***Introduction to Graphics and Mobile Gaming***

**LAB 8**

**Vertex Buffer Objects**

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc36110329)

[2 Generating buffers 1](#_Toc36110330)

[3 Altering the way we store data 2](#_Toc36110331)

[4 Altering the render function 3](#_Toc36110332)

# Introduction

As OpenGL ES is made for embedded systems such as phones, tablets, etc.; one of the most important things we must consider when designing our applications/graphics is bandwidth. This lab will look at “vertex buffer objects” which help reduce the usage of bandwidth within our application/graphics. Looking back at our previous examples, we have defined attributes in main memory and then uploaded them to the GPU every frame so that the scene can be rendered. The use of vertex buffer objects eliminates the need to upload the data to the GPU every frame, and instead uploads it all at once, and it stays there.

# Generating buffers

To start, we need to create some buffers that will hold our data. First, we define two arrays of integers that hold the IDs for our buffer objects: one of them for all the vertex-related data and one for the indices.

**static** GLuint vboBufferIds[2];

We now need to give the integers valid values because they have not yet been initialized. We will do this by adding a new function call into our *“setupGraphics”* function. The function needs to generate two buffers (similar to “glGenTextures” which is what we used in the textured cube tutorial). The first parameter for the function is how many buffers we wish to create and the second parameter is a pointer to the arrays we defined that contain the IDs of our buffer objects. You will be left to write this line of code.

/\* [vboCreation] \*/

glGenBuffers(2, vboBufferIds);

glBindBuffer(GL\_ARRAY\_BUFFER, vboBufferIds[0]);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, vboBufferIds[1]); /\* [vboCreation] \*/

Once this is done, we need to bind our buffers, associating the created IDs to appropriate targets. There are two different target types: “GL\_ARRAY\_BUFFER” and “GL\_ELEMENT\_ARRAY\_BUFFER”, which are used for vertices and indices, respectively. If we do not bind the buffers, OpenGL ES will assume you do not want to use vertex buffer objects and will expect the data to be uploaded each time it is required.

Once our buffers are bound, space needs to be allocated to them so it can be filled with data. This is achieved using the function *“glBufferData”* which takes 4 parameters: the target (same one as above), the size, the data, and how it will be used.

**glBufferData**(GL\_ARRAY\_BUFFER, vertexBufferSize, cubeVertices, GL\_STATIC\_DRAW);

**glBufferData**(GL\_ELEMENT\_ARRAY\_BUFFER, elementBufferSize, indices, GL\_STATIC\_DRAW);

You may notice a lot of these variables have not yet been defined, so we must also do that.

**static** GLushort vertexBufferSize = 48 \* 3 \* **sizeof** (GLfloat);

**static** GLushort elementBufferSize = 36 \* **sizeof**(GLushort);

We set *“vertexBufferSize*” to **48 \* (3 \* sizeof(GLfloat))** in order to give the size of the buffer we need in bytes. The reason for this is that each vertex position is defined by 3 GLfloat components (X, Y, Z). The number **48** arises due to us merging the **24 vertices** (4 vertex positions for each face of the cube) with the vertex colour data, which also has 4 values per cube face (24). This is done using “stride” which we will discuss later on.

Looking back to the *“glBufferData”* command, the final parameter tells OpenGL ES how you intend to use the buffer. There are three possible options that this parameter could be: GL\_STATIC\_DRAW, GL\_DYNAMIC\_DRAW, and GL\_STREAM\_DRAW. The first allows the data to only be modified once before it is used (this is the suitable option for us as our geometry will not change in the scene). The next option means that the data can be modified a number of times and then used a number of times. The final option allows it to be modified once and will only be used a few times.

Note that we use GL\_ELEMENT\_ARRAY\_BUFFER instead of GL\_ELEMENT\_BUFFER for the second call of *“glBufferData”* and also use different numbers to obtain a value for *“elementBufferSize”.* We use **36 \* sizeof(GLushort)** because we are drawing 12 triangles, each having 3 indices. We defined these indices as shorts.

# Altering the way we store data

We need to add all of our per vertex data into one vertex buffer object. We do this by interleaving the data we put into the array. For example, if we have vertex position and colour, we put the position values, then colour values, and repeat until all the data is entered. Here is the sample we will use for this tutorial:

**static** GLfloat cubeVertices[] = {-1.0f, 1.0f, -1.0f,/\* Back Face First Vertex Position \*/

1.0f, 0.0f, 0.0f, /\* Back Face First Vertex Colour \*/

1.0f, 1.0f, -1.0f, /\* Back Face Second Vertex Position \*/

1.0f, 0.0f, 0.0f, /\* Back Face Second Vertex Colour \*/

-1.0f, -1.0f, -1.0f,/\* Back Face Third Vertex Position \*/

1.0f, 0.0f, 0.0f, /\* Back Face Third Vertex Colour \*/

1.0f, -1.0f, -1.0f,/\* Back Face Fourth Vertex Position \*/

1.0f, 0.0f, 0.0f, /\* Back Face Fourth Vertex Colour \*/

-1.0f, 1.0f, 1.0f, /\* Front. \*/

0.0f, 1.0f, 0.0f,

1.0f, 1.0f, 1.0f,

0.0f, 1.0f, 0.0f,

-1.0f, -1.0f, 1.0f,

0.0f, 1.0f, 0.0f,

1.0f, -1.0f, 1.0f,

0.0f, 1.0f, 0.0f,

-1.0f, 1.0f, -1.0f, /\* Left. \*/

0.0f, 0.0f, 1.0f,

-1.0f, -1.0f, -1.0f,

0.0f, 0.0f, 1.0f,

-1.0f, -1.0f, 1.0f,

0.0f, 0.0f, 1.0f,

-1.0f, 1.0f, 1.0f,

0.0f, 0.0f, 1.0f,

1.0f, 1.0f, -1.0f, /\* Right. \*/

1.0f, 1.0f, 0.0f,

1.0f, -1.0f, -1.0f,

1.0f, 1.0f, 0.0f,

1.0f, -1.0f, 1.0f,

1.0f, 1.0f, 0.0f,

1.0f, 1.0f, 1.0f,

1.0f, 1.0f, 0.0f,

-1.0f, -1.0f, -1.0f, /\* Top. \*/

0.0f, 1.0f, 1.0f,

-1.0f, -1.0f, 1.0f,

0.0f, 1.0f, 1.0f,

1.0f, -1.0f, 1.0f,

0.0f, 1.0f, 1.0f,

1.0f, -1.0f, -1.0f,

0.0f, 1.0f, 1.0f,

-1.0f, 1.0f, -1.0f, /\* Bottom. \*/

1.0f, 0.0f, 1.0f,

-1.0f, 1.0f, 1.0f,

1.0f, 0.0f, 1.0f,

1.0f, 1.0f, 1.0f,

1.0f, 0.0f, 1.0f,

1.0f, 1.0f, -1.0f,

1.0f, 0.0f, 1.0f,

};

# Altering the render function

We now need to make some changes in the rendering function. Since we are now using vertex buffer objects, we do not need to send a pointer to our vertices as that data is not being used anymore. Instead, we need to change the value to an offset so it can find the data in our vertex buffer objects. Also, we need to let OpenGL know that the data is now interleaved in the array, so it can find what it needs.

**glVertexAttribPointer**(vertexLocation, 3, GL\_FLOAT, GL\_FALSE, strideLength, 0);

**glEnableVertexAttribArray**(vertexLocation);

**glVertexAttribPointer**(vertexColourLocation, 3, GL\_FLOAT, GL\_FALSE, strideLength, (**const**

**void** \*) vertexColourOffset);

**glEnableVertexAttribArray**(vertexColourLocation);

The first line of this code handles our vertex positions. We put 0 as the final parameter for the function call as we are aware that the first element of the VBO is a vertex element; hence, there is no need for the offset. If we look at the colour “glVertexAttribPointer” we can see that we provide the offset; this is because the first element is not a colour element.

**static** GLushort strideLength = 6 \* **sizeof**(GLfloat);

**static** GLushort vertexColourOffset = 3 \* **sizeof** (GLfloat);

The value of *“vertexColourOffset”* is set to **3 \* sizeof(GLfloat)** because we need to define the offset in bytes. Now there is a vertex position before the colour and each vertex position has 3 GLfloat components (X, Y, Z). You will also notice the variable *“strideLength”*; this is basically a number of bytes between the first element in the first component and the first element in the next component. As we can see in the definition, the value assigned to *“strideLength”* is **6 \* sizeof(GLfloat).** This is because if the value is not equal to 0, it should be the difference in bytes between the value of the first element in a component, to the first element in the next component (X Y Z R G B X Y Z). As we can see, there is a gap of six elements between the first element of component 1 and the first element of the other position component. The colour has the same stride length.

The last thing we need to do is alter our *“glDrawElements”* line.

**glDrawElements**(GL\_TRIANGLES, 36, GL\_UNSIGNED\_SHORT, 0);

We wish to draw every element from the beginning, so we set the offset (end parameter) to 0.