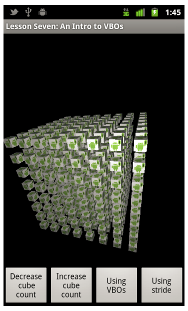
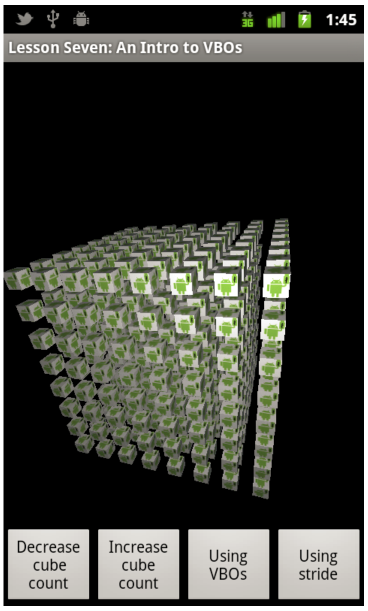
Learn OpenGL ES

Learn how to develop mobile graphics using OpenGL ES 2

Android Lesson Seven: An Introduction to Vertex Buffer Objects (VBOs)





In this lesson, we'll introduce vertex buffer objects (VBOs), how to define them, and how to use them. Here is what we are going to cover:

- How to define and render from vertex buffer objects.
- The difference between using a single buffer with all the data packed in, or multiple buffers.
- Problems and pitfalls, and what to do about them.

WHAT ARE VERTEX BUFFER OBJECTS, AND WHY USE THEM?

Up until now, all of our lessons have been storing our object data in client-side memory, only transferring it into the GPU at render time. This is fine when there is not a lot of data to transfer, but as our scenes get more complex with more objects and triangles, this can impose an extra cost on the CPU and memory usage. What can we do about this? We can use vertex buffer objects. Instead of transferring vertex information from client memory every frame, the information will be transferred *once* and rendering will then be done from this graphics memory cache.

ASSUMPTIONS AND PREREQUISITES

Please read Android Lesson One: Getting Started for an intro on how to upload the vertices from client-side memory. This understanding of how OpenGL ES works with the vertex arrays will be crucial to understanding this lesson.

UNDERSTANDING CLIENT-SIDE BUFFERS IN MORE DETAIL

Once you understand how to render using client-side memory, it's actually not too hard to switch to using VBOs. The main difference is that there is an additional step to upload the data into graphics memory, and an additional call to bind to this buffer when rendering.

This lesson has been setup to use four different modes:

- Client side, separate buffers.
- Client side, packed buffer.
- Vertex buffer object, separate buffers.
- Vertex buffer object, packed buffers.

Whether we are using vertex buffer objects or not, we need to first store our data in a client-side direct buffer. Recall from lesson one that OpenGL ES is a native system library, whereas Dalvik Java runs in a virtual machine. To bridge the gap, we need to use a set of special buffer classes to allocate memory on the native heap and make it accessible to OpenGL:

```
// Java array.
float[] cubePositions;
...
// Floating-point buffer
final FloatBuffer cubePositionsBuffer;
```

```
...

// Allocate a direct block of memory on the native heap,
// size in bytes is equal to cubePositions.length * BYTES_PER_FLOAT.
// BYTES_PER_FLOAT is equal to 4, since a float is 32-bits, or 4 bytes.
cubePositionsBuffer = ByteBuffer.allocateDirect(cubePositions.length *
BYTES_PER_FLOAT)

// Floats can be in big-endian or little-endian order.
// We want the same as the native platform.
.order(ByteOrder.nativeOrder())

// Give us a floating-point view on this byte buffer.
.asFloatBuffer();
```

Transferring data from the Java heap to the native heap is then a matter of a couple calls:

```
// Copy data from the Java heap to the native heap.
cubePositionsBuffer.put(cubePositions)

// Reset the buffer position to the beginning of the buffer.
.position(0);
```

What is the purpose of the buffer position? Normally, Java does not give us a way to specify arbitrary locations in memory using pointer arithmetic. However, setting the position of the buffer is functionally equivalent to changing the value of a pointer to a block of memory. By changing the position, we can pass arbitrary memory locations within our buffer to OpenGL calls. This will come in handy when we work with packed buffers.

Once the data is on the native heap, we no longer need to keep the float[] array around, and we can let the garbage collector clean it up.

Rendering with client-side buffers is straightforward to setup. We just need to enable using vertex arrays on that attribute, and pass a pointer to our data:

```
// Pass in the position information
GLES20.glEnableVertexAttribArray(mPositionHandle);
GLES20.glVertexAttribPointer(mPositionHandle, POSITION_DATA_SIZE,
        GLES20.GL_FLOAT, false, 0, mCubePositions);
```

Explanation of the parameters to glVertexAttribPointer:

- **mPositionHandle:** The OpenGL index of the position attribute of our shader program.
- **POSITION_DATA_SIZE:** How many elements (floats) define this attribute.
- **GL_FLOAT:** The type of each element.
- **false:** Should fixed-point data be normalized? Not applicable since we are using floating-point data.
- **0:** The stride. Set to 0 to mean that the positions should be read sequentially.
- **mCubePositions:** The pointer to our buffer, containing all of the positional data.

Working with packed buffers

Working with packed buffers is very similar, except that instead of using a buffer each for positions, normals, etc… one buffer will contain all of this data. The difference looks like this:

Using separate buffers

positions = X,Y,Z, X, Y, Z, X, Y, Z, …
colors = R, G, B, A, R, G, B, A, …
textureCoordinates = S, T, S, T, S, T, …

Using a packed buffer

buffer = X, Y, Z, R, G, B, A, S, T, …

The advantage to using packed buffers is that it should be more efficient for the GPU to render, since all of the information needed to render a triangle is located within the same block of memory. The disadvantage is that it may be more difficult and slower to update, if you are using dynamic data.

When we use packed buffers, we need to change our rendering calls in a couple of ways. First, we need to tell OpenGL the *stride*, or how many bytes define a vertex.

```
final int stride = (POSITION_DATA_SIZE + NORMAL_DATA_SIZE +
TEXTURE_COORDINATE_DATA_SIZE)
        * BYTES_PER_FLOAT;

// Pass in the position information
mCubeBuffer.position(0);
GLES20.glEnableVertexAttribArray(mPositionHandle);
GLES20.glVertexAttribPointer(mPositionHandle, POSITION_DATA_SIZE,
        GLES20.GL_FLOAT, false, stride, mCubeBuffer);

// Pass in the normal information
mCubeBuffer.position(POSITION_DATA_SIZE);
GLES20.glEnableVertexAttribArray(mNormalHandle);
GLES20.glVertexAttribPointer(mNormalHandle, NORMAL_DATA_SIZE,
        GLES20.GL_FLOAT, false, stride, mCubeBuffer);
...
```

The stride tells OpenGL ES how far it needs to go to find the same attribute for the next vertex. For example, if element 0 is the beginning of the position for the first vertex, and there are 8 elements per vertex, then the stride will be equal to 8 elements, or 32 bytes. The position for the next vertex will be found at element 8, and the next vertex after that at element 16, and so on.

Keep in mind that the value of the stride passed to glVertexAttribPointer should be in *bytes*, not elements, so remember to do that conversion.

Notice that we also change the start position of the buffer when we switch from specifying the positions to the normals. This is the pointer arithmetic I was referring to before, and this is how we can do it in Java when working with OpenGL ES. We're still working with the same buffer, mCubeBuffer, but we tell OpenGL to start reading in the normals at the first element after the position.

Again, we pass in the stride to tell OpenGL that the next normal will be found 8 elements or 32 bytes later.

Dalvik and memory on the native heap

If you allocate a lot of memory on the native heap and release it, you will probably run into the beloved OutOfMemoryError, sooner or later. There are a couple of reasons behind that:

1. You might think that you've released the memory by letting the reference go out of scope, but native memory seems to take a few extra GC cycles to be completely cleaned up, and Dalvik will throw an exception if there is not enough free memory available and the native memory has not yet been released.
2. The native heap can become fragmented. Calls to allocateDirect() will inexplicably fail, even though there appears to be plenty of memory available. Sometimes it helps to make a smaller allocation, free it, and then try the larger allocation again.

What can you do about these problems? Not much, other than hoping that Google improves the behaviour of Dalvik in future editions (they've added a *largeHeap* parameter to 3.0+), or manage the heap yourself by doing your allocations in native code or allocating a huge block upfront, and spinning off buffers based off of that.

Moving to vertex buffer objects

Now that we've reviewed working with client-side buffers, let's move on to vertex buffer objects! First, we need to review a few very important points:

1. Buffers must be created within a valid OpenGL context.

This might seem like an obvious point, but it's just a reminder that you have to wait until onSurfaceCreated(), and you have to take care that the OpenGL ES calls are done on the GL thread. See this document: OpenGL ES Programming

Guide for iOS. It might be written for iOS, but the behaviour of OpenGL ES is similar on Android.

2. Improper use of vertex buffer objects will crash the graphics driver.

You need to be careful with the data you pass around when you use vertex buffer objects. Improper values will cause a native crash in the OpenGL ES system library or in the graphics driver library. On my Nexus S, some games freeze up my phone completely or cause it to reboot, because the graphics driver is crashing on their commands. Not all crashes will lock up your device, but at a minimum you will *not* see the "This application has stopped working" dialogue. Your activity will restart without warning, and the only info you'll get might be a native debug trace in the logs.

3. The OpenGL ES bindings are broken on Froyo (2.2), and incomplete/unavailable in earlier versions.

This is the most unfortunate and most important point to consider. For some reason, Google really dropped the ball when it comes to OpenGL ES 2 support on Froyo. The mappings are incomplete, and several crucial functions needed to use vertex buffer objects are unavailable and cannot be used from Java code, at least with the standard SDK.

I don't know if it's because they didn't run their unit tests, or if the developer was sloppy with their code generation tools, or if everyone was on 8 cups of coffee and burning the midnight oil to get things out the door. I don't know why the API is broken, but the fact is that it's broken.

There are three solutions to this problem:

1. Target Gingerbread (2.3) and higher.
2. Don't use vertex buffer objects.
3. Use your own Java Native Interface (JNI) library to interface with the native OpenGL ES system libraries.

I find option 1 to be unacceptable, since a full quarter of devices out there still run on Froyo as of the time of this writing. Option 2 works, but is kind of silly.

The option I recommend, and that I have decided to go with, is to use your own JNI bindings. For this lesson I have decided to go with the bindings generously provided by the guys who created libgdx, a cross-platform game development library licensed under the Apache License 2.0. You need to use the following files to make it work:

- /libs/armeabi/libandroidgl20.so
- /libs/armeabi-v7a/libandroidgl20.so
- src/com/badlogic/gdx/backends/android/AndroidGL20.java
- src/com/badlogic/gdx/graphics/GL20.java
- src/com/badlogic/gdx/graphics/GLCommon.java

You might notice that this excludes Android platforms that do not run on ARM, and you'd be right. It would probably be possible to compile your own bindings for those platforms if you want to have VBO support on Froyo, though that is out of the scope of this lesson.

Using the bindings is as simple as these lines of code:

```
AndroidGL20 mGlEs20 = new AndroidGL20();
...
mGlEs20.glVertexAttribPointer(mPositionHandle, POSITION_DATA_SIZE,
GLES20.GL_FLOAT, false, 0, 0);
...
```

You only need to call the custom binding where the SDK-provided binding is incomplete. I use the custom bindings to fill in the holes where the official one is missing functions.

Uploading vertex data to the GPU.

To upload data to the GPU, we need to follow the same steps in creating a client-side buffer as before:

```
...
cubePositionsBuffer = ByteBuffer.allocateDirect(cubePositions.length *
BYTES_PER_FLOAT)
.order(ByteOrder.nativeOrder()).asFloatBuffer();
cubePositionsBuffer.put(cubePositions).position(0);
...
```

Once we have the client-side buffer, we can create a vertex buffer object and upload data from client memory to the GPU with the following commands:

```
// First, generate as many buffers as we need.
// This will give us the OpenGL handles for these buffers.
final int buffers[] = new int[3];
GLES20.glGenBuffers(3, buffers, 0);

// Bind to the buffer. Future commands will affect this buffer
specifically.
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, buffers[0]);

// Transfer data from client memory to the buffer.
// We can release the client memory after this call.
GLES20.glBufferData(GLES20.GL_ARRAY_BUFFER,
cubePositionsBuffer.capacity() * BYTES_PER_FLOAT,
        cubePositionsBuffer, GLES20.GL_STATIC_DRAW);

// IMPORTANT: Unbind from the buffer when we're done with it.
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, 0);
```

Once data has been uploaded to OpenGL ES, we can release the client-side memory as we no longer need to keep it around. Here is an explanation of glBufferData:

- **GL_ARRAY_BUFFER:** This buffer contains an array of vertex data.
- **cubePositionsBuffer.capacity() * BYTES_PER_FLOAT:** The number of bytes this buffer should contain.
- **cubePositionsBuffer:** The source that will be copied to this vertex buffer object.
- **GL_STATIC_DRAW:** The buffer will *not* be updated dynamically.

Our call to glVertexAttribPointer looks a little bit different, as the last parameter is now an offset rather than a pointer to our client-side memory:

```
// Pass in the position information
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mCubePositionsBufferIdx);
GLES20.glEnableVertexAttribArray(mPositionHandle);
mGlEs20.glVertexAttribPointer(mPositionHandle, POSITION_DATA_SIZE,
GLES20.GL_FLOAT, false, 0, 0);
...
```

Like before, we bind to the buffer, then enable the vertex array. Since the buffer is already bound, we only need to tell OpenGL the offset to start at when reading from the buffer. Since we are using separate buffers, we pass in an offset of 0. Notice also that we are using our custom binding to call glVertexAttribPointer, since the official SDK is missing this specific function call.

Once we are done drawing with our buffer, we should unbind from it:

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

When we no longer want to keep our buffers around, we can free the memory:

```
final int[] buffersToDelete = new int[] { mCubePositionsBufferIdx,
mCubeNormalsBufferIdx,
        mCubeTexCoordsBufferIdx };
GLES20.glDeleteBuffers(buffersToDelete.length, buffersToDelete, 0);
```

Packed vertex buffer objects

We can also use a single, packed vertex buffer object to hold all of our vertex data. The creation of a packed buffer is the same as above, with the only difference being that we start from a packed client-side buffer. Rendering from the packed buffer is also the same, except we need to pass in a stride and an offset, like when using packed buffers in client-side memory:

```
final int stride = (POSITION_DATA_SIZE + NORMAL_DATA_SIZE +
TEXTURE_COORDINATE_DATA_SIZE)
        * BYTES_PER_FLOAT;

// Pass in the position information
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mCubeBufferIdx);
GLES20.glEnableVertexAttribArray(mPositionHandle);
mGlEs20.glVertexAttribPointer(mPositionHandle, POSITION_DATA_SIZE,
        GLES20.GL_FLOAT, false, stride, 0);

// Pass in the normal information
GLES20.glBindBuffer(GLES20.GL_ARRAY_BUFFER, mCubeBufferIdx);
GLES20.glEnableVertexAttribArray(mNormalHandle);
mGlEs20.glVertexAttribPointer(mNormalHandle, NORMAL_DATA_SIZE,
        GLES20.GL_FLOAT, false, stride, POSITION_DATA_SIZE *
BYTES_PER_FLOAT);
...
```

Notice that the offset needs to be specified in *bytes*. The same considerations of unbinding and deleting the buffer apply, as before.

Putting it all together

This lesson is setup so that it builds a cube of cubes, with the same number of cubes in each dimension. It will build a cube of cubes between 1x1x1 cubes, and 16x16x16 cubes. Since each cube shares the same normal and texture data, this data will be copied repeatedly when we initialize our client-side buffer. All of the cubes will end up inside the same buffer objects.

You can view the code for the lesson and view an example of rendering with and without VBOs, and with and without packed buffers. Check the code to see how some of the following was handled:

- Posting events from the OpenGL thread back to the main UI thread, via runOnUiThread.
- Generating the vertex data asynchronously.
- Handling out of memory errors.

- We removed the call to glEnable(GL_TEXTURE_2D), since that is actually an invalid enum on OpenGL ES 2. This is a hold over from the fixed pipeline days; In OpenGL ES 2 this stuff is handled by shaders, so no need to use a glEnable/glDisable.
- How to render using different paths, without adding too many if statements and conditions.

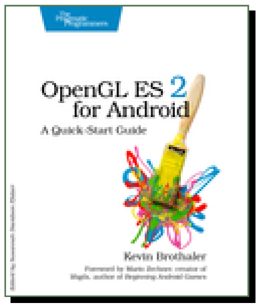
Further exercises

When would you use vertex buffers and when is it better to stream data from client memory? What are some of the drawbacks of using vertex buffer objects? How would you improve the asynchronous loading code?

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. A special thanks goes out again to the guys at libgdx for generously providing the source code and libraries for their OpenGL ES 2 bindings for Android 2.2!



Buy Now

It's never been a better time to learn how to create your own 3D games and live wallpapers. If you can program in Java and you have a creative vision that you'd like to share with the world, then this is the book for you.

Share / Save ...

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 / March 7, 2012 / Android, Android Tutorials / Dalvik, endianness, Froyo, Gingerbread, heap fragmentation, memory management, native heap, Nexus S, pointer arithmetic, recycling, VBOs, Vertex Buffer Object

54 thoughts on "Android Lesson Seven: An Introduction to Vertex Buffer Objects (VBOs)"

Android Research

March 7, 2012 at 2:52 pm

Always was afraid of starting Android Graphics, have to give it a try with your tutorials.

Admin

March 9, 2012 at 3:42 pm

You definitely should. Some pretty neat tutorials over at your site!

Shailendra

March 10, 2012 at 4:48 am

HI
Is there any process for Importing 3D .Obj file. Need to know for game development. In my case the Box models are showing nicely but when object becomes complex it shows problems. Main problem is Mesh looks distorted/Misplaces Vertices/Missing triangles etc. how to over come this problem?? plzz help

Admin

March 10, 2012 at 4:52 am

I have no idea what could be causing that, but it sounds like it's a problem either with generation of the vertex arrays / vertex buffer objects or with loading the data. You could try some of these tutorials:

http://www.bayninestudios.com/2011/01/importing-3d-models-in-android/
http://stackoverflow.com/questions/204363/is-there-a-way-to-import-a-3d-model-into-android

Stack Overflow may be able to help if you have some sample code and models to show. Hope this helps out!

Pingback: OpenGL ES Roundup, March 9, 2012 | Learn OpenGL ES

Martin

March 16, 2012 at 9:38 am

Thank you very much , but What's the next tutorial on?
There are so many topics like skinning, 3D modelfiles, native C/C++, realistic water rendering by using shaders, shadows, collission detection and terrain rendering.

Admin

March 16, 2012 at 1:38 pm

That's right! Tough to pick a next topic. There is still a lot of ground to cover. 🙂

Martin

March 16, 2012 at 9:55 am

I don't understand, why haven't you used indices to reduce the vector and color stuff

Martin

March 16, 2012 at 9:57 am

Little grammar mistake:
I don't understand, why you haven't use indices to reduce the vector and color stuff.

Admin

March 16, 2012 at 1:41 pm

You're right; I could have used them, but I figured there was enough in this lesson that I didn't need to throw on something else. Besides, it's debatable whether you always need to use index buffer arrays. You could always use triangle strips. 😉

I should do a benchmark between the different combos (VBOs versus vertex arrays, IBOs, with and without separate buffers for the different components, and with/without triangle strips). Maybe the next tutorial could take a look at that.

Martin

March 18, 2012 at 9:44 am

Isn't it possible to use both(IBOs and VBOs) together?

Admin

March 19, 2012 at 1:26 am

Yes of course, sorry if my reply was confusing. I'll go more into detail in a subsequent tutorial.

Jayce

May 3, 2012 at 3:34 am

Thanks a lot for these tutorials. Learned so much from them. And when's the next tutorial coming? Waiting for it.

Admin

May 3, 2012 at 3:56 pm

Working on a book actually! Hopefully I can tie in some future tutorials with that.

Pingback: Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs) | Learn OpenGL ES

Ray Ingles

May 30, 2012 at 4:26 pm

Interestingly, this lesson fails on my original Droid if I turn it to "Not using VBOs" and any redraw occurs; the activity aborts and returns to the main menu.

Everything else works – and renders quite snappily, even up to a 16×16 array of cubes.

Admin

June 6, 2012 at 2:19 am

Interesting, I wonder why that happens. Could be a bug in my code or in the Droid's drivers 😉

blubee

July 18, 2012 at 7:07 am

Hey thanks for the amazing tutorials, i've been following keenly. I have a question that I am not all to sure how to solve.

If I am working with just vanilla android + the JNI bindings to fix froyo gl issues. How do I actually load blender .obj files in android?

I would prefer to stay away from other frameworks and work with just android if that's even possible, I hope someone knows. Again I prefer not to use other libraries, I just would like to export the blender .obj file and load it in android, thanks.

Admin

July 18, 2012 at 9:17 pm

Hi blubee,

I'm curious myself! Do you think it's time I start a forums here, or try to integrate with stack exchange or something? Would be nice to have a discussions forums.

Rene van der Lende

November 20, 2012 at 1:53 pm

HI,

Great tuts!! I'm messing a lot with OpenGL currently and use your code frequently for reference. At the moment i'm trying to grasp using TextureView instead of SurfaceView, seems easy but I can't get my head around it. Always the same (google) example on the net.

Request: When/If you've got time, could you convert lesson 7 to use TextureView? Or do a lesson on the subject? Would be a great help…

Thanks!

Admin

November 23, 2012 at 5:16 pm

Thanks Rene! I haven't used those myself but I think it would be a good idea for a future lesson!

Faizan

December 6, 2012 at 2:32 am

Hi,

Just one very urgent ans quick question, Please tell me why this "36" for the for loops in the source code for lesson 7.

For instance in this loop:

for (int i = 0; i < generatedCubeFactor * generatedCubeFactor * generatedCubeFactor; i++) {
for (int v = 0; v < 36; v++) {
cubeBuffer.put(cubePositions, cubePositionOffset, POSITION_DATA_SIZE);
cubePositionOffset += POSITION_DATA_SIZE;

cubeBuffer.put(cubeNormals, cubeNormalOffset, NORMAL_DATA_SIZE);
cubeNormalOffset += NORMAL_DATA_SIZE;
cubeBuffer.put(cubeTextureCoordinates, cubeTextureOffset,
TEXTURE_COORDINATE_DATA_SIZE);
cubeTextureOffset += TEXTURE_COORDINATE_DATA_SIZE;
}

The normal and texture data is repeated for each cube.
cubeNormalOffset = 0;
cubeTextureOffset = 0;
}

Admin

December 6, 2012 at 3:41 am

It’s been a while since I took a look at this, but I’m guessing it’s due to 6 vertices per cube face, 2 triangles with 3 per triangle.

Alfonso

January 14, 2013 at 9:31 pm

I found a curious case: increasing cube count to max and wiw VBOs + stride, when you rotate the big cube (cube of cubes) 180 grades to any direction, performance drops by 50%. For my tablet ASUS TF101 in portrait mode I get 44fps for no rotation and 20fps when made a 180 grade rotation for any direction. Any ideas?

Deduu

April 4, 2013 at 3:54 am

Hi, Thanks for this great tutorial, I’m still reading and learning this subject and walk through your codes. By the way, what is the smallest size of cube that OpenGL can handle? I tried to increment the number of cubes using your app,and the maximum cubes I got were 16 on each axis, means the maximum size of each cubes was 0,125 x 0,125 unit scene. Thanks

---

Rene van der Lende

April 4, 2013 at 11:30 am

Hi Deduu, OpenGL can handle many 0000's behind the comma, so you could create more cubes than your processor AND your eyes can handle. But that is not the way to go:

Create a default cube of 1x1x1 and SCALE it to the size you need. A scale of 10% would give a cube of 0,1×0,1x,1. OpenGL doesn't care about specific units like inches or centimeters. It is you who gives meaning to the units used. When you decide that the default cube is in centimeters, then 10% scale means 0,1cm.

Essentially, you can have a real scale model of a Boeing 747 and keep zooming in to the smallest bolt. Scale the model down to the width of the screen and you still can zoom in to the same bolt…

Cheers, Rene

Uwe

April 9, 2013 at 7:38 pm

Hi,
great tutorials. Thanks a lot!
I am a OpenGL beginner and have a question.
In the Sourcecode i found the following lines:

Matrix.multiplyMM(mvpMatrix, 0, viewMatrix, 0, modelMatrix, 0);

// Pass in the modelview matrix.
GLES20.glUniformMatrix4fv(mvMatrixUniform, 1, false, mvpMatrix, 0);

```
// This multiplies the modelview matrix by the projection matrix,
// and stores the result in the MVP matrix
// (which now contains model * view * projection).
Matrix.multiplyMM(temporaryMatrix, 0, projectionMatrix, 0, mvpMatrix, 0);
System.arraycopy(temporaryMatrix, 0, mvpMatrix, 0, 16);

// Pass in the combined matrix.
GLES20.glUniformMatrix4fv(mvpMatrixUniform, 1, false, mvpMatrix, 0);
```

Is it necessary to call the “glUniformMatrix4fv” function twice?
Wouldn’t it be enough to call it only once after the second Matrix-multiplication?

Regards
Uwe

Admin

April 17, 2013 at 2:05 am

Hi Uwe,

In the first case we update the “model view” matrix, whereas in the second case we update the “model view projection” matrix. That’s why we have two calls to glUniformMatrix4fv(). Hope this makes sense! 🙂

Deduu

April 17, 2013 at 8:09 am

Hi Admin, do you have c++ version code for lesson 7? I’ve tried to implement part of your code in c++ myself but facing problem of how to save and manage the number vertices updating, and pass those values into glVertexAttribPointer.

Thanks

Admin

April 19, 2013 at 3:02 am

I haven't done a C++ version yet. Maybe you could try with the NDK samples as a base?

Deduu

May 11, 2013 at 3:56 pm

Hi,

I would like to ask that in this sample code the number of triangle will increases as the number of cube increases. How to calculate the limit of our chip can handle toward the number of triangle regardless of rendering speed? Is there any way to draw the cube more efficiently (with less number of triangle) in case of limited memory capacity..

Thanks

Msh

June 26, 2013 at 12:03 am

Thanks for a greate tutorial
But if I want to change the data frequently to be displayed , what modifications should I do ?
Thanks

Admin

July 8, 2013 at 10:04 pm

For frequent modification of the data, you could set the buffer type to GL_STREAM_DRAW and then use glBufferSubData to upload the new data.

Cawfree

July 1, 2013 at 6:39 am

I can’t express how thankful I am for you creating these tutorials. The Android OpenGL Frameworks looks so complex from the outset, but you manage to explain these concepts so succinctly. The example source code is fantastic too.

Thanks for everything.

Admin

July 8, 2013 at 10:08 pm

Glad that it helped you out! 🙂

štěty

August 22, 2013 at 8:03 am

// First, generate as many buffers as we need.
// This will give us the OpenGL handles for these buffers.
final int buffers[] = new int[3];
....
cubeBuffer.limit(0);
cubeBuffer = null;

We should not have to empty the “buffers“?
For example: buffers = null;

Admin

November 4, 2013 at 5:59 pm

Sorry, this is a late reply… the int buffers[3] just holds the OpenGL handles for the buffers and only takes up 12 bytes and will be garbage collected, so there is no need to empty it out. The important thing is that the native buffers would be cleaned up as well as the GPU VBOs.

Chris

October 30, 2013 at 8:22 am

Hi,

Thanks for these tutorials, they've been great for getting me started.

I've run into one problem that maybe someone could help me with. I can create VBOs and render my model to screen successfully, but I'd like to create an animation by updating the vertex coordinates in a loop, and re-render the model in each iteration of a loop, i.e. like this:

for(int i=0; i<10; i++) {

mRenderer.updateFrame(i);
requestRender();
}

where updateFrame() updates the VBOs using glBufferSubData().

I'm running this loop in the onTouchEvent() method of my GLSurfaceView when the user touches the screen with four fingers. Because the VBOs need to be updated on the OpenGL thread I enclosed the loop like this:

this.queueEvent(new Runnable() {

@Override
public void run() {

for(int i=0; i<10; i++) {

mRenderer.updateFrame(i);
requestRender();

```
}
}

});
```

I would expect the model to be re-rendered in each iteration of the loop, but it only gets updated after the last iteration (which at least shows that the VBOs are getting updated).

I can't figure out what I'm doing wrong. Perhaps it's something to do with calling requestRender() within the openGL thread?

Any help appreciated.

Admin

November 4, 2013 at 5:48 pm

Hi Chris,

I'm guessing it's because requestRenderer() is queued and the first requestRender() doesn't fire until after the last updateFrame. For this to work you'd need to wait for the render to complete and then call updateFrame only after each frame.

What you could do is use a state machine to put the renderer into an "animation state", then put it into continuous rendering mode. Something like this:

```
/// In the renderer
interface OnAnimationCompleteListener {
void onAnimationComplete();
}

boolean isAnimating = false;
int animationCounter = 0;
OnAnimationCompleteListener listener = null;
```

```
void beginAnimation(OnAnimationCompleteListener listener) {
this.isAnimating = true;
this.animationCounter = 0;
this.listener = listener;
}

@Override
public void onDrawFrame(GL10 unused) {
// ...
if (this.isAnimating) {
this.updateFrame(animationCounter++);

if (animationCounter >= 10) {
this.isAnimating = false;

if (listener != null) {
listener.onAnimationComplete();
}
}
}
}

/// Your activity
Activity implements Renderer.OnAnimationCompleteListener

// ...
// In your onTouchEvent
this.queueEvent(new Runnable() {
@Override
public void run() {
mRenderer.beginAnimation(this);
mGlSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOU
SLY);
}
}
```

```
@Override
public void onAnimationComplete() {
mGlSurfaceView.queueEvent() {
@Override
public void run() {
mGlSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRT
Y);
}
}
}
```

There are also other approaches possible.

Chris

November 5, 2013 at 9:32 pm

Thanks, that put me on the right path.

I implemented the OnAnimationCompleteListener in my GLSurfaceView rather than an Activity and I also removed the queueEvent bits as it wouldn't compile with them in, but I don't think they're needed anyway as the call to beginAnimation() is quick enough to be run on the UI thread.

Thanks again.

Admin

November 8, 2013 at 1:41 pm

Hi Chris, glad that this helped you out. 🙂 I would personally keep the queueEvent in as it's really more for running things on the right thread rather than performance. I did make a mistake though and you'd probably want to do it more like this (where post() runs something on the UI thread and queueEvent runs something on the renderer thread):

```
// In your onTouchEvent
this.queueEvent(new Runnable() {
@Override
public void run() {
mRenderer.beginAnimation(this);
// Replace GLSurfaceView with the name of your custom GLSurfaceView
GLSurfaceView.this.post(new Runnable() {
@Override
public void run() {
mGlSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOU
SLY);
}
});
}
}

@Override
public void onAnimationComplete() {
post(new Runnable() {
@Override
public void run() {
mGlSurfaceView.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRT
Y);
}
}
}
```

Gerardo

June 17, 2014 at 6:47 pm

Hi there, I have been practising with your lessons and I have a little question, If I wanted to implement a flat hexagon tiles map, would I make the tile by giving the coordinates of all the hexagons in the buffer or by drawing a lot of hexagons and then translating them? Whats the most practical option?

Regards

Admin

June 17, 2014 at 7:57 pm

Hi Gerardo, the simple way is probably to draw a lot of hexagons and translate them. I'd get it working this way, first. If you see that it's not drawing fast enough and the bottleneck is your draw calls, then you can try batching it.

To batch it, you can generate all of the triangles for all the hexagons, put it into a VBO, then draw them in one shot. The way to do that is similar to how the height map is done here: http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/

For example, if you need one shader, then you generate all of the triangles for the hexagons with the right coordinates and put that into a VBO, which you can then draw with 1 draw call. If you need two shaders, then you can sort the hexagons into those that use the first shader, and those that use the second, then draw each separately.

This starts to get complicated, so I'd try the simple way first as it's easier to get going with that.

Fra

October 16, 2014 at 4:01 pm

Hi,

Great tutorials!

I am working on a terrain rendering program. Using a grid of heights and colours I create a surface. I turn the surface into 64×64 vertex tiles and upload each one as a single VBO. So I end up with a 2d array of VBO's representing my scene.
I am able to upload 40×40 tiles but can't do more. No real errors occur other than a simple crash of the program.

My initial though was that I am running out of GPU memory so I am checking for a GL_OUT_OF_MEMORY error after each call to glbufferdata. This error is never returned. I was also thinking that maybe I need to spawn a thread to do the uploading as maybe it takes a while and is messing with the main UI thread. Any ideas?
What are the upload limits in how much data you can put on the GPU? I would like to get a lot of high resolution tiles on the card then implement a LOD algorithm for drawing to keep a high FPS.

Any help would be HUGE!
thanks in advance!

Admin

October 16, 2014 at 5:49 pm

Hi Fra,

Hmm, what does the stack trace of the crash say? Are you able to pinpoint where in the code that it's crashing? It might help to step through or to use logging statements to print out what's happening.

Would you be able to post a question on Stack Overflow with some sample code? That might help to pinpoint what's going on.

Admin

October 16, 2014 at 5:54 pm

Let me know if this helps: https://groups.google.com/forum/#!topic/android-developers/U6GbozlcueI

Fra

October 16, 2014 at 6:47 pm

Admin,

Wow, thanks for the quick reply!
I have been trying to debug as best I can but the stack trace is really big. And no real error pertaining to Opengl seems to get thrown. I took a look at the StackOverflow post but I dont think that is where my problem lies. I can create 30×30 tiles at 4096 vertices each and the program runs flawlessly. It is when I get up around 40+ I start to get problems.
Other than checking glerror on each call to glbufferdata is there some way to know if I am overloading the GPU? Or could the fact that it takes a while to upload mean some UI thread interruptions are happening? I noticed in your code you use a thread to upload data.
I haven't worked much with VBO's or Opengl es so this is all new. I am trying to feel out the limits of what I can get away with. Is loading this much info (7+ million vertices) at 7bytes each so 50mb of data to the GPU too much? I though these things have up to 1GB ram. I would like to put a ton of data on the GPU then as I mentioned come up with my own LOD algorithm for deciding how much drawing I actually do….
I tidy up my code and get an example on here or StackOverflow to see if you pros can sort me out.

thanks again!

Admin

October 16, 2014 at 7:26 pm

Hi Fra,

The data should be uploaded on the GL thread, from inside one of the Renderer interface's callbacks. It's hard to say beyond that without seeing the specific code or the crash traces. 50MB is well within the limits of the device, but it might be blowing past Android's process-specific limits which are much lower than that. A post on StackOverflow would definitely help to get more details.

For a quick test you can add "android:largeHeap="true"" to the <application/> node of your AndroidManifest.xml.

Fra

October 16, 2014 at 7:31 pm

Hi Admin,

I posted code and my problem a bit more thoroughly on StackOverflow as you suggested:

https://stackoverflow.com/questions/26412374/limitations-of-loading-vbos-opengl-es

---

Fra

October 16, 2014 at 9:02 pm

I do have the largeHeap enabled. Wouldn't that be a separate issue though as all of the vertex data will be in VRAM? I don't get any out of memory errors. I have noticed though I can force some "Frames skipped" warnings as well as a "thread may be overloaded" warning….
I think it could have to do with the way I am loading the VBO's or possibly binding to them.
I'll stop taking up this comment thread though and contact you directly.

Seth Gibson

April 30, 2015 at 8:21 pm

So for this tutorial, are you basically just creating a giant buffer of interleaved data? I.e. if I wanted to draw a grid of 10x10x10 triangles, would I need to make my buffer hold 10x10x10x3 floats for position data, 10x10x10x3 floats for color data, and 10x10x10x2 floats for texcoord data, so 8000 elements total, or would I have one buffer that was 10x10x10x3 floats for transforms and one buffer that was 8×3 floats for one triangle worth of position, color, and texture data?

Pingback: [Android]OpenGL Study | Dion's Note

herve terrolle

March 13, 2016 at 11:16 am

Sorry for the comment i found out.

regards

twi

April 2, 2016 at 12:31 pm

How can you draw with multiple separate buffers ? (position buffer, normal buffer, color buffer …)
Knowing that you can only bind one unique buffer at a time.

As I know the only way to render using multiple buffers is to use VAOs (vertex array objects) which don't exist yet on opengl ES 2.0

So explaining separates buffers in this tutorial maybe a little bit pointless