1.0 respectivelly for a texture to be showed on a square.

I assume the qty of buffers by the qty of framed on a rectangular bitmap. I then tell opengl to use certain buffer of the uv mapping depending on the frame time. Is this a good aproach?

I also notice little freezes (hardly detectable for an average user) that are not even detected on FPS meassurement.

If the GC acts, it would be shown on an FPS meassurement (based on time and times updated) wouldn’t it?

I’m also a bit dissapointed of myself because I study alot your tutorials and learn but at the same time I feel I don’t achieve the expected results (performance and math wise).

Help would be much appreciated, thank you for your efforts!

P.s. Is there a way to detect in Eclipse the performance of your gamr Memory Wise and to know if you use to many objects?

Learn OpenGL ES

June 6, 2012 at 1:04 am

Hi Oscar,

No need to feel disappointed! I'm happy that you're putting the effort in and asking some good questions! 🙂

About buffers: The iPhone guide recommends one buffer for everything, and then you'd just use an offset, but the best way to see is to test for yourself. For small sets of data you could also stick with client-side, and just change the data. In my own testing I haven't seen a big difference either way, but that could change from phone to phone.

For the little freezes, unfortunately there's not too much you can do about that on Android. Even Angry Birds and other games have that problem; it's just the way the platform is. You can have small twitches here and there without affecting the FPS significantly, so you won't really see it on that measurement. There's a saying somewhere that a consistent 30 fps is much better than a shaky 60 fps. 😉 As the hardware gets more powerful and the platform gets better this will become less of an issue.

For memory analysis, this blog post has a lot of good info: http://android-developers.blogspot.ca/2011/03/memory-analysis-for-android.html

Hope this helps! 🙂

Oscar

July 25, 2012 at 1:57 pm

Hello again!

I posted time ago, in fact I'm replying your much appreciated response :). Thanks to your tutorials, I made a custom engine, using what I learned in here!

I published a game I made using such engine, and plan to publish more of them!

It's a 2D engine but it's a start, I do use VBOs and many techniques I learned with you.

The link to google play is:
https://play.google.com/store/apps/details?id=com.ThePathGames.PingPongMania&feature=search_result#?t=W251bGwsMSwxLDEsImNvbS5UaGVQYXRoR2FtZXMuUGluZ1BvbmdNYW5pYSJd

Thank you again for all our tutorials!

P.S. If you could make a tutorial of how to use a 3D editor and make a class that take a text object into a 3D mesh with draw_elements that would be so good!

Oscar

July 25, 2012 at 2:00 pm

I also forgot to ask:

When you make your games/apps, do you make it compatible with Open GL ES 1.x?
I find it a bit complicated to find a way to make it compatible with my current engine 🙁

hellbert

June 26, 2012 at 1:57 pm

Hi, thank you for these great tutorials! It’s a very enjoyable course. I wanted to ask you if are you planning to extend it with new lessons soon or the course is finished by now.

Thank you and keep going with that nice work!

Admin

June 26, 2012 at 2:47 pm

Thanks hellbert! I would definitely like to continue the course. Each post takes a long time for me to write, so that's why they don't come as often as I'd like. 😉 Maybe the next step is to add a wiki and/or forums to the site. What do you think?

sankar

August 4, 2012 at 9:43 am

hi,

can you provide me a link or source code to generate a simple cube using IBO. i have tried to search for it but could not find for open gl es 2.0

thanks
sankar

sankar

August 7, 2012 at 11:56 am

code for my project can be downloaded from here. i do not see anything on the screen when i run applicaiton

http://code.google.com/p/my3dcube/downloads/list

sankar

August 8, 2012 at 7:16 am

hi,

i solved it myself..thanks

Admin

August 8, 2012 at 6:33 pm

No worries! Sometimes that's the best way to learn. 😉

---

jason

November 9, 2012 at 7:08 am

really the best opengl es tutorial I have ever seen. any more lessons about stencils?

Admin

November 20, 2012 at 2:49 am

Thanks Jason! Have unfortunately not made it to stencils yet. 😉

Zhenya

November 25, 2012 at 11:01 am

Great tutorials!! Omg I just cant wait for Lesson Nine to come out!!! Would that be an advanced blending lesson?? Oh, or introduction to compressed texture format maybe? Here I have a small question, recently I'm curious in those mirror effect, for example water reflection, can you please give a hint on how to do this? Much appreciate!!

RayDeeA

December 7, 2012 at 11:15 am

Hey I used this approach for my terrain, since I think it is really a very good one. I have only a lil problem which I'm describing here in more detail http://stackoverflow.com/questions/13746373/weird-order-when-painting-triangle-outlines-using-gl-line-strip. When I paint the mesh using gl_line_strip the resulting triangle outlines are mirrored vertically. Does somebody know why or could u tell me if the same problem occurs also in ur application when switching the drawing mode to gl_line_strip? Thank u in advance.

Rob

January 31, 2013 at 6:00 am

Thanks a lot for this. This is a great start for someone that doesn't know OpenGL!

Amila Madusanka

February 17, 2013 at 7:21 pm

Hi , this is a great tutorial. I have a question. What is most preferable(considering performance) language(java or native code) for android game development. Thanks

Admin

February 18, 2013 at 10:14 pm

For the best performance and portability, you might want to look into native development; however, if you want to focus on Android and you're not a C/C++ expert, you might be able to get things done a lot more quickly in Java! The performance of Dalvik is acceptable for a lot of things, and you can always use Renderscript/NDK for integrating native libraries or for improving the performance of specific functions.

berloger

February 22, 2013 at 8:21 am

sorry, I think there is a small mistake in the code:
//——————————————–
for (int y = 0; y 0) {
// Degenerate begin: repeat first vertex
heightMapIndexData[offset++] = (short) (y * yLength);
}

```
for (int x = 0; x < xLength; x++) {
// One part of the strip
heightMapIndexData[offset++] = (short) ((y * yLength) + x);
heightMapIndexData[offset++] = (short) (((y + 1) * yLength) + x);
}

if (y < yLength – 2) {
// Degenerate end: repeat last vertex
heightMapIndexData[offset++] = (short) (((y + 1) * yLength) + (xLength – 1));
}
//———————————————–
```

in some places instead of yLength must be xLength:

```
//———————————————–
for (int y = 0; y 0) {
// Degenerate begin: repeat first vertex
heightMapIndexData[offset++] = (short) (y * xLength);
}

for (int x = 0; x < xLength; x++) {
// One part of the strip
heightMapIndexData[offset++] = (short) ((y * xLength) + x);
heightMapIndexData[offset++] = (short) (((y + 1) * xLength) + x);
}

if (y < yLength – 2) {
// Degenerate end: repeat last vertex
heightMapIndexData[offset++] = (short) (((y + 1) * xLength) + (xLength – 1));
}
//———————————————–
```

Admin

February 22, 2013 at 5:34 pm

Interesting, I'll have to check this out. 🙂 I think you may be right but it's been a long time since I wrote this code.

Linda

February 22, 2013 at 8:29 pm

Love your tutorials. Really helpful! I am following it.

I implemented a horizontal 360 degree panorama viewer using OpenGL ES on Android.
The idea is:
Draw a cylinder.
Bind the flat image on the cylinder as the texture;
Place the camera at the center of the cylinder.
Rotate the cylinder so we can see different orientation of the panorama view.

I have some ideas on improving performance (or maybe not necessary if the move is smooth?). Since I am new to OpenGL, and its version on Android, before working on it, want to ask for some insights from experts.
(a) Since the cylinder is always the same and the only change is with texture, I may use vertex buffer objects to draw the cylinder object.
From the tutorial on http://www.learnopengles.com/android-lesson-seven-an-introduction-to-vertex-buffer-objects-vbos/, it seems it can improve performance.

(b) Will using Index Buffer Objects (IBOs) improve performance?

Thanks in advance!

Admin

February 22, 2013 at 8:37 pm

Hi Linda,

The best way is to measure! You probably won't need an IBO unless you are reusing a lot of vertices; then it can make a difference. It's also a good practice to use VBOs since if you wanted to go to WebGL for example, you'd find that you need to use VBOs there.

Linda

February 23, 2013 at 1:51 am

Thank you so much!
Yes, I will use one piece of panorama video to test it, to see whether the move is smooth, or measure the time to see whether rendering is finished within one frame period.

Previously I used touch motion to rotate the cylinder (using ACTION_MOVE)
.
Now I added another function of motion sensor to serve the same purpose, i.e. when I move the mobile phone left, the cylinder will rotate rightwards, so I can see the partial image to the left. I use sensorManager TYPE_ACCELEROMETER.

I find that the movement through accelerometer sensor is very wild. I investigated the real-time acceleration values along the x axis, thinking the values are wild, not very making sense, and having a sever latency.

Have you met the similar problem (I just guess Android OpenGL developers might use the phone sensor functions sometimes)?

Thanks again! (Really appreciate your tutorials. I developed the Android panorama viewer just based on your tutorials.)

Admin

February 23, 2013 at 4:38 pm

Thanks for the kind compliments! 🙂 I'd be happy to link to your viewer in an upcoming roundup if it's something publicly accessible.

The phone sensors can definitely be a lot “noisier” than the touch input; even Google’s applications like Google Sky suffer from panning problems when you move the phone around to different areas of the sky. There are different ways you can cope with this, like buffering and smoothing the input. You might have read these following links already; if not, they can help out with an overview of how to use the different sensors:

http://developer.android.com/guide/topics/sensors/sensors_overview.html
http://developer.android.com/guide/topics/sensors/sensors_motion.html

This post might help you out with reducing the “jumpiness” of the input data: http://stackoverflow.com/questions/4611599/help-smoothing-data-from-a-sensor/

Hope this helps out. 🙂

Linda

February 23, 2013 at 6:39 pm

Thanks a lot! I am sorry my code is not well written, not wanting to displease readers with bad code. There are many other better written code with the similar function in panorama viewer. These two below are well written:
http://www.frank-durr.de/panodroid.html
http://code.google.com/p/panoramagl-android/

Yes, I have read the document on developer.android.com. I will try your suggestion on buffering and smoothing the input. I am thinking first to get the value curve versus the real movement, and then figure out how to adjust the value to curve to be close to the real movement. Thanks again for your great idea!

Linda

February 25, 2013 at 6:44 pm

In case it is helpful to others, I write down my finding:
I took a closer look at the acceleration values (first I just watched the constanly changing values on the screen, which is not clear), the values are file with some noise (the static values are x: -0.3, y: -0.3, z: 9.8):
If I move the mobile phone to the right, the x value will be:
-0.3 -0.3 2.0 -2.8 -0.3 -0.2 …
I think I should use the rotation sensor for my purpose, since if there is a positive acceleration value, there will be a negative accelration following.

Chibuzor

June 10, 2013 at 8:03 pm

I intend to get a copy of OpenGL Es 2 for Android ,I aim interested in developing an android App that will stream a panoramic stitched video in 360 deg View. will the book be sufficient any other guide will do.

Admin

June 10, 2013 at 9:31 pm

Thank you, I would really appreciate that! 🙂 The book will show you how to use OpenGL to setup a perspective projection and rotate/translate your triangles, which will help you get the right setup for the panorama. Your specific question goes beyond the scope of the book as it's more advanced, and it's not something that I've personally tried. As far as I know, if you can get the video frame decoded into a bitmap, then you can easily upload it to OpenGL. You'd just need to update that texture at the beginning of each frame, whenever the source data has changed. You might need to use the Jellybean APIs to get access to the video data; these links may help you out:

http://stackoverflow.com/questions/11321825/how-to-use-hardware-accelerated-video-decoding-on-android
http://developer.android.com/about/versions/android-4.1.html#Multimedia

---

Leszek

July 5, 2013 at 11:32 am

Hello! Let me first say this a fantastic resource, really helps me write my simple OpenGL game here. I have just ordered the book.

Let me point out one error in the code however. The generation of the vertex indices should really look like this

```
for (int y = 0; y < yLength – 1; y++) { if (y > 0) {
// Degenerate begin: repeat first vertex
heightMapIndexData[offset++] = (short) (y * xLength);
}

for (int x = 0; x < xLength; x++) {
// One part of the strip
heightMapIndexData[offset++] = (short) ((y * xLength) + x);
heightMapIndexData[offset++] = (short) (((y + 1) * xLength) + x);
}

if (y < yLength – 2) {
// Degenerate end: repeat last vertex
heightMapIndexData[offset++] = (short) (((y + 1) * xLength) + (xLength – 1));
}
}
```

(yLength has been replaced by xLength in 4 places). Your code only appears to work because in your example xLength==yLength.

Cheers!

Admin

July 8, 2013 at 10:11 pm

Thanks for pointing this out! I'll check it out and fix things up. Thank you as well for your book order, definitely appreciate that. 🙂

JD

November 29, 2013 at 7:14 am

Hi. I'm using this tutorial to create a terrain height map of a surface. Is there a way to properly create a new Height Map such as when the user clicks on a different location to model and then have it render that new heightmap. When I try creating a new HeightMap I get GL_INVALID_OPERATION.

Admin

December 9, 2013 at 8:34 pm

Hi JD,

Please try posting your question to StackOverflow with some code, it might help us narrow down on where things are going wrong.

Paul

December 10, 2014 at 3:05 am

Hi, I know this is a few years old but it still gets good google ranking so…

I think your description of the GL_TRIANGLE_STRIP is a bit wrong and it caused me some trouble. You indicate a strip of [v0, v1, v2, v3, v4] will be evaluated as [v0, v1, v2], [v1, v2, v3], … which if you think about it causes the second triangle to be CW (if the first was CCW). The spec actually says the order is [v0, v1, v2], [v2, v1, v3] which makes the strip a viable surface (ensures the normals all face the same direction)

The wikipedia page even makes special mention: *using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on.*

Paul

December 10, 2014 at 3:08 am

(oops… to continue my comment)
the complete wikipedia quote (actually taken from the OpenGL guide) is:

> *GL_TRIANGLE_STRIP draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface.*

Admin

January 5, 2015 at 4:01 pm

Hi Paul,

It's been a long time since I looked at this, but would this address your comment? "OpenGL does this by swapping the first two vertices of every even triangle (swapped vertices are in bold)"

---

klondike

December 31, 2014 at 12:54 am

Is it possible to use IBOs with separate Vertex Buffers instead of a single packed Vertex Buffer?

Admin

January 5, 2015 at 4:11 pm

I don't see why not as long as you are binding things properly for each call. As a disclaimer, I haven't tried it myself. 😛

Pingback: Ceiba3D Studio | Heightmap not rendering correctly

Paul

January 6, 2015 at 2:22 am

I suppose so, perhaps also saying '… to ensure all triangles are facing the same direction' or something. The only reason I bring it up is because as written it contradicted my intuition about how OpenGL worked, so caused confusion. If it just had a little note on this tweaking it would have saved some time looking up on other sites for clarification.

Thanks for the response

Saintriver

February 4, 2016 at 6:39 am

Hi Sir, i just want to say thank you for your great tutorials.
For the past two weeks, i have been studying and replicating your tutorial into my own model from lesson one to lesson eight here. Without your concise and straightforward tutorials, i think i will spent months learning these materials.
I want to purchase your book but unfortunately i am a bit tight on budget right now :(.

One question thought, do you have any suggestion of free resource where i can learn frame buffer object ? Thank you.