UNIVERSITÉ Concordia UNIVERSITY

COMP 371
Computer Graphics

Lab 01 – Hello OpenGL

Prepared by Nicolas Bergeron

This Week

Tutorial: Complete the code

OpenGL Program Structure

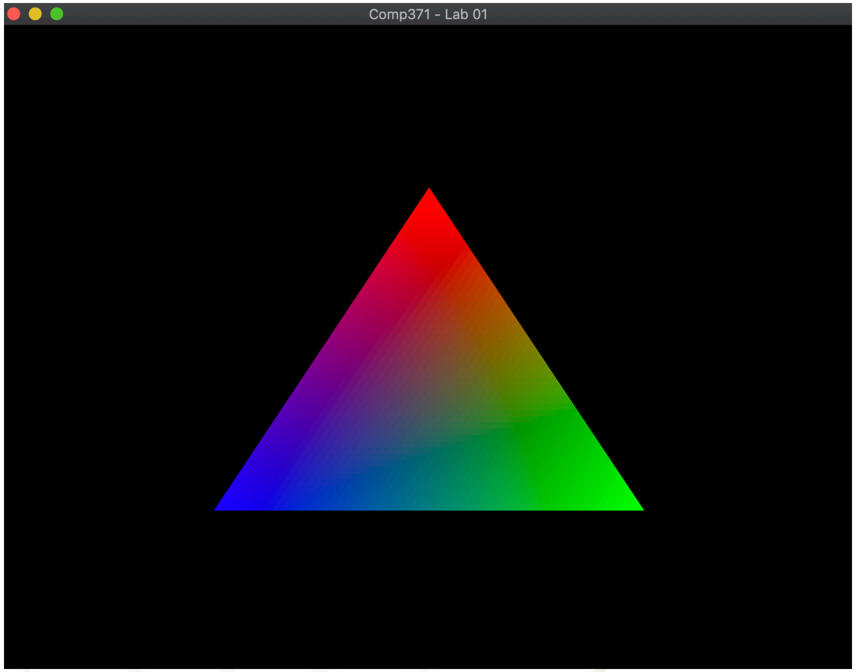
Uploading Geometry to Graphics Hardware

Drawing Geometry with Shaders

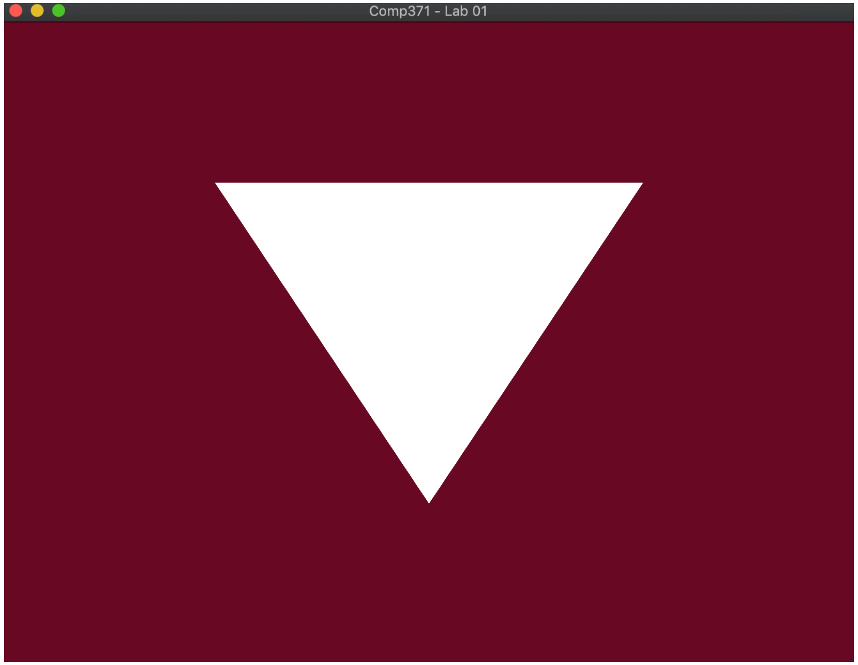
Exercises

Expected results

Tutorial results



Exercises results



Getting Started

- Download Lab01.zip from Moodle
- Open the Project (Visual Studio or Xcode)
- These slides are on Lab01.pdf in the .zip file

- Pair programming is recommended (work with a classmate), you need to be precise while programming graphics, typos can be hard to debug!

❖ Disclaimer: The tutorial may feel a bit like magic! With time, you will get familiar with these terms

Provided Lab Framework

- Visual Studio (Windows) and Xcode (Mac) projects are provided, it should build and run with all dependencies!
  - **GLEW**: we cannot link OpenGL functions directly. GLEW sets up the OpenGL function pointers matching the OpenGL version requested.
  - **GLFW**: Cross Platform API for creating window, OpenGL context, binding inputs and managing OS events
  - **GLM**: Optimized library with syntax similar to the OpenGL Shading Language (GLSL)
- For upcoming labs, you will reuse the same framework by adding new source files

Provided Lab01 Code

- Instantiates a window using GLFW, and specify options for the OpenGL context
- Initializes OpenGL with GLEW
- Binds the ESC key to exit the application
- Implements the Rendering Main Loop
  - Sets the frame buffer (pixels) to black
  - Poll events from OS
  - Detects if ESC key is pressed to exit

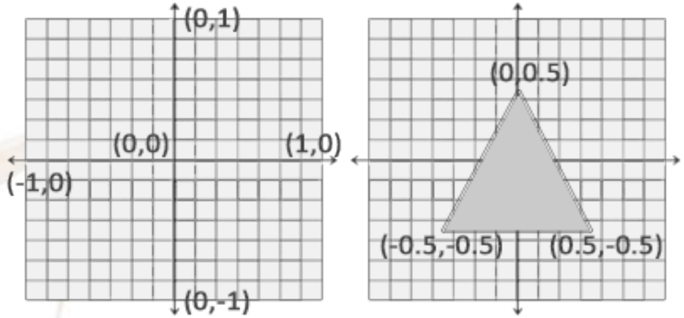
TUTORIAL
DRAW RAINBOW TRIANGLE

Provided Code Overview

- GLFW and GLEW initialization: Create a window and sets the version of OpenGL
- glClearColor – State containing the color to clear the screen
- (Missing vertex buffer object, containing geometry)
- (Missing shader strings)
- (Missing shader compilation)
- MainLoop for Drawing – Draws a single frame (image)
  - Clears the screen with clear color
  - (missing code to draw geometry)
  - glfwSwapBuffers(window) – Frame is ready to draw (finished)
  - Detects if ESC is pressed to close program

Defining Geometry

- By default, OpenGL draws the geometry inside Normalized Device Coordinates (NDC) [see below]
- To draw a triangle visible on the screen, the position of its vertices must be in the range [-1, 1]
- Points on a polygon are called a vertices. A vertex contains positions and other data (color, normal, etc)
- In Lab02, we will see how to change these boundaries (View, Projection, and Viewport transforms)



Uploading Geometry (VBO) to GPU (1/2)

(Feel free to copy/paste code)

```
int createVertexBufferObject()
{
    // A vertex is a point on a polygon, it contains positions and other data (eg: colors)
    glm::vec3 vertexArray[] = {
        glm::vec3( 0.0f,  0.5f, 0.0f),  // top center position
        glm::vec3( 1.0f,  0.0f, 0.0f),  // top center color (red)
        glm::vec3( 0.5f, -0.5f, 0.0f),  // bottom right
        glm::vec3( 0.0f,  1.0f, 0.0f),  // bottom right color (green)
        glm::vec3(-0.5f, -0.5f, 0.0f),  // bottom left
        glm::vec3( 0.0f,  0.0f, 1.0f),  // bottom left color (blue)
    };

    // Create a vertex array
    GLuint vertexArrayObject;
    glGenVertexArrays(1, &vertexArrayObject);
    glBindVertexArray(vertexArrayObject);

    // Upload Vertex Buffer to the GPU, keep a reference to it (vertexBufferObject)
    GLuint vertexBufferObject;
    glGenBuffers(1, &vertexBufferObject);
    glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObject);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexArray), vertexArray, GL_STATIC_DRAW);

    // cont. next slide!
```

Uploading Geometry (VBO) to GPU (2/2)

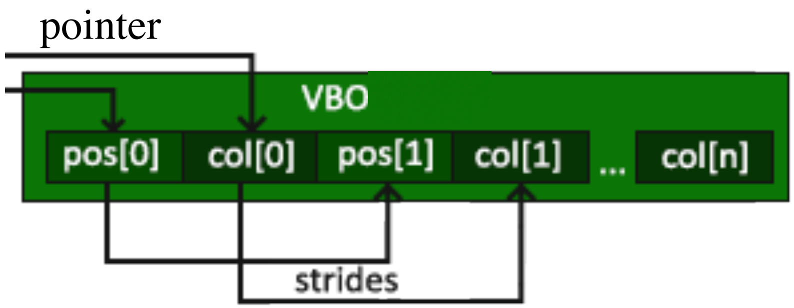
```
    // Specify where the data is in the VAO – this allows OpenGL to bind data to vertex shader attributes
    glVertexAttribPointer(0,                   // attribute 0 matches aPos in Vertex Shader
                          3,                   // size
                          GL_FLOAT,            // type
                          GL_FALSE,            // normalized?
                          2*sizeof(glm::vec3), // stride - each vertex contain 2 vec3 (position, color)
                          (void*)0             // array buffer offset
                          );
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1,                            // attribute 1 matches aColor in Vertex Shader
                          3,
                          GL_FLOAT,
                          GL_FALSE,
                          2*sizeof(glm::vec3),
                          (void*)sizeof(glm::vec3)      // color is offseted a vec3 (comes after position)
                          );
    glEnableVertexAttribArray(1);

    return vertexBufferObject;
}
```

Vertex Buffer Object (VBO)

- The Vertex Buffer object is using data from a Vertex Array Object (VAO)
- The layout of the VAO is arbitrary, the memory layout of the data is set by **glVertexAttribPointer**. This allows the programmable graphics pipeline to assign data to vertex shader attributes.
- Think of a VBO as something containing a large 3D model. If we want to draw the same model multiple times (eg: tree in a forest), we can re-use the same VBO.

Using glVertexAttribPointer

```
void glVertexAttribPointer( GLuint index,
                            GLint size,
                            GLenum type,
                            GLboolean normalized,
                            GLsizei stride,
                            const GLvoid * pointer);
```

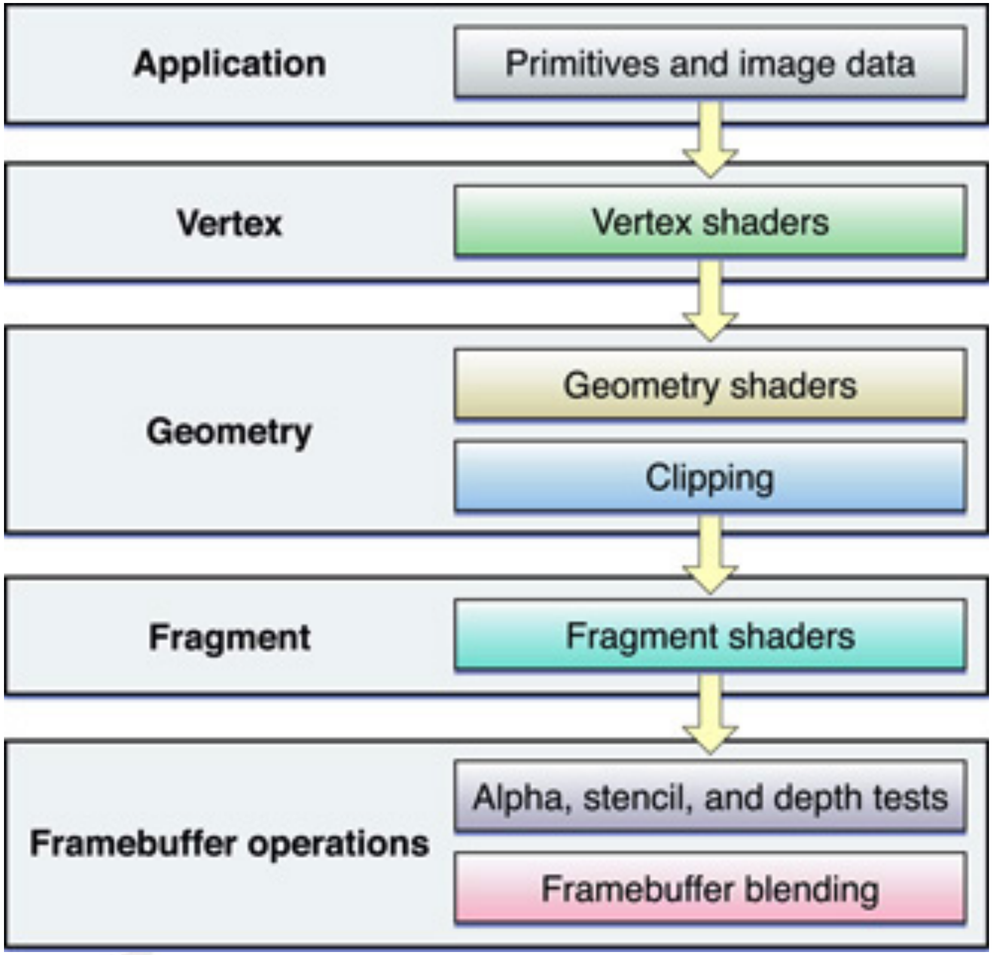


- Describes Vertex Data Format for Drawing
- Parameters are
  - Index: layout index in Vertex Shader
  - Size, Type: Example - vec3 is 3 floats
  - Normalized: maps integers to [0,1] or [-1,1]
  - Stride: How many bytes between values
  - Pointer: Offset for first data (# bytes from start address of the vertex buffer object)

The 3D Graphics Pipeline

(Transformation)

(Rasterization)



(CPU)

(GPU)

Vertex and Fragment Shaders
(Standalone Programs compiled and linked at runtime)

Vertex Shader

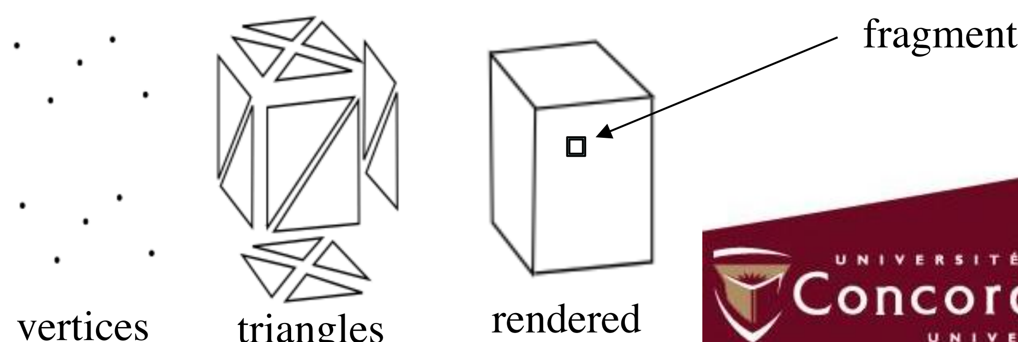
- The vertex shader is a program that computes output from inputs provided (main function)
- Input types
  - Vertex Attributes (Position, Color, Normals, …)
  - Uniform values (constant for every vertex)
- Outputs are sent to next stage (fragment shader)

triangulation

interpolation

Fragment Shader

- A fragment is an element of a triangle (can be a pixel)
- The fragment shader receives interpolated inputs from vertex shader and uniform values
- It computes color of the output fragment



Adding Shaders to Tutorial
(Feel free to copy/paste code)

Vertex Shader

const char* getVertexShaderSource()
{

```
return
    "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;"
    "layout (location = 1) in vec3 aColor;"
    "out vec3 vertexColor;"
    "void main()"
    "{"
    "   vertexColor = aColor;"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);"
    "}";
```

}

- aPos and aColor are vertex attributes
- vertexColor is the output
- gl_Position is the output position of the vertex. In later labs, it will be computed based on other transformations
- The string will be later **compiled** and **linked** in program.

Fragment Shader

const char* getFragmentShaderSource()
{

```
return
"#version 330 core\n"
    "in vec3 vertexColor;"
    "out vec4 FragColor;"
    "void main()"
    "{"
    "   FragColor = vec4(vertexColor.r, vertexColor.g,"
    "                    vertexColor.b, 1.0f);"
    "}";
```

}

- VertexColor is the interpolated color initially calculated on the vertex shader
- FragColor is the output of the fragment shader, it will determine the color of the pixel
- Later, we will compute lighting on the fragment based on light properties

Compiling and Linking Shaders (1/2) (add to compileAndLinkShaders())

```
// vertex shader
int vertexShader = glCreateShader(GL_VERTEX_SHADER);
const char* vertexShaderSource = getVertexShaderSource();
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

// check for shader compile errors
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success) {
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cerr << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}

// fragment shader
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
const char* fragmentShaderSource = getFragmentShaderSource();
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);

// check for shader compile errors
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success){
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cerr << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

Compiling and Linking Shaders (1/2)

```
// link shaders
int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);

// check for linking errors
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cerr << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}

glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);

return shaderProgram;
```

- During compilations, errors (if any) are displayed
- Shaders are uploaded to GPU, the shaderProgram integer will be used to reference these shaders when drawing geometry

Drawing Geometry (type code in image below)

- Now that we have shaders and geometry on the GPU, we can render it on the screen!
- The first step is to set OpenGL's rendering state (in this case, shader and VBO to use).
- The second step is to draw the VBO, in mainloop:

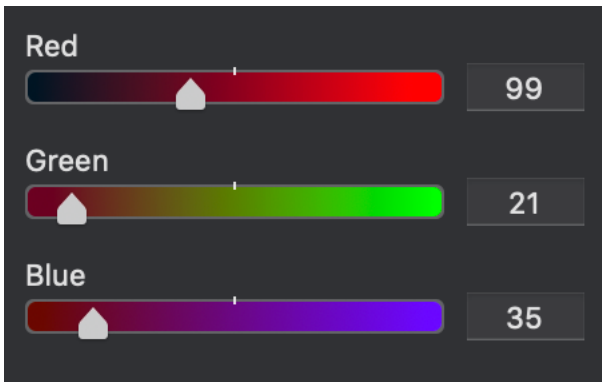
```
// TODO - draw rainbow triangle
glUseProgram(shaderProgram);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glDrawArrays(GL_TRIANGLES, 0, 3); // 3 vertices, starting at index 0
```

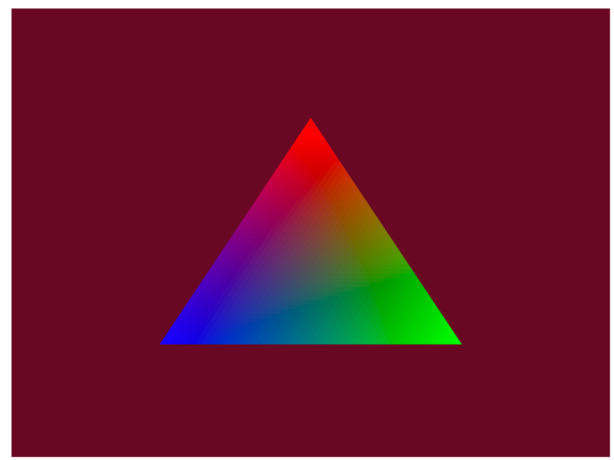
- For drawing multiple types of geometry, we may use different shaders and vbo.

EXERCISES

Exercise 1

- Change the background to the Concordia Burgundy

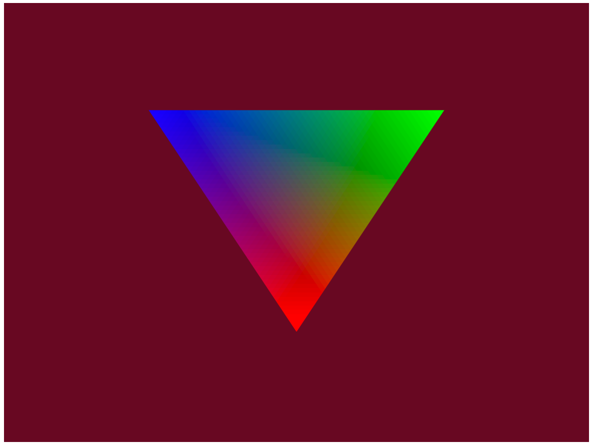




- Clue: RGB values are in [0, 1], not [0,255]

Exercise 2

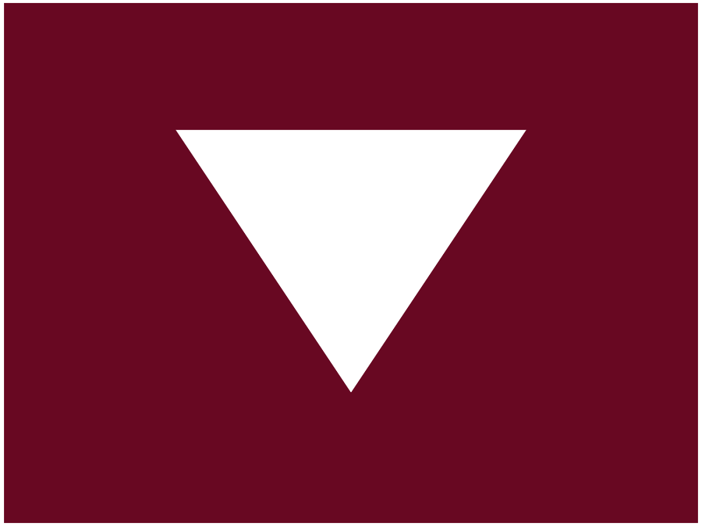
- **Without** changing the geometry data (vertex array), make the program render the triangle upside down



- *Clue: You can recalculate positions in a Vertex Shader*

Exercise 3

- **Without** changing the geometry data (vertex array), make the program render the triangle white



- *Clue: You can override colors in a Fragment Shader*