Chapter 1

1.1 Start

OpenGL = API = Application Programming Interface

OpenGL is more accurately a specification of some functions and how they behave not their implementation though. The implementation is done by the graphics card manufacturers.

OpenGL is a state machine. Then state of OpenGL is referred to as the context

We will use state-changing functions = functions that change the state and state-using functions = functions that do stuff by using the state

in OpenGL we use an abstraction called objects. Objects are just collection of data that represent some state.

the workflow for using objects is typically this

1. create the object
2. bind the object (activate it)
3. set its settings
4. unbind the object (deactivate it)

1.2 Creating a window

Link for setting up project LearnOpenGL - Creating a window

1.3 Hello Window

▼ Code

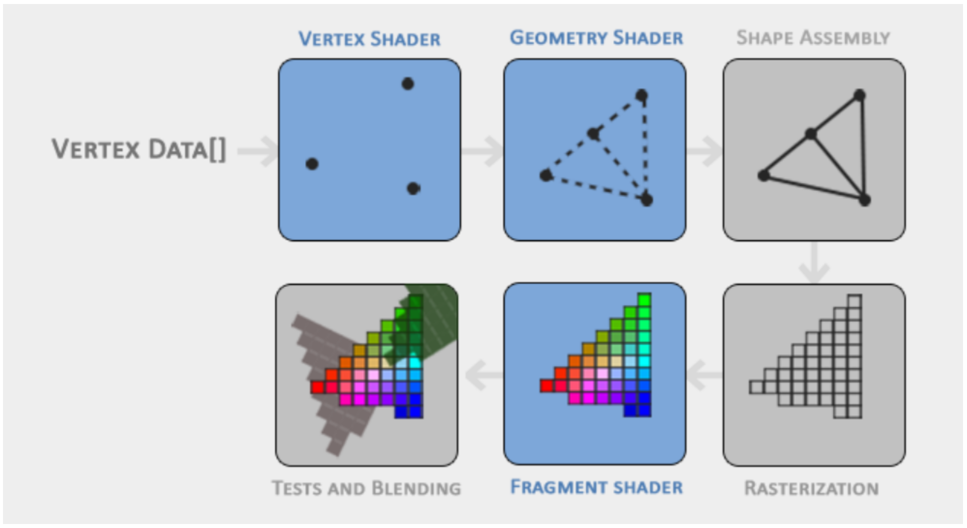
```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include <iostream>

void resizeEvent(GLFWwindow* window, int newWidth, int newHei
{
    glViewport(0, 0, newWidth, newHeight);
}

int main()
{
    // INIT GLFW
    glfwInit();

    // SET THE OPENGL VERSION AND PROFILE
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROF

    // CREATE THE WINDOW
    GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpe
    if (!window)
    {
        std::cout << "Failed to create GLFW window" << std::e
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);

    // GET THE OPENGL IMPLEMENTATIONS USING GLAD
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl
        return -1;
    }

    // TELL OPENGL THE SIZE OF OUR WINDOW
    glViewport(0,0,800,600);

    // ASSIGN A FUNCTION TO THE "RESIZE" EVENT
    glfwSetFramebufferSizeCallback(window, resizeEvent);

    // SET OUR CLEAR COLOR
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);

    // RENDERING LOOP
    while (!glfwWindowShouldClose(window))
    {
        // CLOSE THE APPLICATION IF WE PRESS ESCAPE
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS
            glfwSetWindowShouldClose(window, true);

        glClear(GL_COLOR_BUFFER_BIT);

        glfwSwapBuffers(window);
        glfwPollEvents(); // CHECKS IF ANY EVENTS ARE TRIGGER
    }

    glfwTerminate();
    return 0;
}
```

1.4 Hello Triangle

OpenGL takes care of transforming 3d to 2d colored pixels

programs that run on the GPU = shaders

Graphics pipeline



vertex = collection of data in a 3d coordinate/position. This data is represented using vertex attributes. A vertex can have data about its position, color etc.

Vertex shader

turns 3d coordinates to normalized device coordinates. 3d coordinates will range $(-\infty, +\infty)$ and normalized device coordinates will range $[-1, 1]$

Fragment shader

the fragment shader calculates the final color of our pixels

We will store our 3D data in our GPU so that we can access it quickly. We then tell OpenGL how it should manage that block of memory we created in the GPU. We achieve this using vertex buffer objects (VBO). This vertex buffer object is an type of object we discussed above

```
unsigned VBO;
glGenBuffers(1, &VBO);
```

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle, GL_ST
```

Vertex Attributes

a vertex attribute is subdata of each vertex. For example a vertex attribute is a position, a color etc.. Each vertex will have many vertex attributes.

```
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(fl
```

this associates the data we have in memory, using the currently bound VBO to the GL_ARRAY_BUFFER target, with the input we have in the vertex shader

Vertex array object

if we wanted to draw 100 objects we would have to tell OpenGL 100 different times how we want our vertex data to be interpreted. With vertex array objects we can tell it once how our data will be stored in the GPU and what it will represent

Element buffer objects

element buffer objects are just buffers that are used to store, in what order we want to draw our vertices

1.5 Shaders

shaders = programs that run on GPU

shaders in OpenGL are written in GLSL

each shader has some inputs, outputs and uniforms

inputs = vertex attributes

the vertex shader will get its inputs from vertex data

Uniforms

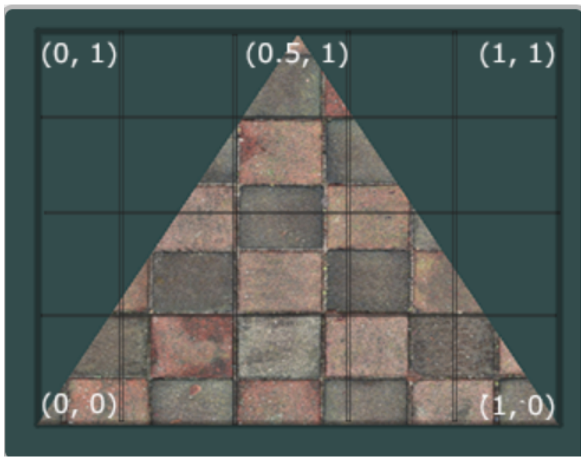
allows us to send data from CPU to GPU. They are global in our shader programs.

1.6 Textures

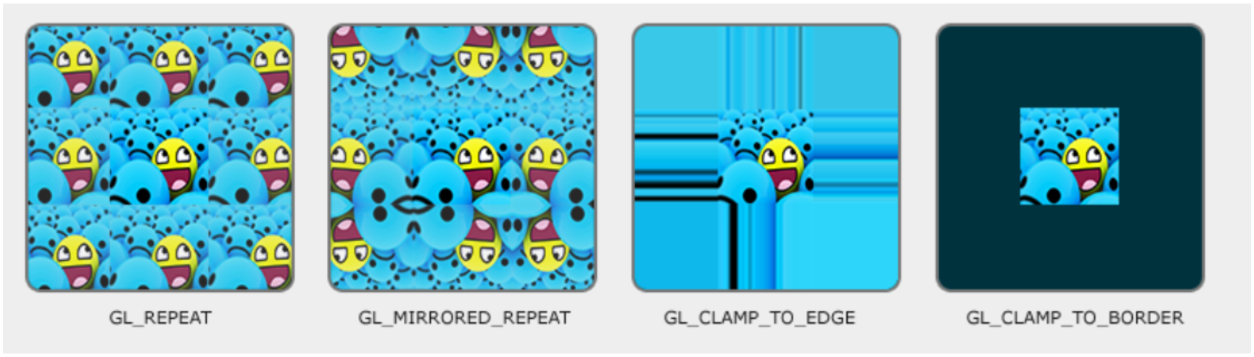
texture = 2d image

each vertex will have a vertex attribute specifying its texture coordinates. Texture coordinates are basically the “position” in our image. OpenGL will interpolate the texture between vertices.

taking the texture color using textures coordinates = sampling



if we specify coordinates outside of $[-1, 1]$? The default behavior is to repeat the texture



defaults is `GL_REPEAT`

these options can be set per axis

```
unsigned texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_R
glGenerateMipmap(GL_TEXTURE_2D);
```

1. generate the texture
2. bind it to the `GL_TEXTURE_2D` target
3. use `glTexImage2D` to create the texture. When this function is called the texture object will have the texture bound to it.
4. generate the mipmap

1.7 Transformations

Just use the glm library and some linear algebra stuff

1.8 Coordinate systems