Computer Graphics
– Hello Triangle

J.-Prof. Dr. habil. Kai Lawonn

OpenGL Basics

Basics

- In OpenGL everything is in 3D space
- Screen and window are a 2D array of pixels

  → large part of OpenGL's work is about transforming 3D coordinates to 2D pixels
- Transforming is managed by the graphics pipeline of OpenGL
- Pipeline can be divided into two large parts:
  - 1. Transformation of 3D coordinates into 2D coordinates
  - 2. Transformation of 2D coordinates into colored pixels

Basics

There is a difference between a 2D coordinate and a pixel:

- **2D coordinate is a very precise representation of a point**
- **2D pixel is an approximation of that point limited by the resolution**

Basics

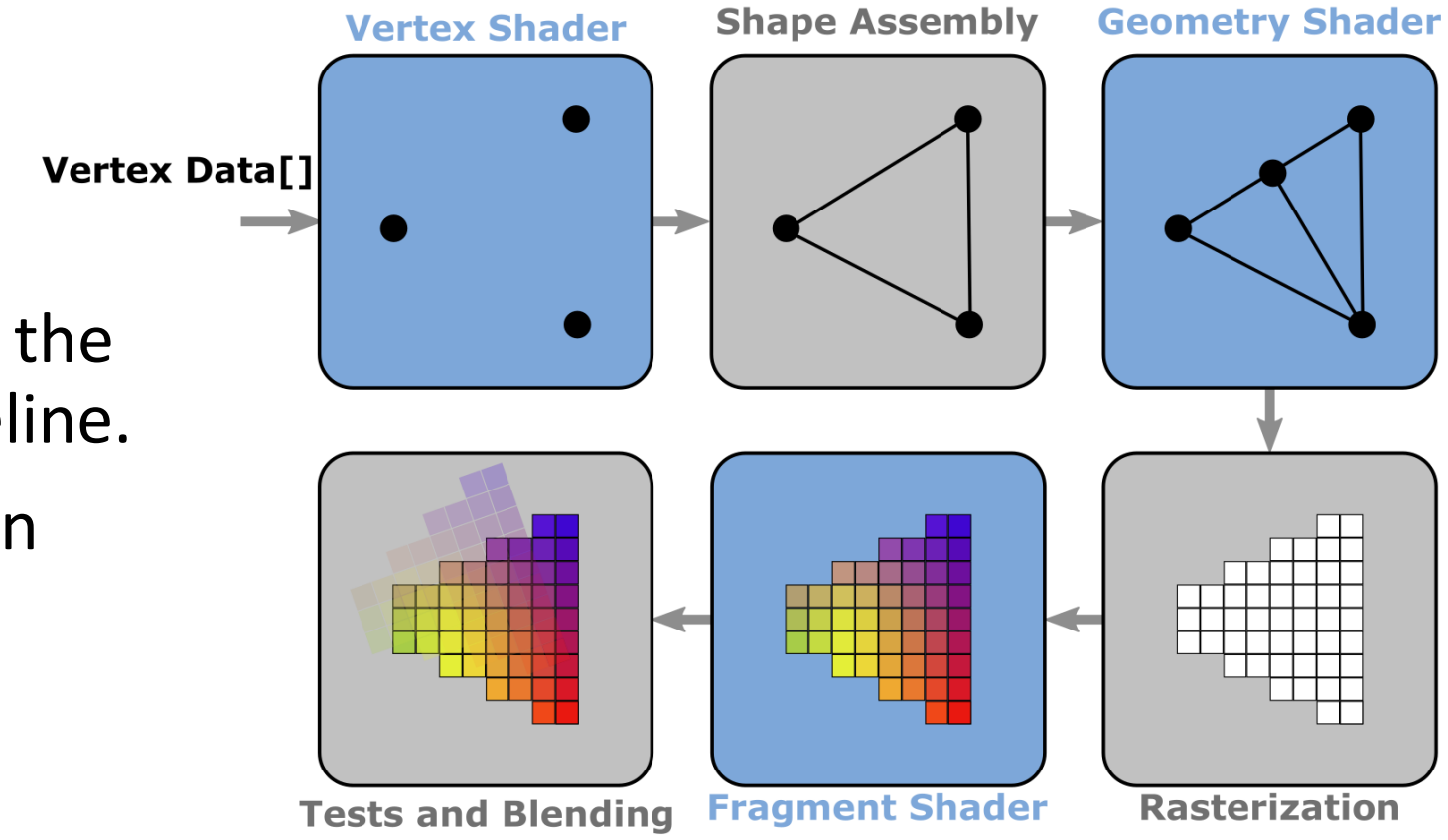
- Graphics pipeline takes as input a set of 3D coordinates and transforms these to colored 2D pixels on your screen
- The graphics pipeline can be divided into consecutive several steps
- All steps can easily be executed in parallel
- Most graphics cards of today have thousands of small processing cores that run small programs on the GPU for each step of the pipeline: shaders

Basics

- Some shaders are configurable: possible to replace shaders
- Because they run on the GPU, they can also save us valuable CPU time
- Shaders are written in the OpenGL Shading Language (GLSL)

Basics

- Abstract representation of the stages of the graphics pipeline.
- Blue: we can inject our own shaders



Vertex Data

Vertex Data[]

- As input pass a list of three 3D coordinates that should form a triangle (array Vertex Data)
- Vertex is basically a collection of data per 3D coordinate
- Vertex attributes can contain any data, assume each vertex consists a 3D position and a color value

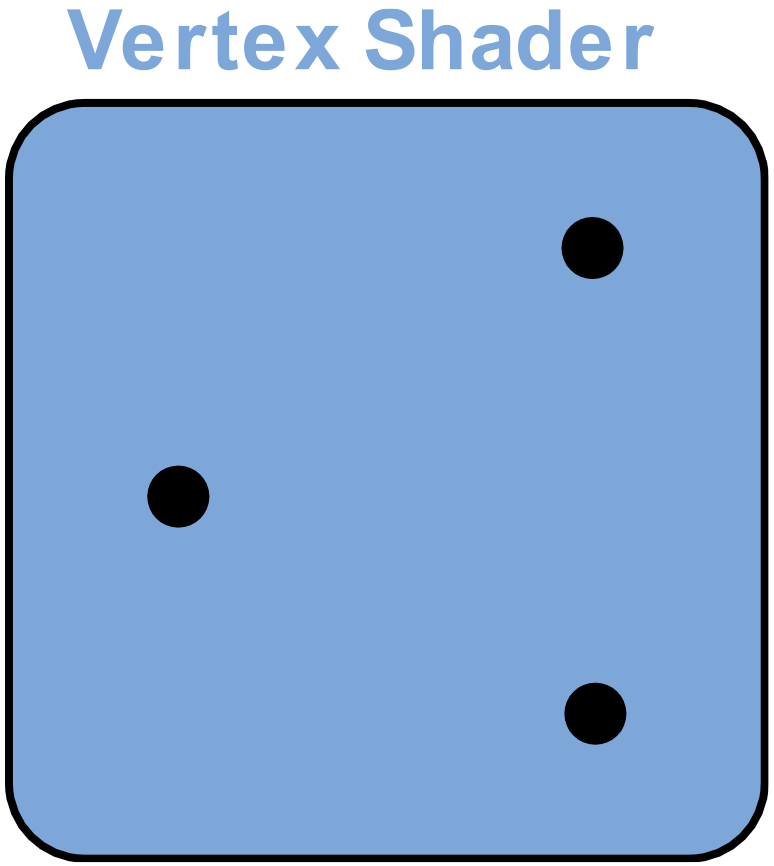
Primitives

OpenGL requires the render types you want to form with the data:

- **points, triangles, lines, etc.**
- **GL_POINTS, GL_TRIANGLES and GL_LINE_STRIP.**

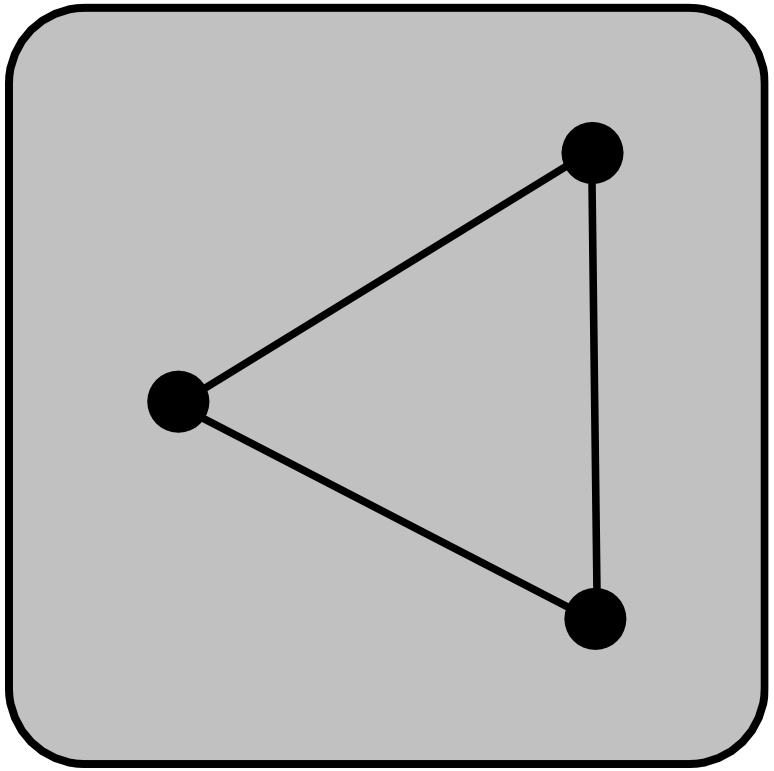
Vertex Data

- First: vertex shader
- Input a single vertex
- Main purpose of the vertex shader is to transform 3D coordinates into different 3D coordinates and some basic processing on the vertex attributes



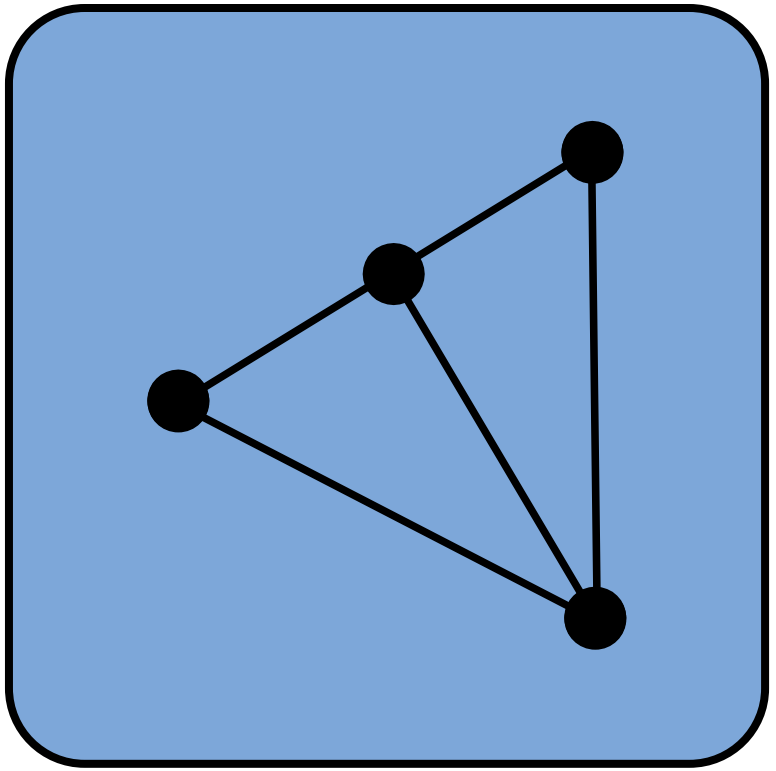
Shape Assembly

- Assembly stage takes as input all the vertices from the vertex shader that form a primitive and assembles all the point(s) in the primitive shape given
- In this case a triangle



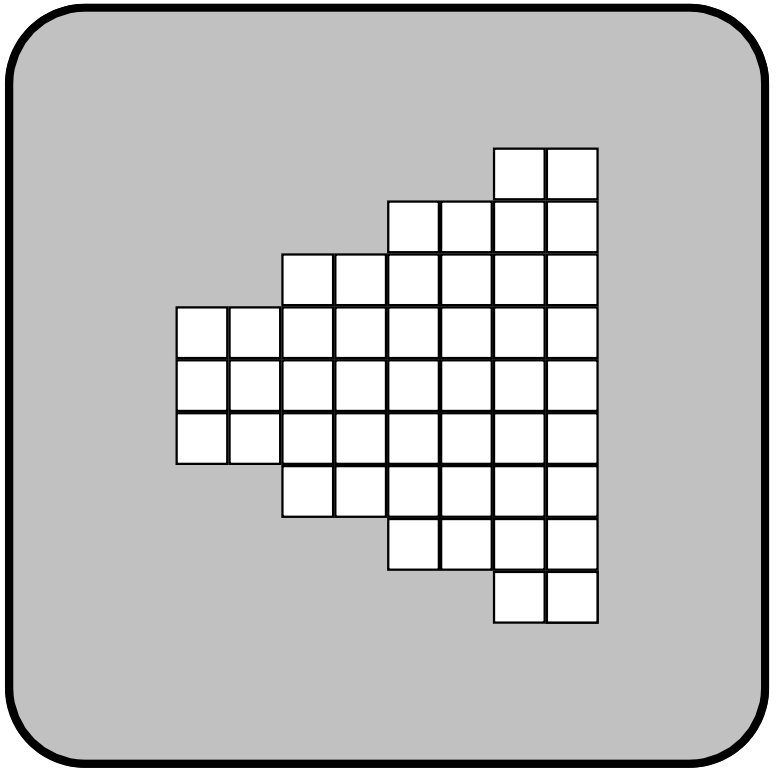
Geometry Shader

- Geometry shader takes as input a collection of vertices that form a primitive
- Has the ability to generate other shapes by emitting new vertices to form new (or other) primitive(s)
- In this example case, it generates a second triangle out of the given shape



Rasterization

- Rasterization stage maps the resulting primitive(s) to the corresponding pixels on the final screen (resulting in fragments)
- Here, clipping is performed and discards all fragments that are outside the view (performance)

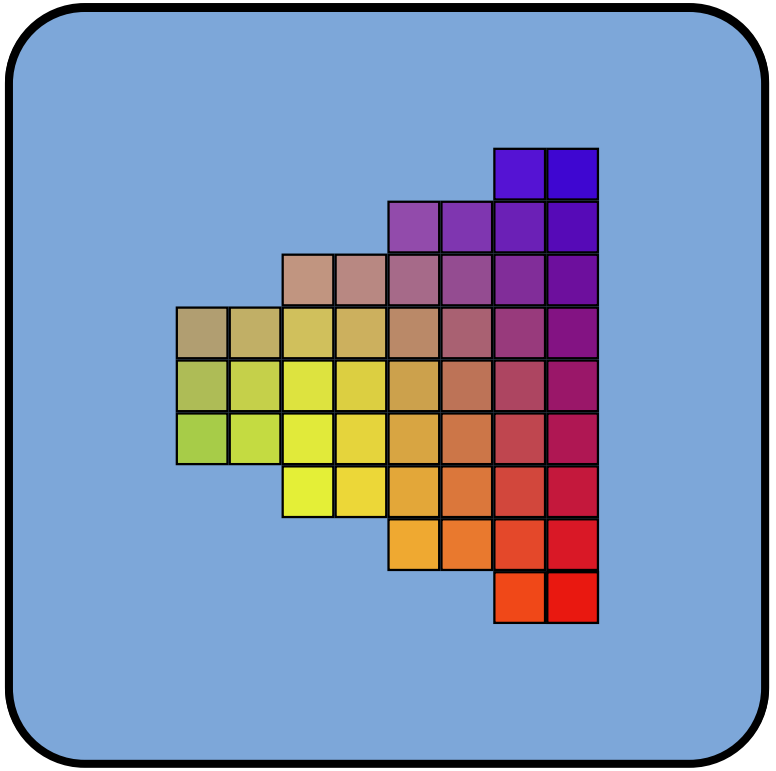


Primitives

A fragment in OpenGL is all the data required for OpenGL to render a single pixel.

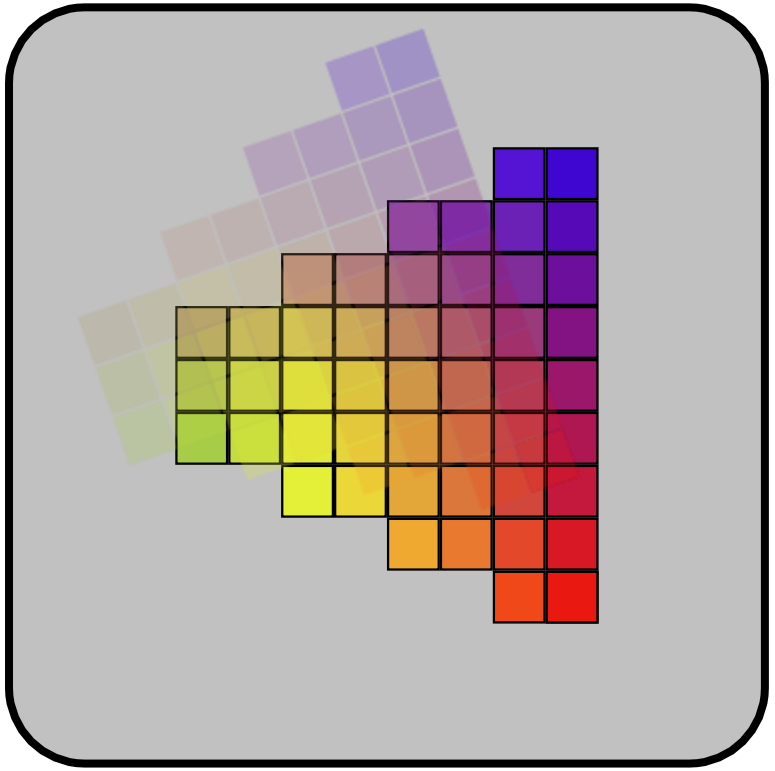
Fragment Shader

- Fragment shader calculates the final color
- Usually the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color (like lights, shadows, color of the light and so on)



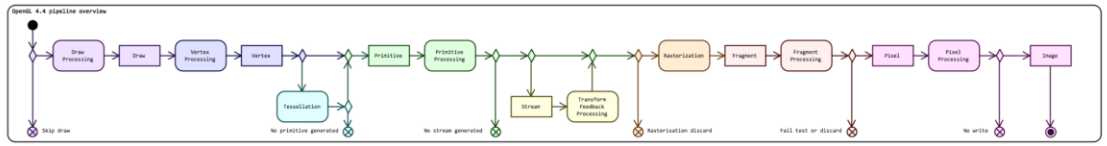
Tests

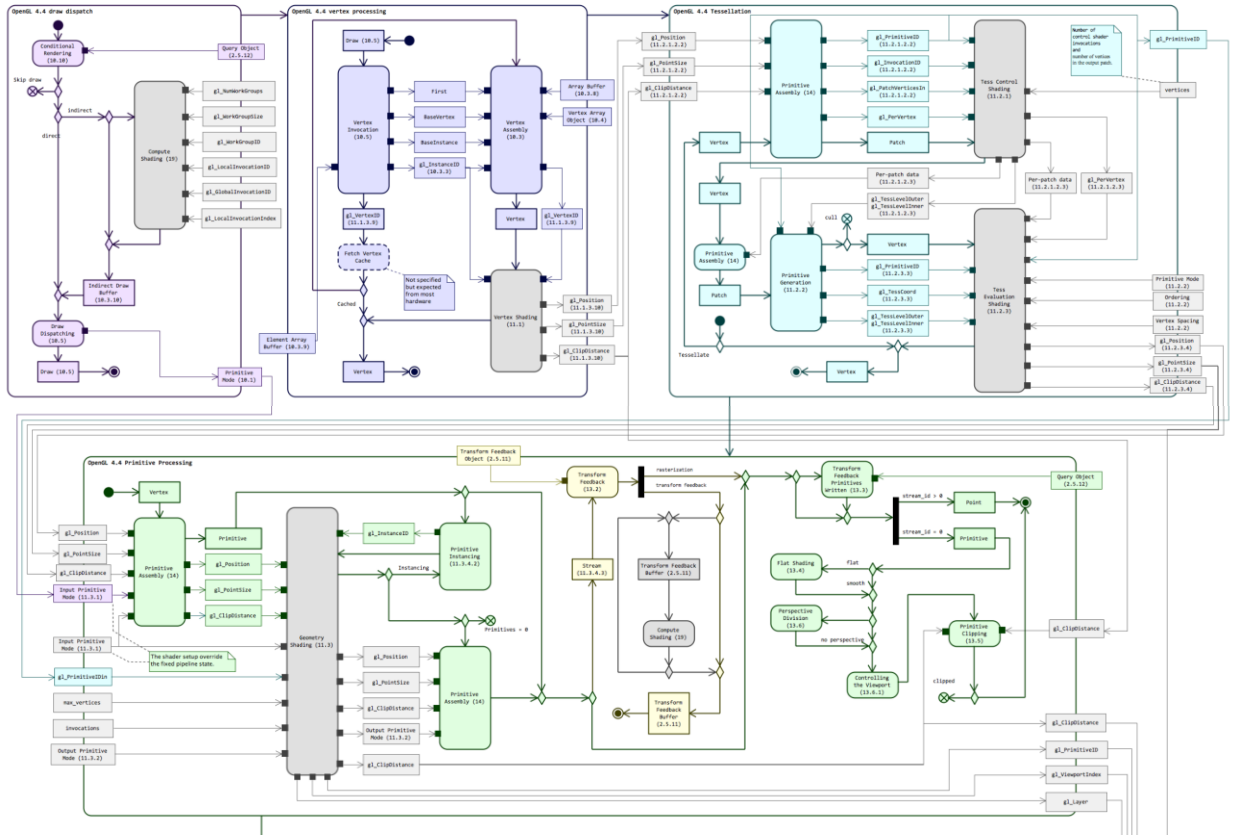
- After all the corresponding color values have been determined, the final object will then pass through one more stage that we call the alpha test and blending stage
- This stage checks the corresponding depth(check if resulting fragment is in front or behind other objects), also checks for alpha values

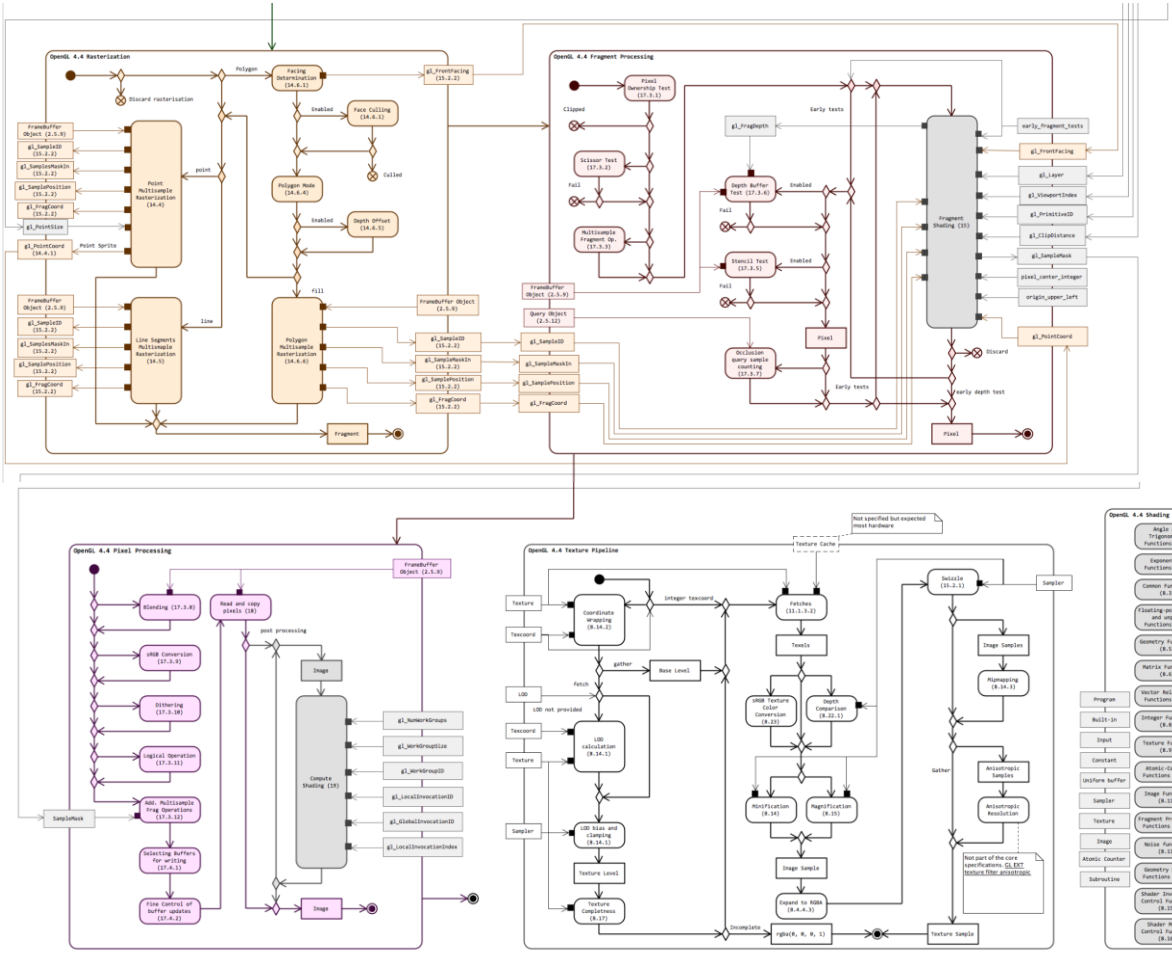


Tests and Blending

Pipeline







https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGL44PipelineMap.pdf

Input

Vertex Input

- OpenGL needs some input vertex data (coordinates are in 3D)
- OpenGL doesn’t simply transform 3D coordinates to 2D pixels on screen
- **OpenGL only processes 3D coordinates when they’re in a specific range: [-1.0, 1.0] for x, y and z**
- All coordinates within this (*normalized device coordinates*) range will be visible (coordinates outside this region won’t)

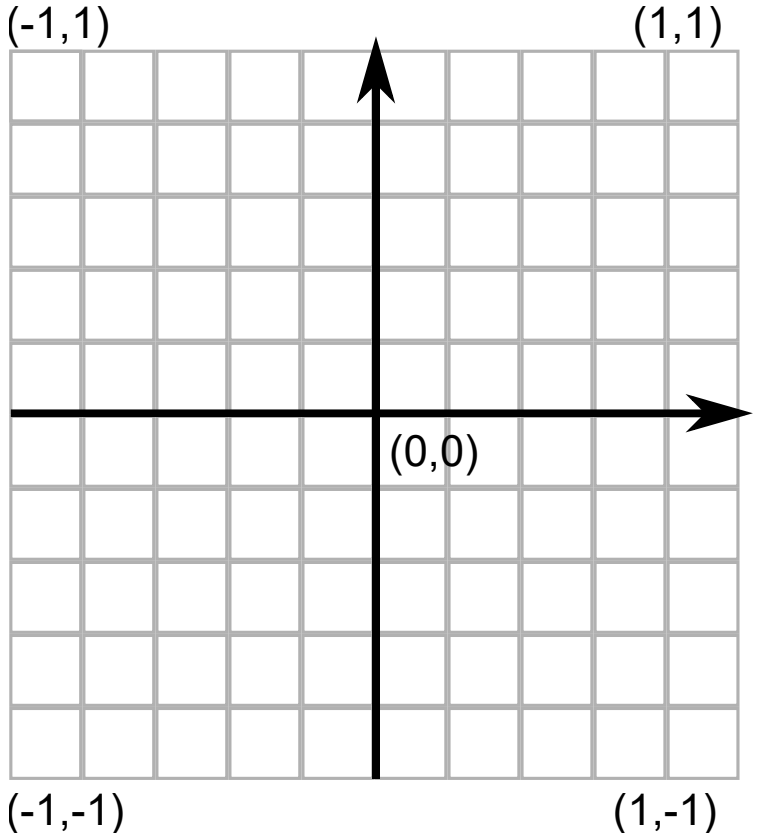
Vertex Input

- Want to render a single triangle: need to specify three vertices 3D positions
- Define them in normalized device coordinates (visible region) in a float array:
- OpenGL works in 3D space: render a 2D triangle with each z=0.0
- → Depth of the triangle remains the same making it look like it's 2D

```
float vertices[] = {
        -0.5f, -0.5f, 0.0f, // left
         0.5f, -0.5f, 0.0f, // right
         0.0f,  0.5f, 0.0f  // top
};
```

Normalized Device Coordinates (NDC)

- **OpenGL only processes 3D coordinates when they're in a specific range: [-1.0, 1.0] for x, y and z**
- **Positive y-axis points in the up-direction and the (0,0) coordinates are at the center of the graph**
- **NDC coordinates will then transformed to screen-space coordinates via the viewport transform (using the data from glViewport)**
- **Resulting screen-space coordinates transformed to fragments as inputs to your fragment shader.**



Start

- Vertex data need to be send it as input to the first process of the graphics pipeline: the vertex shader
- Therefore, memory needs to be created on the GPU to store the vertex data
- Configure how OpenGL should interpret the memory
- Specify how to send the data to the graphics card
- Vertex shader then processes the vertices

Vertex Buffer Objects (VBOs)

- Manage this memory via the vertex buffer objects (VBOs): store vertices in the GPU's memory
- Advantage of VBOs: can send large batches of data all at once
- Sending data to the graphics card from the CPU is relatively slow → send as much data as possible at once
- Once the data is in the graphics card's memory vertex shader has almost instant access to the vertices making it extremely fast

glGenBuffers

- A vertex buffer object is our first OpenGL object
- Any object in OpenGL has a unique ID
- Can generate one with a buffer ID using the glGenBuffers function:

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

glBindBuffer

- OpenGL has many buffer objects, for a vertex buffer object it is GL_ARRAY_BUFFER
- OpenGL allows binding several buffers at once (if they have a different buffer type)
- Bind the created buffer to the GL_ARRAY_BUFFER target with the glBindBuffer function:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

glBufferData

- Then make a call to glBufferData function that copies the previously defined vertex data into the buffer's memory:
- glBufferData is a function targeted to copy user-defined data into the currently bound buffer
- First argument is the type of the buffer (copy data into)
- Second argument specifies the size of the data (in bytes) (pass to the buffer)
- Third parameter is the actual data

```
glBufferData(GL_ARRAY_BUFFER,
sizeof(vertices), vertices,
GL_STATIC_DRAW);
```

glBufferData

- Fourth parameter specifies how the graphics card manages the given data (3 possible forms)
- GL_STATIC_DRAW: the data will most likely not change at all or very rarely
- GL_DYNAMIC_DRAW: the data is likely to change a lot
- GL_STREAM_DRAW: the data will change every time it is drawn
- The position data of the triangle does not change and stays the same for every render call so its best be GL_STATIC_DRAW

```
glBufferData(GL_ARRAY_BUFFER,
sizeof(vertices), vertices,
GL_STATIC_DRAW);
```

Shader

- The data are now stored on the graphics card
- Next step is to create a vertex and fragment shader that actually processes this data and draws it

Shader

Vertex Shader

- Modern OpenGL requires to set up at least a vertex and fragment shader to do some rendering
- First, write the vertex shader in the shader language GLSL (OpenGL Shading Language) and then compile this shader
- Example of a very basic vertex shader in GLSL:

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
{
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
{
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

- First, version declaration (330 = OpenGL 3.3, 450 = 4.5, etc.)
- Next, declare all the input vertex attributes in the vertex shader with the *in* keyword
- GLSL has a vector datatype that contains 1 to 4 floats, since each vertex has a 3D coordinate → vec3 with the name aPos
- Set the location of the input variable via layout (location = 0)

Vector

The mathematical concept of a vector is used quite often.

A vector in GLSL has a maximum size of 4 and each of its values can be retrieved via vec.x, vec.y, vec.z and vec.w.

Note that the vec.w component is not used as a position in space (3D, not 4D) but is used for something called perspective division

Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
{
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

- Output of the vertex shader need to assigned with gl_Position, which is a vec4
- Input is a vector of size 3 → cast this to a vector of size 4
- So, insert the vec3 values inside the constructor of vec4 and set w=1

Compiling a Shader

- Shader is defined as a C string (const char *)

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
    "}\0";
```

Compiling a Shader

- First, create a shader object, again referenced by an ID
- Store the vertex shader as a GLuint and create the shader with glCreateShader
- Type of shader is an argument to glCreateShader: GL_VERTEX_SHADER.
- Next, attach the shader source code to the shader object and compile the shader

```
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

Compiling a Shader

```
unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

- glShaderSource function:
  - 1. argument: shader object to compile
  - 2. argument: how many strings are passed as source code (only one)
  - 3. argument: actual source code
  - 4. argument: an array of string lengths (just for the case 2. argument >1)

Compiled?

- Compilation successful? If not, what errors?
- Define an integer to indicate success and a storage for the error messages (if any)
- Check if compilation was successful with glGetShaderiv if not: retrieve the error message with glGetShaderInfoLog and print it:

```
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog
    << std::endl;
}
```

Fragment Shader

- The fragment shader is the second and final shader
- Fragment shader is all about calculating the color output
- To keep things simple the fragment shader will always output an orange-ish color:

```
#version 330 core
out vec4 FragColor;
void main()
{
   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Colors

Colors in computer graphics are represented as an array of 4 values: the red, green, blue and alpha (opacity) component (RGBA).

Values are between 0.0 and 1.0.

Fragment Shader

```
#version 330 core
out vec4 FragColor;
void main()
{
   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

- The fragment shader only requires one output variable: a vector of size 4 defining the final color
- Declare output values with the *out* keyword, here named FragColor
- Next, assign a vec4 to the color output as an orange color

Compiling again

- The process for compiling a fragment shader is similar to the vertex shader, but this time GL_FRAGMENT_SHADER is used

```
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
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::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog <<
    std::endl;
}
```

Shader Program

- A shader program object is the final linked version of multiple shader
- To use the recently compiled shaders they need to be linked to a shader program object
- glCreateProgram function creates a program and returns the ID reference to the newly created program object
- Then the compiled shaders are attached to the program object and then linked with glLinkProgram:

```
unsigned int shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Linking failed?

- Just like shader compilation we can also check if linking a shader program failed and retrieve the corresponding log
- However, instead of using glGetShaderiv and glGetShaderInfoLog we now use:

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog <<
        std::endl;
    }
```

Run the Shader

- The program object can be activated by calling glUseProgram:

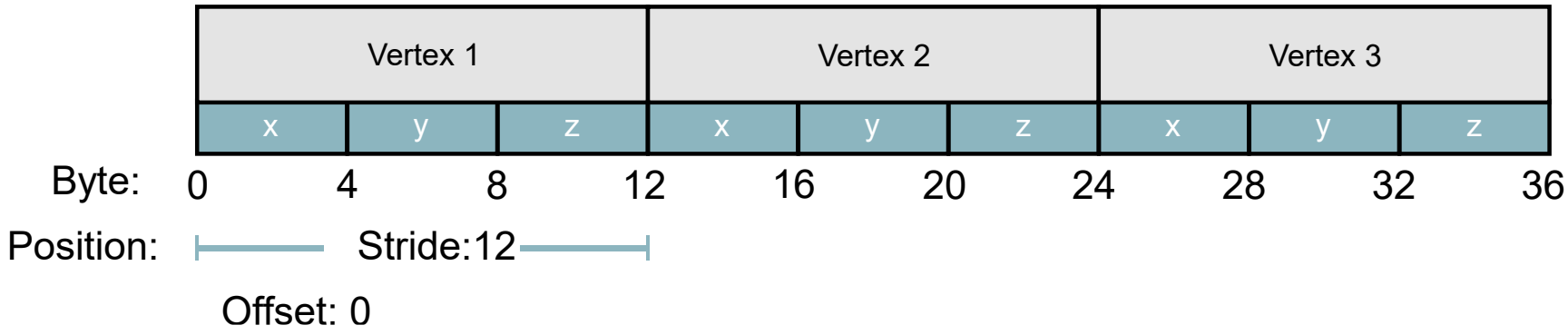
```
glUseProgram(shaderProgram);
```

- Delete the shader objects once they are linked into the program object

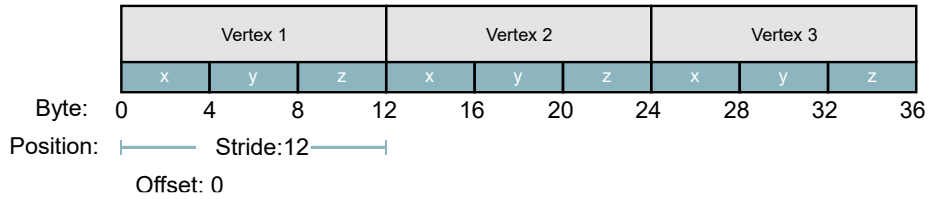
```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Linking Vertex Attributes

- OpenGL does not yet know how it should interpret the vertex data in memory and how it should connect the vertex data to the vertex shader's attributes
- Specify how OpenGL should interpret the vertex data
- Vertex buffer data is formatted as follows:



Linking Vertex Attributes



- With this, provide OpenGL the vertex data using glVertexAttribPointer:
  - 1. parameter: vertex attribute to configure (specified the vertex attribute's location → vertex shader with layout (location = 0)).
  - 2. parameter: size of the vertex attribute (vec3 so it is 3)
  - 3. parameter: type of the data (GL_FLOAT)
  - 4. parameter: normalized data? If GL_TRUE all the data not between 0 (or -1 for signed data) and 1 will be clamped
  - 5. parameter: stride (space between consecutive vertex attribute sets)
  - 6. parameter: is of type void* (offset of the beginning)
- Enable the vertex attribute with glEnableVertexAttribArray(vertex attribute location as argument)

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Colors

Each vertex attribute takes its data from memory managed by a VBO

Multiple VBOs are possible

The previously defined VBO was bound before calling glVertexAttribPointer vertex attribute 0 is now associated with its vertex data.

Draw

- From that point on everything is set up:
  - Initialized the vertex data in a buffer using a VBO
  - Set up a vertex and fragment shader and told OpenGL how to link the vertex data to the vertex shader's vertex attributes
- Drawing an object in OpenGL would now look something like this:

```
// 0. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. use our shader program when we want to render an object
glUseProgram(shaderProgram);
// 3. now draw the object
someOpenGLFunctionThatDrawsOurTriangle();
```

Draw

- This process needs to repeated every time a object should be drawn
- This becomes a cumbersome process if we have multiple objects (e.g., 5 vertex attributes and 100s of different objects)
- What if there was some way we could store all these state configurations into an object and simply bind this object to restore its state?

Vertex Array Object

- A vertex array object (VAO) bound similar like a VBO, any subsequent vertex attribute calls from that point on will be stored inside the VAO
- Advantage: vertex attribute pointers configuration calls once only, and for the drawing binding the corresponding VAO is enough
- All the states that are set are stored inside the VAO.

Vertex Array Object

Core OpenGL requires the use of a VAO!

Otherwise OpenGL will most likely refuse to draw anything.

Vertex Array Object

- VAO stores the following:
  - Calls to glEnableVertexAttribArray or glDisableVertexAttribArray
  - Vertex attribute configurations via glVertexAttribPointer
  - VBOs associated with vertex attributes by calls to glVertexAttribPointer

Vertex Array Object

- The process to generate a VAO looks simliar to that of a VBO:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

Vertex Array Object

- Bind VAO using glBindVertexArray
- Then bind/configure the corresponding VBO(s) and attribute pointer(s) and then unbind the VAO

```
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Vertex Array Object

- For drawing bind the VAO with the preferred settings before drawing the object:

```
// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

The final Drawing

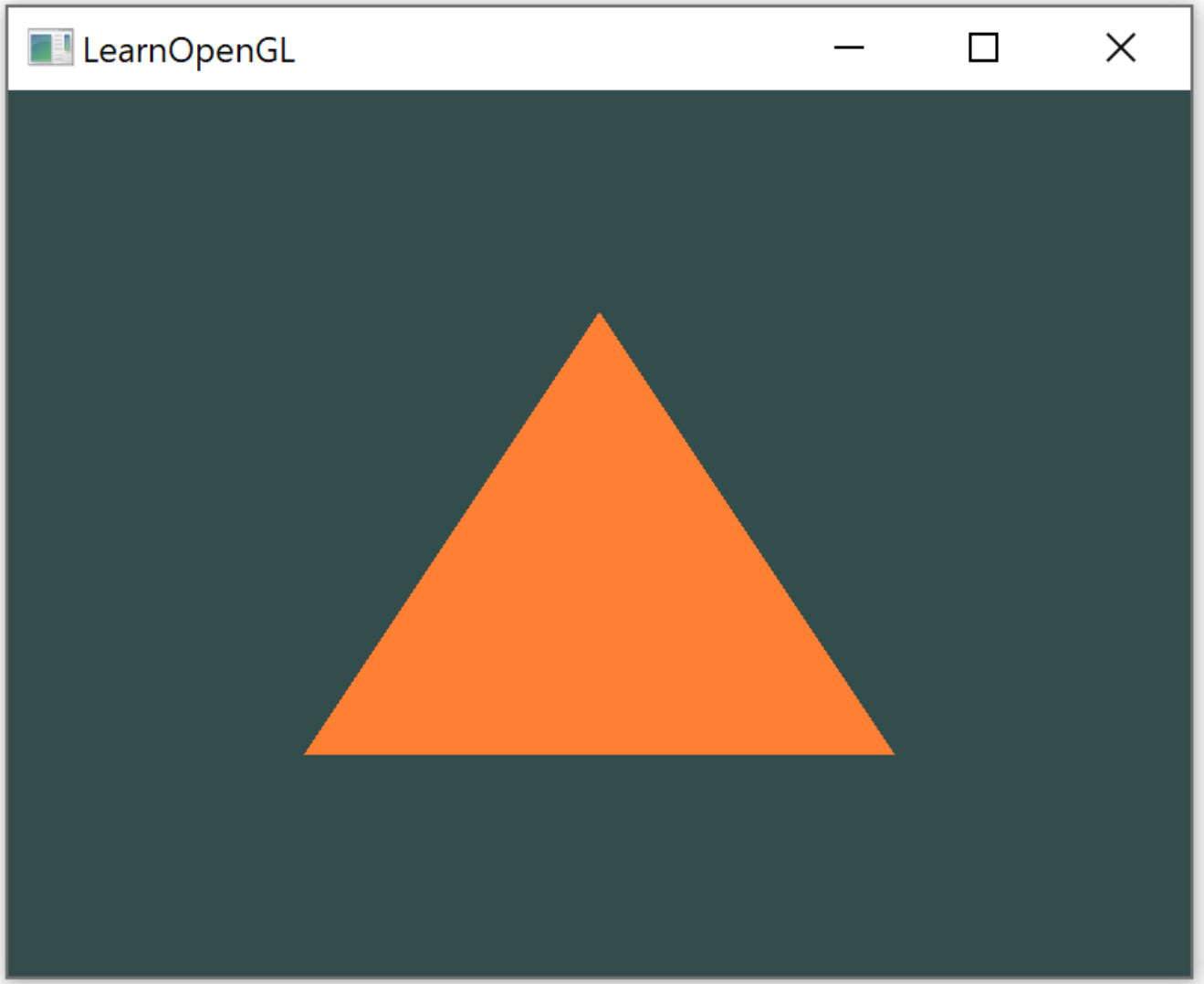
- To draw the objects, OpenGL provides the glDrawArrays function that draws primitives:

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- glDrawArrays function takes as its first argument the OpenGL primitive: GL_TRIANGLES
- Second argument: starting index of the vertex array (0)
- Last argument: how many vertices (3)

F5…

…finally!



Alltogether

```
int main()
{
    // glfw: initialize and configure
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // glfw window creation
    GLFWwindow* window = glfwCreateWindow(800, 600, "LearnOpenGL", NULL, NULL);
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
        return -1;
```

Alltogether

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

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

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

// delete shaders
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Alltogether

```
// set up vertex data (and buffer(s)) and configure vertex attributes
float vertices[] = {-0.5f, -0.5f, 0.0f, 0.5f, -0.5f, 0.0f, 0.0f,  0.5f, 0.0f};

unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Alltogether

```
    // render loop
    while (!glfwWindowShouldClose(window))
    {
        // input
        processInput(window);

        // render
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // draw our first triangle
        glUseProgram(shaderProgram);
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);

        glfwSwapBuffers(window);
        glfwPollEvents();
}
```

Alltogether

```
    // optional: de-allocate all resources once they've outlived their purpose:
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);

    // glfw: terminate, clearing all previously allocated GLFW resources.
    glfwTerminate();
    return 0;
```

Element Buffer Objects

Element Buffer Objects

- Last thing for rendering objects: element buffer objects (EBO)
- For this, look at an example: draw a rectangle instead of a triangle
- Could draw the rectangle using two triangles (OpenGL mainly works with triangles
- This will generate the following set of vertices:

```
float vertices[] = {
        // first triangle
        0.5f, 0.5f, 0.0f, // top right
        0.5f, -0.5f, 0.0f, // bottom right
        -0.5f, 0.5f, 0.0f, // top left
        // second triangle
        0.5f, -0.5f, 0.0f, // bottom right
        -0.5f, -0.5f, 0.0f, // bottom left
        -0.5f, 0.5f, 0.0f // top left
    };
```

Element Buffer Objects

- Overlap on the vertices specified: bottom right = top left
- → An overhead (rectangle consists of 4 vertices, instead of 6)
- Get worse for more complex models
- Better solution: store only unique vertices, then specify the drawing order (store 4 vertices, and the order)

```
float vertices[] = {
        // first triangle
        0.5f, 0.5f, 0.0f, // top right
        0.5f, -0.5f, 0.0f, // bottom right
        -0.5f, 0.5f, 0.0f, // top left
        // second triangle
        0.5f, -0.5f, 0.0f, // bottom right
        -0.5f, -0.5f, 0.0f, // bottom left
        -0.5f, 0.5f, 0.0f // top left
    };
```

Element Buffer Objects

- For this: element buffer objects (EBO)
- EBO is a buffer storing indices for the rectangle:

```
float vertices[] = {
         0.5f,  0.5f, 0.0f,  // top right
         0.5f, -0.5f, 0.0f,  // bottom right
        -0.5f, -0.5f, 0.0f,  // bottom left
        -0.5f,  0.5f, 0.0f   // top left
    };
    unsigned int indices[] = {  // note that we start from 0!
        0, 1, 3,  // first Triangle
        1, 2, 3   // second Triangle
    };
```

Element Buffer Objects

- Create the element buffer object
- Similar to VBO: bind EBO and copy indices into the buffer (glBufferData)
- Like VBO: place calls between a bind and an unbind call
- Specify GL_ELEMENT_ARRAY_BUFFER as the buffer type
- Replace glDrawArrays with glDrawElements (render the triangles with indices)

```
unsigned int EBO;
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Element Buffer Objects

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

- First argument specifies the mode we want to draw in
- Second argument is the number of elements (6 indices)
- Third argument is the type of the indices (GL_UNSIGNED_INT)
- Last argument specify an offset in the EBO

Element Buffer Objects

- glDrawElements function takes its indices from the EBO currently bound to the GL_ELEMENT_ARRAY_BUFFER
- A VAO keeps track of element buffer object bindings
- The EBO currently bound while a VAO is bound, is stored as the VAO's element buffer object
- Binding to a VAO thus also automatically binds its EBO

Element Buffer Objects

VAO stores the glBindBuffer calls when the target is GL_ELEMENT_ARRAY_BUFFER.

VAO also stores its unbind calls so make sure you don’t unbind the element array buffer before unbinding your VAO, otherwise it doesn’t have an EBO configured.

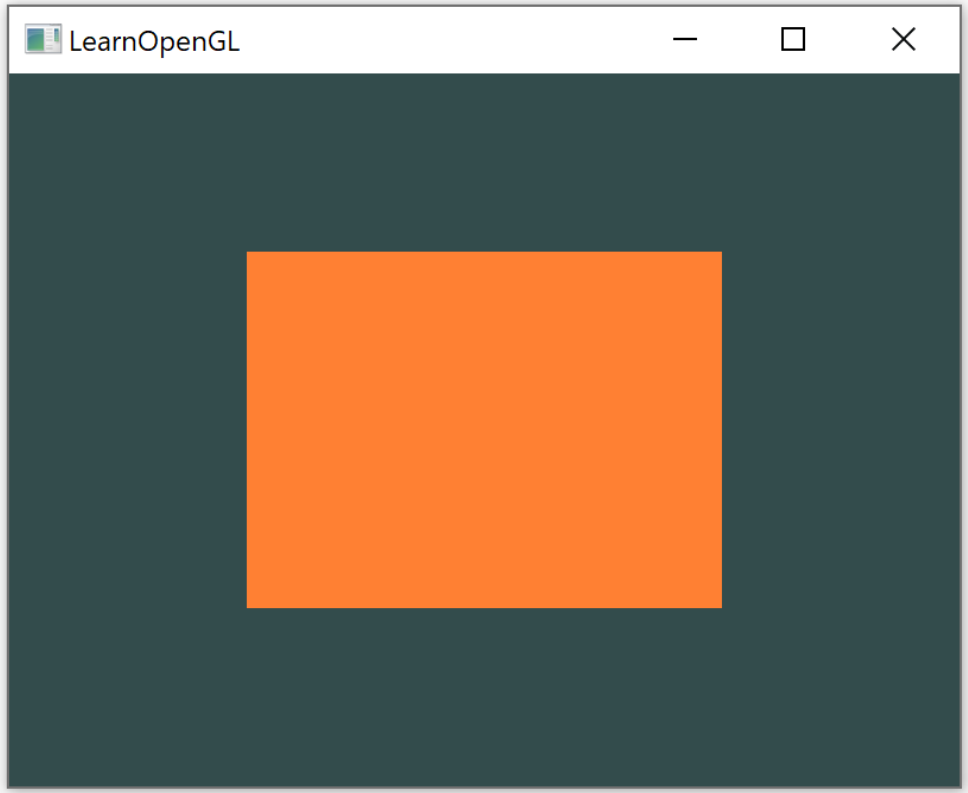
Element Buffer Objects

- The resulting initialization and drawing code:

```
// 1. bind Vertex Array Object
    glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in a element buffer for OpenGL to use
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. then set the vertex attributes pointers
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    [...]
// ..:: Drawing code (in render loop) :: ..
    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

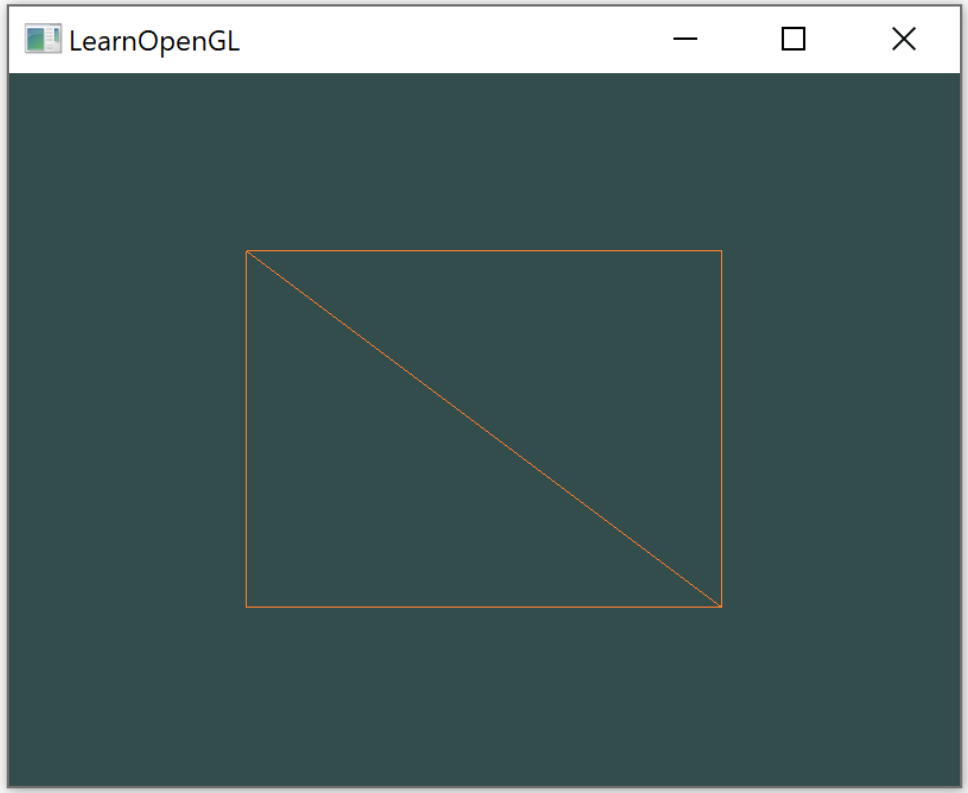
F5…

- The result is a rectangle



Wireframe

- To draw triangles in wireframe mode:
- Set glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
- First argument: apply it to the front and back of all triangles
- Second: draw them as lines
- Any subsequent drawing calls will render the triangles in wireframe mode until: glPolygonMode(GL_FRONT_AND_BACK, GL_FILL).



Draw a Function*

Preparation

- In the previous section, we draw a triangle and a rectangle
- Now, draw a function

Point Set

- First, create an array of size *numPoints*, which contains x and y values:

```
const int numPoints = 50;
float vertices[2* numPoints] = {};
```

Point Set

- The float value should be between [-1,1]
- Here, x={0,1,2,3,…,numPoints-1}

```
for (int i = 0; i < numPoints; i++)
   {
       float x = static_cast<float>(i);
       […]
   }
```

Preparation

- Given x in an interval, how can we transform it to be in [-1,1]?

Preparation

- Given x in an interval, how can we transform it to be in [-1,1]?
- First, we transform it to [0,1]:

$$x \mapsto \frac{x - \min_x}{\max_x - \min_x}$$

$$x \mapsto \frac{x - \min_x}{\max_x - \min_x}$$

Preparation

- Example: $x = \{-1,2,3,6\}$
- $min_x = -1$
- $max_x = 6$
- $\frac{(x-(-1))}{6-(-1)} = \frac{x+1}{7} = \{0, \frac{3}{7}, \frac{4}{7}, 1\}$

Preparation

- x is now in the interval [0,1]
- Multiplying it with 2 and subtracted 1 gives us [-1,1]

$$x \mapsto 2 \cdot \frac{x - \min_x}{\max_x - \min_x} - 1$$

Point Set

- Now our x runs in [-1,1]

```
for (int i = 0; i < numPoints; i++)
   {
       float x = static_cast<float>(i);
       x = 2 * x / (numPoints - 1) - 1;
       […]
   }
```

Point Set

- And we assign the coordinates:

```
for (int i = 0; i < numPoints; i++)
   {
       float x = static_cast<float>(i);
       x = 2 * x / (numPoints - 1) - 1;
       vertices[2 * i] = x;
       vertices[2 * i + 1] = sin(3.14159 * x);
   }
```

The usual Stuff

- Again binding VAO and VBO

```
unsigned int VAO, VBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);

glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

The usual Stuff

- Rendering

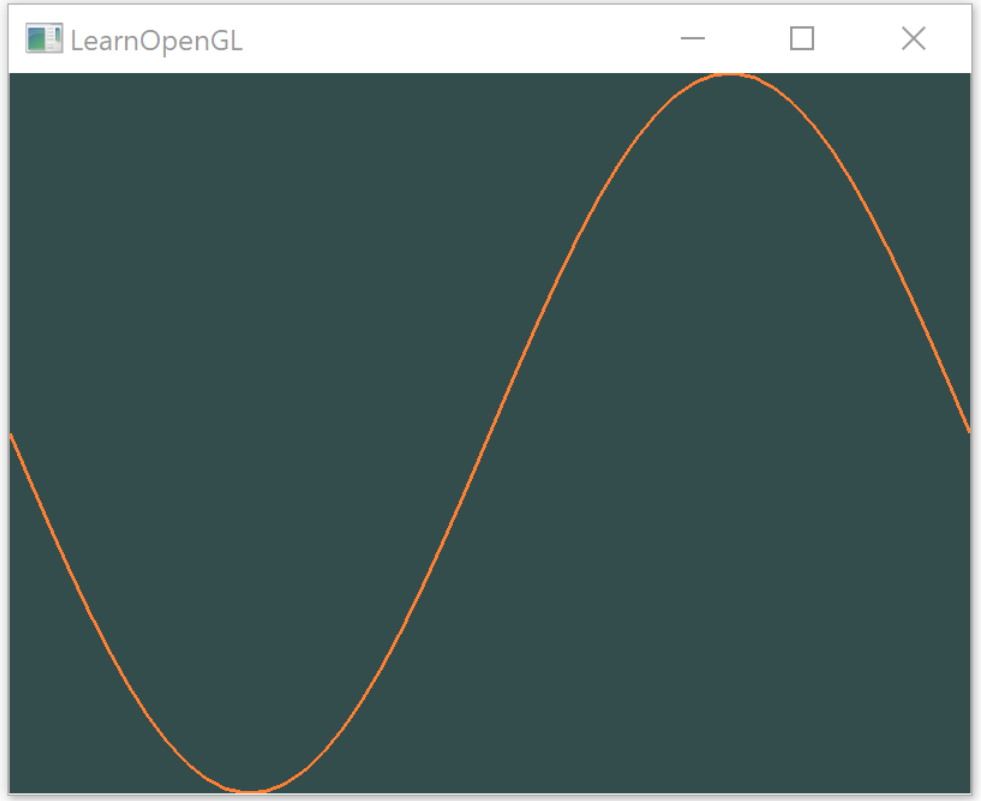
```
glDrawArrays(GL_LINE_STRIP, 0, numPoints);
```

- If the lines are to small, use this before the render loop

```
glLineWidth(3);
```

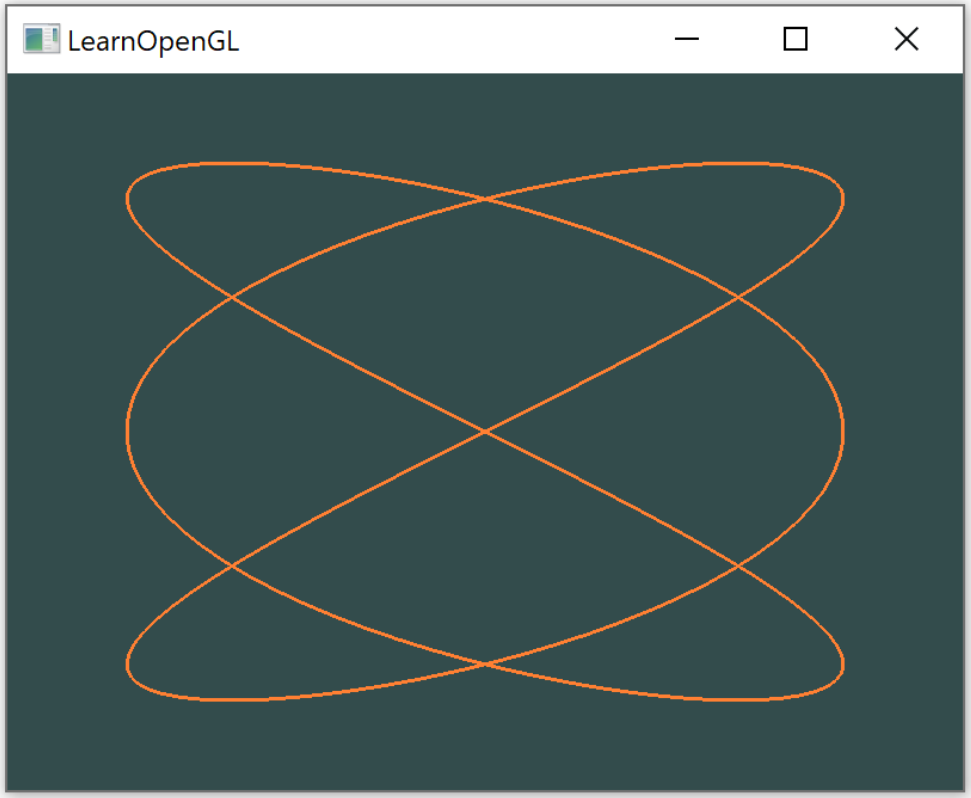
F5…

- This results in a sin function:



Change the Curve

```
const int numPoints = 200;
float vertices[2* numPoints] = {};

for (int i = 0; i < numPoints; i++)
    {
        float x = static_cast<float>(i);
        x = x / (numPoints - 1) * 2 - 1;
        x = x * 3.14159;
        vertices[2 * i] = 0.75 * cos(3 * x);
        vertices[2 * i + 1] = 0.75 * sin(2 * x);
}
```



Two Buffer?

- Now, we want to upload two buffers

```
const int numPoints = 200;
float verticesX[numPoints] = {};
float verticesY[numPoints] = {};


for (int i = 0; i < numPoints; i++)
{
    float t = static_cast<float>(i);
    t = t / (numPoints - 1) * 2 - 1;
    t = t * 3.14159;

    verticesX[i] = 0.75 * cos(3 * t);
    verticesY[i] = 0.75 * sin(2 * t);
 }
```

Two Buffer?

- Now, we want to upload two buffers

```
unsigned int VAO, VBO, VBO2;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &VBO2);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesX), verticesX, GL_STATIC_DRAW);
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, 1 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, VBO2);
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesY), verticesY, GL_STATIC_DRAW);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, 1 * sizeof(float), (void*)0);
glEnableVertexAttribArray(1);
```

Two Buffer?

- Now, we want to upload two buffers

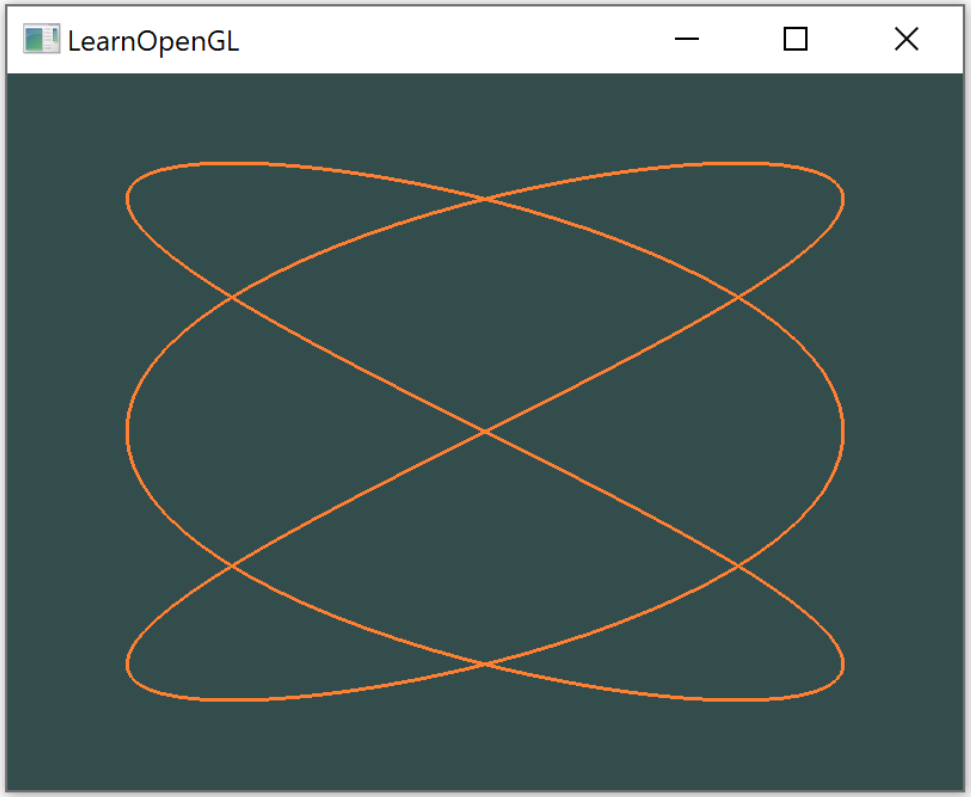
```
unsigned int VAO, VBO, VBO2;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &VBO2);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesX), verticesX, GL_STATIC_DRAW);
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, 1 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, VBO2);
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesY), verticesY, GL_STATIC_DRAW);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, 1 * sizeof(float), (void*)0);
glEnableVertexAttribArray(1);
```

Two Buffer?

- New layout location:

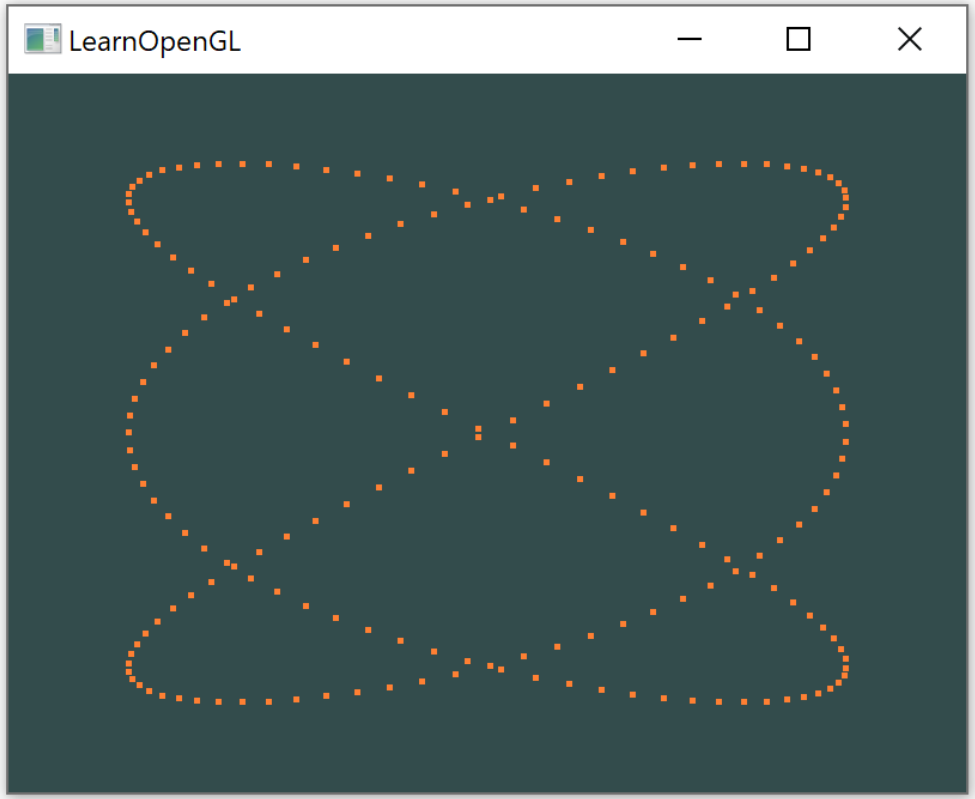
```
const char* vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in float xPos;\n"
"layout (location = 1) in float yPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(xPos, yPos, 1.0, 1.0);\n"
"}\0";
```

Again, the same



Again, the same

```
glDrawArrays(GL_POINTS, 0, numPoints);
```



Color

- Last but not least, we want to add some color

Color

- The fragment shader offers some built-in variables
- For example: vec4 gl_FragCoord;
- gl_FragCoord is a 4-dimensional vector
- The x,y coordinate gives the location of the fragment in window space
- In our case x is between [0,800] and y is in [0,600]
- We can use this to make a nice transition

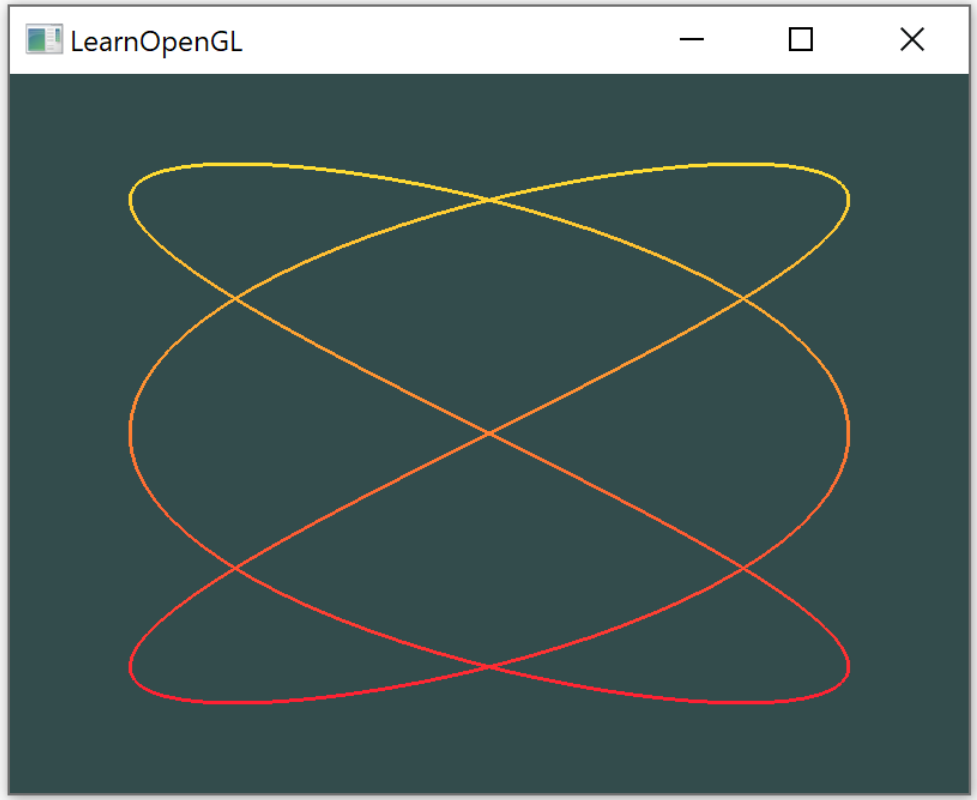
Color

- We use gl_FragCoord for coloring:

```
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f,gl_FragCoord.y/600.0f, 0.2f, 1.0f);\n"
"}\n\0";
```

F5…

- … beautiful
- (Note, we used the fix resolution of 600, resizing the window causes a non consistent coloring)



Questions???