Vertex Buffers

GAME 300

James Dupuis

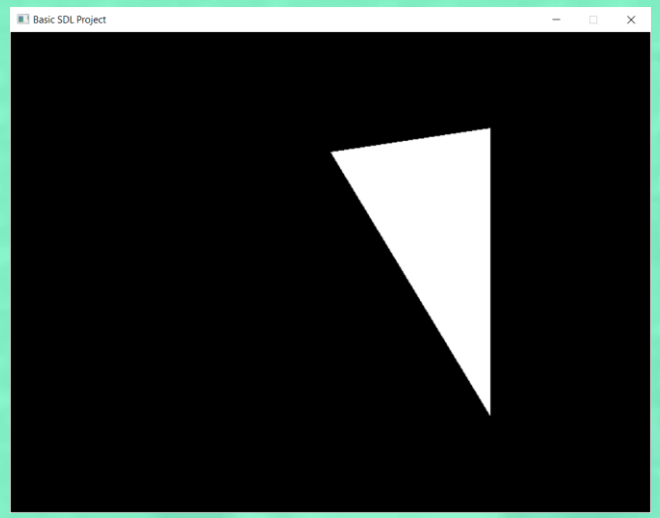
OBJECTIVES

- Learn to efficiently transfer data to our Vertex Shaders.
- Learn how to setup and use vertex buffers

RAM vs VRAM

- When we write a use immediate mode and define our vertices data by calling glVertex3f() we are storing our vertices inside of our RAM.
  - The same thing happens when we pass our data in using Vertex Arrays like so:

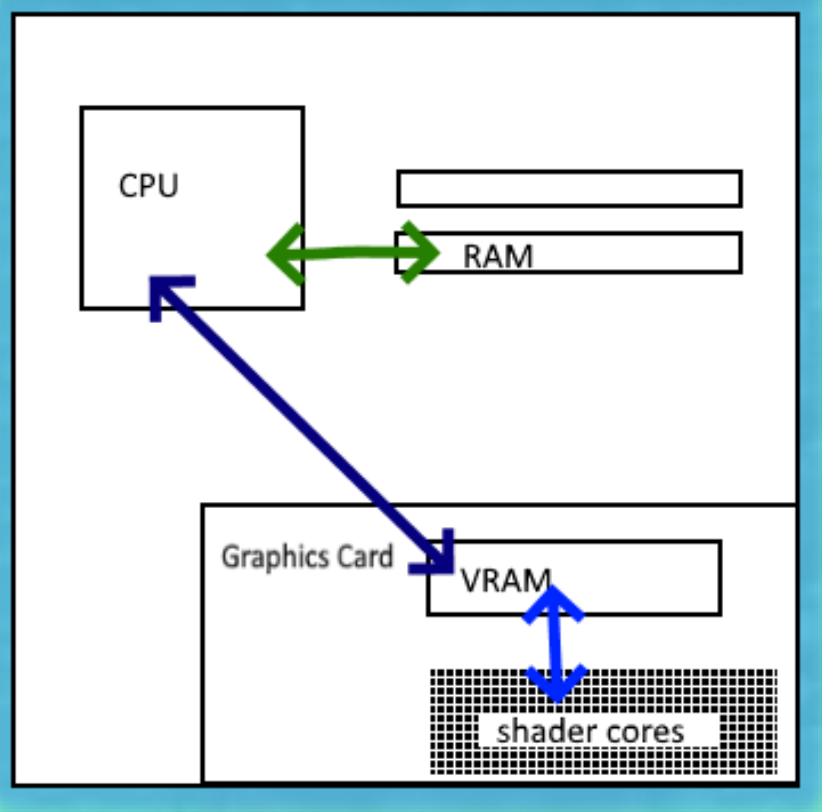
```
float triangleVertices[9] = {   0.0f, 0.5f, 0.0f,
                                0.5f, -0.6f, 0.0f,
                                0.5f, 0.6f, 0.0f, };

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, &triangleVertices[0]);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableClientState(GL_VERTEX_ARRAY);
```



  - Although it is more efficient as it saves on RAM consumed.
- It would be much more efficient if we had a way to cut out the middle man and directly pass data to our VRAM on our graphics card, wouldn’t it?

VERTEX BUFFERS

- The solution to this problem of data transfer comes in the form of Vertex Buffers.
- Using Vertex Buffers we can directly assign data to the VRAM instead of housing it within the OpenGL program located in RAM.
  - We can then modify it directly in VRAM and not interact with the RAM afterwards when we modify the data.
  - Vertex Buffers are often shortened to an acronym of VBO (Vertex Buffer Object)



STEPS INVOLVED

- Creating and dealing with Vertex Buffers requires a few more steps than immediate mode and vertex Arrays…
  - More overhead on us the programmer and less overhead on the GPU/CPU communications.
- The steps to create a VBO are:
  - Generate a name (or handle)
    - This can be performed once in your Init() functionality of the objects class
      - assuming the object is persistent throughout all gameplay.
  - Bind
    - This can also be done through the Initialization area of your object. It should only occur once.
  - Populate the Buffer with Data
    - This could possibly happen repeatedly so it's a good candidate for your update function.
  - Use the Data
    - We will most likely be using this data to forward along to OpenGL so this may be best used inside a render or draw call of the objects class.
  - Clean up the memory when finished.

WHATS IN A NAME?

- To generate a VBO handle, we will need to call the following function:

```
GLuint bufferHandle;

// generate a handle or name
glGenBuffers(1, &bufferHandle);
```

- First we create a GLuint to store the handle in.
- Then we pass it as a reference to the glGenBuffers function.
  - The glGenBuffers function establishes a buffer to setup on the OpenGL side of things and returns to us a handle for future interactions.
  - We can then use this moving forward any time we interact with the data OpenGL has stored for our buffer.

THE BINDING OF BUFFER

- Any time we want to use our buffer we must first bind it.
- Binding the buffer is similar to setting our states.
  - It let's openGL know that this is the current buffer we want to use.
  - We could have many different models loaded, each with their own buffer handle.
- To bind a buffer we can use the following:

```
// bind the buffer as the active buffer for OpenGL to modify
glBindBuffer(GL_ARRAY_BUFFER, bufferHandle);
```

- The two parameters passed here are the type of buffer and the handle to our buffer to be bound.

SUPPLYING DATA

- To Begin with we must have some sort of data to pass to the buffer on the Graphics Cards VRAM.
  - We can use a simple array of floats like we previously would have used to draw a triangle:

```
// should be defined inside the header file
float vertex[] = { -0.5f, 0.0f, -2.0f,
                    0.5f, 0.0f, -2.0f,
                    0.5f, 0.5f, -2.0f
};
```

- To Supply this array of data to our Buffer (after it's been bound) we can use the **glBufferData** function:

```
// insert the data from the data array into the buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), &vertex[0], GL_STATIC_DRAW);
```

SUPPLYING DATA

```
// insert the data from the data array into the buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), &vertex[0], GL_STATIC_DRAW);
```

- The parameters of the **glBufferData** function are:
  - The type of buffer (We are using an ARRAY of data for the vertices)
  - The size of the data being passed in (use sizeof here for accuracy)
  - A pointer to the starting location of the data (index 0 of the array)
  - The last param is the most complex, it can be one of the following:
    - This option is a flag to openGL so that it understands how the data being passed in will be used.
      - STREAM_ = used minimally
      - STATIC_ = used many times, but data remains somewhat constant
      - DYNAMIC_ = Data will be modified and accessed often.

```
#define GL_STREAM_DRAW 0x88E0
#define GL_STREAM_READ 0x88E1
#define GL_STREAM_COPY 0x88E2
#define GL_STATIC_DRAW 0x88E4
#define GL_STATIC_READ 0x88E5
#define GL_STATIC_COPY 0x88E6
#define GL_DYNAMIC_DRAW 0x88E8
#define GL_DYNAMIC_READ 0x88E9
#define GL_DYNAMIC_COPY 0x88EA
```

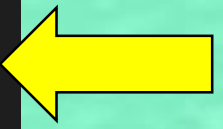
INITIALIZE BUFFER EXAMPLE

```
// should be defined inside the header file
float vertex[] = { -0.5f, 0.0f, -2.0f,
                    0.5f, 0.0f, -2.0f,
                    0.5f, 0.5f, -2.0f
};

GLuint bufferHandle;

// generate a handle or name
glGenBuffers(1, &bufferHandle);

// bind the buffer as the active buffer for OpenGL to modify
glBindBuffer(GL_ARRAY_BUFFER, bufferHandle);

// insert the data from the data array into the buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), &vertex[0], GL_STATIC_DRAW);
```

USING THE DATA

- When it's time to render our data as a shape we can do so similarly to how we did with vertex Arrays previously.
- In fact we technically are supplying our Vertex Buffer data as the data of the Vertex Array:

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, nullptr);
glDrawArrays(GL_TRIANGLES, 0, 3);
glDisableClientState(GL_VERTEX_ARRAY);
```



- Note that the last param of our glVertexPointer() function would typically point to our data.
  - We can instead replace it with a 0 or nullptr to indicate that it should use the data from the currently bound array.

FULL EXAMPLE

INIT: (do once)

1. Create an array of data.
2. Create a handle variable
3. Generate the handle
4. Bind the buffer
5. Set the data of the buffer and type

Update/Render: (do often)

1. Bind the buffer again to ensure it is the current one.
2. Draw using Vertex Array and setting the pointer to null.

```
void GameObject3D::Init()
{
    // should be defined inside the header file
    float vertex[] = { -0.5f, 0.0f, -2.0f,
                        0.5f, 0.0f, -2.0f,
                        0.5f, 0.5f, -2.0f
    };

    GLuint bufferHandle;

    // generate a handle or name
    glGenBuffers(1, &bufferHandle);

    // bind the buffer as the active buffer for OpenGL to modify
    glBindBuffer(GL_ARRAY_BUFFER, bufferHandle);

    // insert the data from the data array into the buffer
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertex), &vertex[0], GL_STATIC_DRAW);
}

// used to render the object to the screen
void GameObject3D::Draw()
{
    glPointSize(10.0f);

    glBindBuffer(GL_ARRAY_BUFFER, bufferHandle);

    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, nullptr);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glDisableClientState(GL_VERTEX_ARRAY);
}
```

SUMMARY

- We have discussed 3 ways of supplying OpenGL with data to render.
  - Immediate mode was the early ways of supplying data to our API and although easier to understand, it was less efficient.
  - Vertex Arrays were created to ensure data was not duplicated across vertices and memory was conserved as much as possible.
  - Last Buffers were created to allow the GPU to have direct access to the data for retrieval when rendering.
  - Together, these improvements allow us to communicate more data to the GPU and at a quicker rate.
    - This results in the ability to render more data, which in turn gives us the ability to create more realistic game scenes.