Lab 1 - OpenGL Basics

Intro
In this lab you will learn how to draw shapes using OpenGL. **OpenGL**, which stands for Open Graphics Library, is a cross-platform API\(^1\) that can let you create 2D and 3D graphics using the GPU. It has existed since 1992 but has undergone drastic changes over the years. Today it is an industry standard that is used in many applications like video games, CAD, and scientific visualization.

The labs for this course will start with the basics of OpenGL and work up to cool effects like a particle simulation, rendering metal and glass, and procedurally generating mountainous terrain.

When completing this lab you’ll notice that it takes a lot of work to draw something simple, like a triangle. In earlier versions of OpenGL this was easier to do, but it was difficult or impossible to render more complicated scenes/effects in real time. After taking the time to learn the newer OpenGL API, you will have the power to create really cool stuff in your final project!

A high-level diagram of the OpenGL pipeline is given below. For this lab we will mainly deal with the first part: sending vertex data to OpenGL. After a few more labs you’ll begin to understand the inner steps of the pipeline too.

The GPU
The **GPU**, which stands for Graphics Processing Unit, is a chip located on the graphics card that is specially designed to do graphics-related computations. Rendering using the GPU is ubiquitous in real-time graphics, and nowadays some other fields besides graphics are starting to use the GPU to speed up computations.

---

\(^1\) Technically OpenGL is really a *specification* for an API. Graphics cards typically have their own OpenGL *implementation* that implements the functionality in the OpenGL specification.
It is likely all the programs you have written so far have run on the CPU, or Central Processing Unit. One main difference between the two is parallelism: while the CPU has only a few cores with lots of cache memory, the GPU has hundreds or thousands of smaller cores that can be used for highly parallel tasks. Many algorithms have to do things one step at a time, but computer graphics has a lot of algorithms that are often called “embarrassingly parallel.” For example, computing the color of a pixel usually does not depend on the color of any other pixels, so they can all be computed in parallel.

OpenGL will act as your interface between the CPU and the GPU. In this lab you will write everything in C++, but next week you will start writing programs called shaders that run directly on the GPU.

OpenGL as a state machine
One important thing to know about OpenGL before getting started is that it acts as a state machine. OpenGL has a global state that can be changed in lots of different ways. For example, certain settings (e.g. GL_DEPTH_TEST, GL_BLEND) can be enabled or disabled using the functions glEnable and glDisable. Depending on OpenGL’s state, certain operations might behave very differently.

Because of this, it is important to be careful about managing OpenGL’s state during the execution of your programs. There are two basic strategies for this:

- **Assume nothing.** When writing a function that uses OpenGL, assume the state could be anything. Explicitly set the values of the state parameters you depend on.
- **Default configuration.** Decide on a set of reasonable default state values, and set the state to these values at the start of your application. Anywhere you change one of these state parameters, restore the parameter to its previous value after you’re done rendering a component of the scene.

Getting Started
The support code is located in /course/cs123/src/labs/lab01. Copy this directory into your course directory and open your copy of the provided .pro file with Qt Creator. From a terminal window:

```
    cs123_qtcreator *.pro
```

On the Configure Project screen, make sure Desktop is selected and select “Configure Project.”

Your job will be to fill in the OpenGLShape and GLWidget classes.

Note that there is also a demo version of the lab that can be run by inputting
/course/cs123/bin/cs123_lab01_demo into your terminal.
Vertex Buffer Objects

Shapes in OpenGL are composed of triangles, which are defined by three endpoints, or vertices. In order to draw a triangle, you have to send the vertex positions to the GPU.

OpenGL is able to store arrays of data on the GPU in something called a buffer object. When a buffer object contains vertex data (such as vertex positions), it’s called a vertex buffer object, or VBO.

Vertex positions (x, y, z) stored in a VBO

Go to OpenGLShape.cpp. This class will represent a triangle mesh that can be drawn using OpenGL. We want this class to contain a VBO that will store its triangles’ vertex positions. The first step is to create a new VBO.

**Task 1:**
*In OpenGLShape’s constructor, create a new VBO by calling glGenBuffers:*
  - Use 1 for the first argument. This indicates we only want to create one VBO.
  - For the second argument, pass a pointer to m_vboID. The ID of the newly created VBO will be stored in m_vboID. This will allow us to refer to this VBO later.

Now that a VBO has been created, we can send vertex data to it. This is done with a function called glBufferData. Because an OpenGL program might have many different buffer objects, we need a way to indicate which one we’d like to send data to. One way OpenGL could have done this would be to have glBufferData take the VBO’s ID as an additional argument. Instead, OpenGL uses the currently bound buffer object, which you can set using glBindBuffer.
To fill in buffer object #5 with vertex data, we first bind buffer object #5 using `glBindBuffer`. Then we call `glBufferData`, which stores the data in the currently bound buffer object. Finally, we unbind the buffer object by passing 0 to `glBindBuffer`. (Note that `glBindBuffer` takes an additional argument, omitted in this diagram.)

This is one example of what we mean when we say OpenGL is a state machine. Depending on the current state (i.e. the currently bound buffer object) `glBufferData` will do different things.

**Task 2:**
In `OpenGLShape::setVertexData(...)`, send the “data” array to the VBO you created in Task 1.

Things to keep in mind:
- When calling `glBindBuffer` and `glBufferData`, use `GL_ARRAY_BUFFER` as the target. This indicates that the buffer object will store vertex attributes (discussed in next section).
- Use `GL_STATIC_DRAW` for the last parameter of `glBufferData`. This indicates that we do not expect the vertex data to modified, allowing OpenGL to optimize for that.
- Don’t forget to unbind the VBO after calling `glBufferData`, by passing 0 to `glBindBuffer`. Remember: it is good practice to undo any changes you make to OpenGL’s state.

**Vertex Array Objects**
Now we are able to send vertex positions to OpenGL, but we still have a few more steps before our `OpenGLShape` class can draw this data. Specifically, we need to let OpenGL know that the VBO will be storing vertex positions! In this lab we will only be worried about positions, but in
larger programs each vertex could have lots of other attributes that we'd want to store in a VBO (color, normal vector, etc.).

In order to associate attributes with the data in the VBO, we need something called a vertex array object, or VAO. This is a pretty confusing name, as it’s not actually an array of vertex data. (That's the VBO!) The VAO is an object that knows how to draw the data in a VBO because it knows what attributes are stored in it.

Task 3: In OpenGLShape’s constructor, generate a new VAO using glGenVertexArrays, passing in a pointer to m_vaoID.

Now we can fill in OpenGLShape’s setAttribute method, which tells the VAO how an attribute is stored in the VBO.

Task 4: In OpenGLShape::setAttribute(...):
1. Bind the VAO (glBindVertexArray) and the VBO (glBindBuffer). Both of these are needed because we are linking the VAO with the VBO’s data.
2. Enable the vertex attribute using glEnableVertexAttribArray, passing in the “index” parameter.
3. Call glVertexAttribPointer. This is the function that defines how an attribute is stored in a VBO. Note that you must cast “pointer” to a (void*).
4. Unbind the VAO and VBO.

Now OpenGLShape can store data in a VBO and define how attributes are stored in it using a VAO.

We are finally ready to fill in the draw method!

Task 5: In OpenGLShape::draw(), bind the VAO and call glDrawArrays. This draws the vertices associated with the currently bound VAO. For the first and last parameters, use OpenGLShape’s member variables. For the second parameter assume that we want to draw starting at the beginning of the VBO’s data store. Don't forget to unbind the VAO!
Note that you do **not** need to bind the VBO in order to draw. The VAO knows which VBO(s) to use based on which VBO was bound when the VAO was last bound.

To summarize, we now have a class OpenGLShape that can:
1. Store data in a VBO.
2. Enable/define attributes using a VAO.
3. Draw the vertices.

The last thing we need to do is clean up the GPU memory. You already know that when you call “new” to allocate CPU memory, you need to call “delete” to free it or else your program will leak memory. Similarly, calling glGen* allocates memory on the GPU, which we need to free using glDelete*.

**Task 6:** In OpenGLShape’s destructor delete the VBO and VAO using glDeleteBuffers and glDeleteVertexArrays.

**Drawing a triangle**
Phew! We have now done all the dirty work necessary to draw shapes using OpenGL. Now let's get drawing!

Go to glwidget.cpp, a subclass of QGLWidget. This has two methods that we will be overriding:
- initializeGL() – Called once at the beginning of the program. Used for initializing shapes and specifying OpenGL settings.
- paintGL() – Called every time the screen should be updated. In an animated program like a video game, this might be called ~60 times per second.

Let's set up the vertex positions for a triangle. First we need to know what coordinate system we're using.

OpenGL’s screen coordinate system looks like this, where -1 ≤ x ≤ 1 and -1 ≤ y ≤ 1:
When giving OpenGL a vertex position, you will also need to specify a z-coordinate. Since we are only drawing 2D shapes in this lab, set z equal to 0 for all vertices. (In lab 2 we will start using other z values to create a 3D scene.)

For this lab you should choose three vertex positions for a triangle that looks like this:

![Triangle](image)

When storing the vertex positions in an array, make sure they’re defined in counterclockwise order! In GLWidget::initializeGL() we enabled GL_CULL_FACE, which makes sure only counter-clockwise faces are drawn. This is a common feature to enable, because without it the program would spend time drawing both sides of each face.

**Task 7:**
In GLWidget::initializeTriangle():
1. Make an array of floats containing the X, Y and Z coordinates of the triangle’s three vertices.
2. Pass the vertex data to m_triangle using setVertexData(...), where “size” is the size of your array in bytes, and “mode” is GL_TRIANGLES. The best way to get the size of an array is using the built-in C++ function “sizeof.” An array of 3 floats has a size of
   \[3 \times \text{sizeof(float)}\] bytes.
3. Call setAttribute. Some notes about the parameters:
   a. The “index” of the position attribute is 0 in this lab.
   b. “Size” refers to the number of number of coordinates of the attribute - in this case we have three: x, y, and z.
   c. “Type” is the data type: use GL_FLOAT for float data.
   d. Set “normalized” to GL_FALSE. Setting it to true automatically maps integers to the range [-1, 1].
   e. Set “stride” and “pointer” to 0. We will discuss these parameters more in Lab 3.

Now that the triangle has been initialized, we can **finally** draw it!

**Task 8:**
In GLWidget::paintGL(), draw m_triangle in the corresponding switch case. You should see your very first triangle!
If you don’t see your triangle:

- Make sure your vertices are defined in counterclockwise order.
- Make sure you define X, Y, and Z coordinates.
- Double check the arguments to your OpenGL function calls.
- If none of these work, ask a TA for help.

Oof, that sure was a lot of work to draw a lousy triangle. The good news is, now that we’ve made a handy dandy class to wrap all those OpenGL functions, we can make new shapes fairly easily.

**Drawing a Triangle Strip**

Let’s say we want to draw a rectangle made up of four triangles, like so:

![Diagram of a rectangle made up of four triangles](image)

One way to draw this would be to make an array with the vertices of triangle A, then the vertices of triangle B, etc. However, since triangles A and B share two of their vertices, this array would have a lot of repeated information.

Fortunately, OpenGL has some ways to avoid repeating vertices. When we’re drawing a strip of triangles like this, instead of using the GL_TRIANGLES mode we can use the GL_TRIANGLE_STRIP mode. With this mode, vertices are specified in the pattern shown below, and OpenGL assembles it as a “triangle strip” instead of independent triangles.

![Diagram of a rectangle made up of four triangles using GL_TRIANGLE_STRIP mode](image)

Note that the triangle strip will not show up if you go in the reverse order; that’s equivalent to drawing counter-clockwise using GL_TRIANGLES.
Task 9: Initialize and draw a rectangle using m_strip and the GL_TRIANGLE_STRIP drawing mode.

Drawing a Triangle Fan
The final drawing mode we will use is a triangle fan. This is good for drawing “circley” shapes, where many edges meet a single vertex. Using the GL_TRIANGLE_FAN mode, vertices are defined like so:

![Triangle Fan Diagram]

Note that to make a complete circle, the first outer vertex must be repeated at the end. Otherwise, it will look like a pizza that’s missing a slice.

Task 10: Initialize and draw an approximate circle using m_fan and the GL_TRIANGLE_FAN drawing mode.

Congratulations! You have finished your first OpenGL lab. Show your program to a TA to get checked off.