Politecnico di Torino

GRAINS

GRAphics and INtelligent Systems group

Laboratorio 04

OpenGL - Part I

Learning Outcomes

- Learn about the low-level graphic primitives for the creation of interactive graphic applications

Outline

- Creating a window
- First rendering
- Draw a rectangle
- Exercises

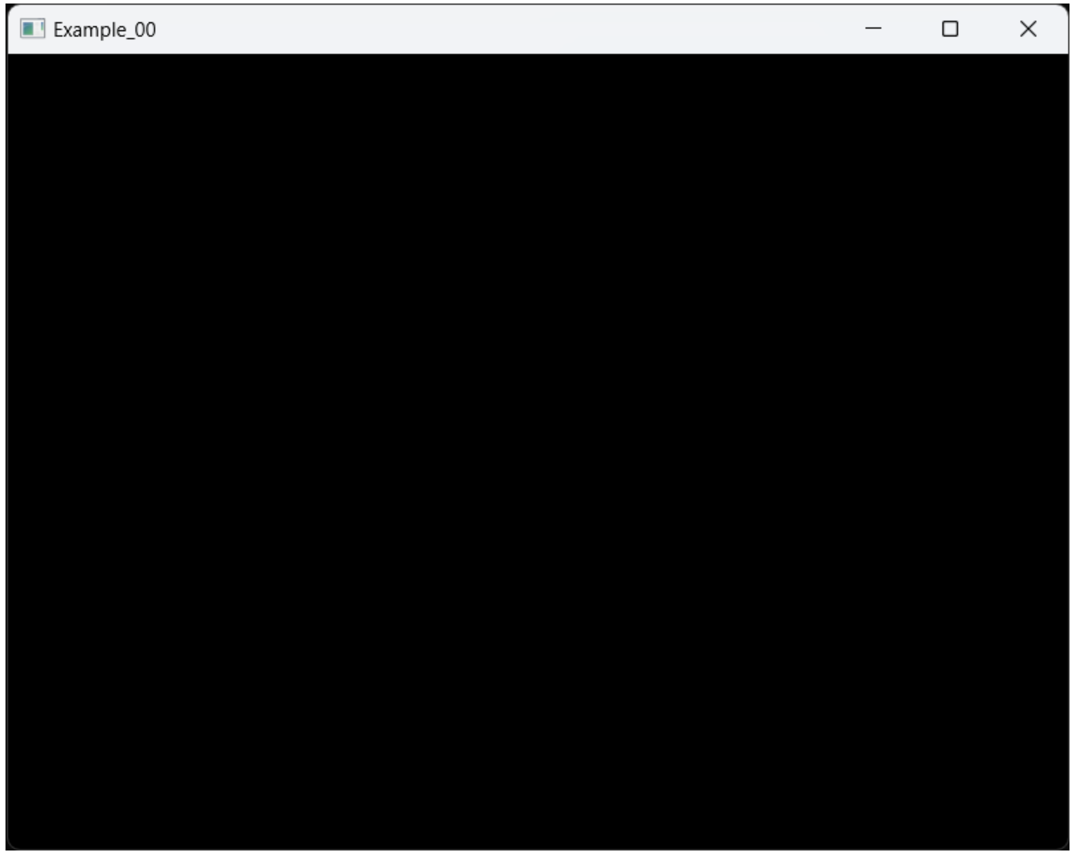
Creating a window

Example0.cpp

The project provided as supplemental material, already contains **<u>all the necessary dependencies</u>**.

To create your first OpenGL application:

- Unzip the content of the archive named **OpenGLApp**.
- Open the project solution file **OpenGLApp.sln**, by taking care to use the Visual Studio Community 2022 version.
- Compile the project and run.



Creating a window

Include files

Let’s see the structure of the project.

GLAD should be included before GLFW, since it contains the required OpenGL headers behind the scenes (like GL/gl.h). Therefore, it must be included before other header files that require OpenGL (like GLFW).

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

Creating a window

Main function

The `main` function instantiates the GLFW window by using the `glfwInit()` and `glfwWindowHint()` functions.

```
int main()
{
    // glfw init and configuration
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

...
```

GLFW: `glfwInit()`

This function initializes the GLFW library. If this function fails, it calls `glfwTerminate()` before returning. If it succeeds, you should call `glfwTerminate()` before the application exits in order to free any resources allocated during or after initialization.

Usage: `int glfwInit(void)`

Returns

- `GLFW_TRUE` if successful,
- `GLFW_FALSE` if an error occurred.

glfwInit: https://www.glfw.org/docs/3.3/group__window.html#ga7d9c8c62384b1e2821c4dc48952d2033

GLFW: `glfwWindowHint()`

This function sets hints for the next call to `glfwCreateWindow()`. The hints, once set, retain their values until changed by a call to this function or `glfwDefaultWindowHints()`, or until the library is terminated. Only integer value hints can be set with this function. There are a number of hints that can be set before the creation of a window and context. Some affect the window itself, others affect the framebuffer or context. The hints are set to their default values each time the library is initialized with `glfwInit()`.

Usage: `void glfwWindowHint(int hint, int value)`

Parameters:

- `hint`: The window hint to set.
- `value`: The new value of the window hint.

glfwWindowHint: https://www.glfw.org/docs/3.3/group__init.html#ga317aac130a235ab08c6db0834907d85e

windows hints: https://www.glfw.org/docs/3.3/window_guide.html#window_hints

GLFW: `glfwWindowHint()`

Since the focus of this course is OpenGL version 3.3, the major (`GLFW_CONTEXT_VERSION_MAJOR`) and minor (`GLFW_CONTEXT_VERSION_MINOR`) version are set to 3. In this way, GLFW can make the proper arrangements when creating the OpenGL context. The call of these two functions also ensure that when a user does not have the proper OpenGL version installed, GLFW fails to run.

Moreover, the OpenGL core-profile is enabled, thus making accessible only a smaller subset of OpenGL features without backwards-compatible features no longer supported.

Creating a window

Window creation with GLFW

Afterwards a window object is created. This object holds all the windowing data and is required by most of GLFW's other functions.

```
// glfw window creation
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
"LearnOpenGL", NULL, NULL);
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
glfwMakeContextCurrent(window);
```

GLFW: `glfwCreateWindow()`

This function creates a window and its associated OpenGL context. Most of the options controlling how the window and its context should be created are specified with window hints. To create a full screen window, you need to specify the monitor the window will cover. If no monitor is specified, the window will be windowed mode. By default, newly created windows use the placement recommended by the window system. To create the window at a specific position, make it initially invisible using the GLFW_VISIBLE window hint, set its position and then show it.

Usage: `GLFWwindow* glfwCreateWindow(int width, int height, const char *title, GLFWmonitor* monitor, GLFWwindow *share)`

glfwCreateWindow: https://www.glfw.org/docs/3.3/group__window.html#ga3555a418df92ad53f917597fe2f64aeb

GLFW: `glfwCreateWindow()`

Parameters:

- `width, height`: The desired width and height, in screen coordinates, of the window. Must be greater than zero.
- `title`: The initial, UTF-8 encoded window title.
- `monitor`: The monitor to use for full screen mode, or NULL for windowed mode.
- `share`: The window whose context to share resources with, or NULL to not share resources.

Returns

- `GLFWwindow:` the handle of the created window,
- `NULL:` if an error occurred.

glfwCreateWindow: https://www.glfw.org/docs/3.3/group__window.html#ga3555a418df92ad53f917597fe2f64aeb

GLFW: `glfwCreateWindow()`

To enable full screen mode, add the code below to retrieve the settings of the primary monitor and replace the how the `glfwCreateWindow()` function is called.

```
// Retrieve the primary monitor
GLFWmonitor* primaryMonitor = glfwGetPrimaryMonitor();

// Retrieve the video setting of the monitor
const GLFWvidmode* mode = glfwGetVideoMode(primaryMonitor);

// glfw window creation
// --------------------
GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
"Example_00", primaryMonitor, NULL);
```

Creating a window

GLAD

In the previous slides, it was mentioned that GLAD manages function pointers for OpenGL. Therefore, GLAD has to be initialized before calling any OpenGL function.

```cpp
// glad: load all OpenGL function pointers
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
```

Creating a window

Set the viewport

Before rendering, it is necessary to tell OpenGL the size of the rendering window. In this way, OpenGL can compute the data and coordinates with respect to the window. The operation can be executed with the glViewport function:

```
glViewport(0, 0, 800, 600);
```

`glViewport()`

This function sets the viewport, by specifying the affine transformation of x and y from normalized device coordinates to window coordinates.

Parameters:

- `x,y`: Specify the lower left corner of the viewport rectangle, in pixels. The initial value is (0,0).
- `width,height`: Specify the width and height of the viewport. When a GL context is first attached to a window, width and height are set to the dimensions of that window.

glViewport: https://docs.gl/gl3/glViewport

Creating a window

Resize callback

However, the moment a user resizes the window the viewport should be adjusted as well. It is possible to register a callback function on the window that gets called each time the window is resized. This resize callback function has the following prototype:

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
```

Creating a window

Resize callback

The framebuffer size function takes a GLFW window as its first argument and two integers indicating the new window dimensions. Whenever the window changes in size, GLFW calls the following function and fills in the proper arguments for you to process.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    glViewport(0, 0, width, height);
}
```

The function can be registered in the main

```
ĝľğxŞêţƒGsǻņêčųğğêsŞîċêCǻľľčǻçl xîŋđộx  ğsǻņêčųğğês şîċê çǻľľčǻçl
```

Creating a window

Render loop

In its current state, the application draws a single image and then immediately quits and closes the window.

In order to make the application keep drawing images and handling user input until the program has been explicitly closed, it is necessary to create a while loop (in the following referred to as the render loop).

The render loop keeps on running until an instruction stops the application. The following code shows a very simple render loop:

```
// render loop
while (!glfwWindowShouldClose(window))
{
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

GLFW: `glfwSwapBuffers()`

This function swaps the front and back buffers of the specified window.

Usage: `void glfwSwapBuffers (GLFWwindow * window)`

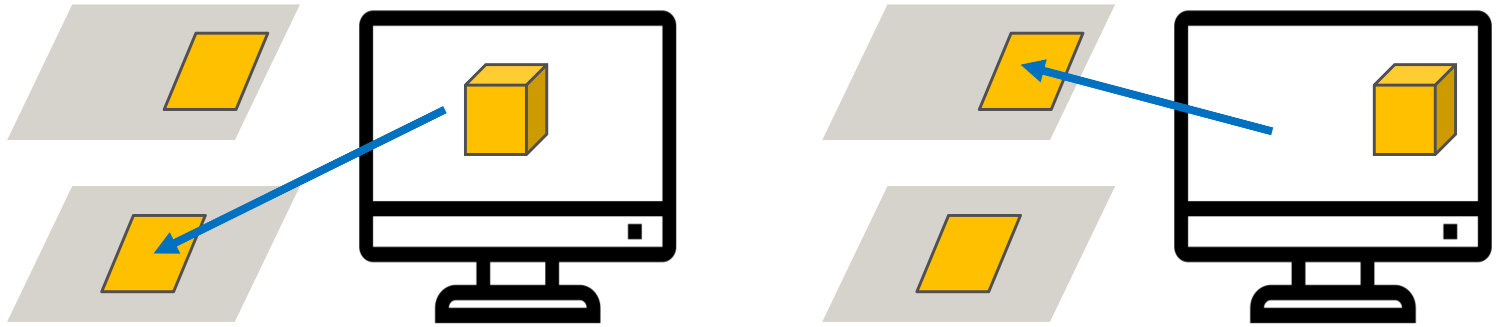
Parameters:

- `window:` The window whose buffers to swap

glfwSwapBuffers(): https://www.glfw.org/docs/3.0/group__context.html#ga15a5a1ee5b3c2ca6b15ca209a12efd14

Double buffer

When an application draws in a single buffer the resulting image may display flickering issues. This is because the resulting output image is not drawn in an instant, but drawn pixel by pixel and usually from left to right and top to bottom. Because this image is not displayed at an instant to the user while still being rendered to, the result may contain artifacts. To circumvent these issues, windowing applications apply a double buffer for rendering. The front buffer contains the final output image that is shown at the screen, while all the rendering commands draw to the back buffer. As soon as all the rendering commands are finished, the back buffer is swapped with the front buffer so the image can be displayed without still being rendered to, removing all the aforementioned artifacts.



GLFW: `glfwPollEvents()`

This function processes only those events that have already been received and then returns immediately. Processing events will cause the window and input callbacks associated with those events to be called.

Usage: `void glfwPollEvents(void)`

glfwPollEvents: https://www.glfw.org/docs/3.0/group__window.html#ga37bd57223967b4211d60ca1a0bf3c832

Creating a window

Render loop

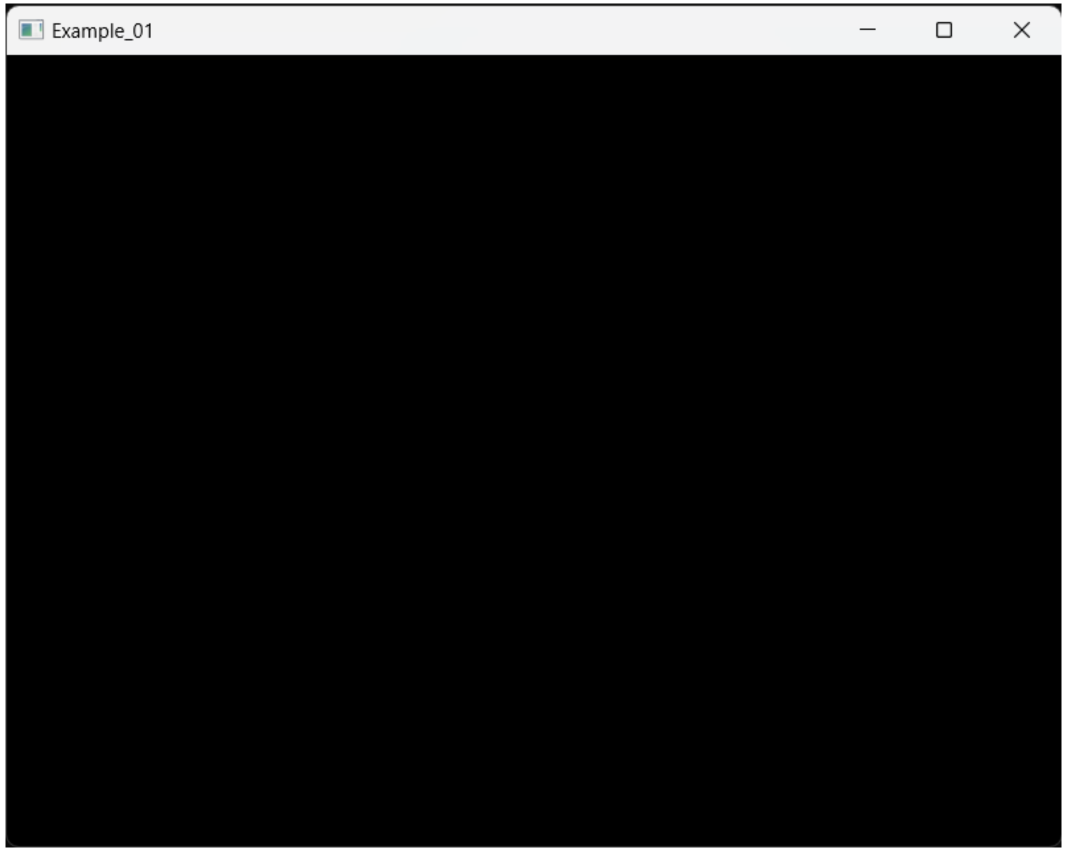
As soon as the application exits the render loop it is fundamental to properly clean/delete all of GLFW's resources that were allocated. This operation can be performed via the `glfwTerminate()` function that is called at the end of the main function.

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

Managing user input

Example1.cpp

In this example, the GLFW's functions are used to manage the user input. To this aim, a function named `processInput()` is defined.



Managing user input

processInput function

The `processInput()` function contains the instructions to recognize and manage the user input. In particular, it makes use of the `glfwGetKey` function. The GLFW's function takes the windows as input together with a key and returns its status. In the following example, the application is closed by calling the function `glfwSetWindowShouldClose()`, when the user press the escape key.

```
void processInput(GLFWwindow* window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}
```

GLFW: `glfwGetKey()`

This function returns the last state reported for the specified key to the specified window. The returned state is one of `GLFW_PRESS` or `GLFW_RELEASE`. Do not use this function to implement text input.

Usage: `int glfwGetKey(GLFWwindow * window, int key)`

Parameters:

- `window`: The desired window.
- `key`: The desired keyboard key. GLFW_KEY_UNKNOWN is not a valid key for this function.

Returns:

- `GLFW_PRESS`
- `GLFW_RELEASE`

glfwGetKey(): https://www.glfw.org/docs/3.3/group__input.html#gadd341da06bc8d418b4dc3a3518af9ad2

GLFW: `glfwGetKey()`

A comprehensive list of key codes can be found at:

https://www.glfw.org/docs/3.3/group__keys.html

Macros	Keycode
`GLFW_KEY_RIGHT`	262
`GLFW_KEY_LEFT`	263
`GLFW_KEY_DOWN`	264
`GLFW_KEY_UP`	265
`GLFW_KEY_W`	87
`GLFW_KEY_A`	65
`GLFW_KEY_S`	83
`GLFW_KEY_D`	68
`GLFW_KEY_SPACE`	32

Managing user input

processInput function

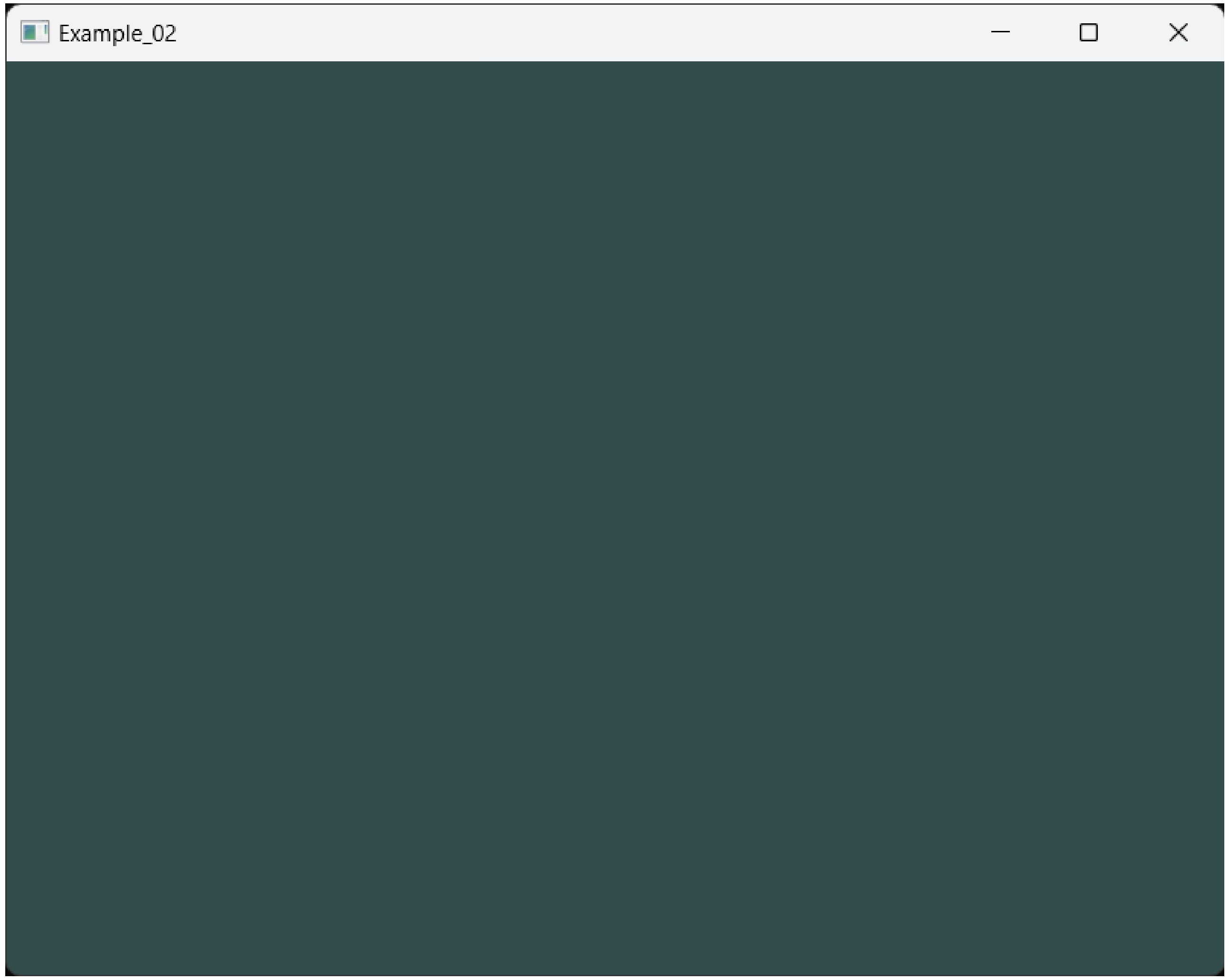
The `processInput()` function is called every iteration of the render loop. This approach represents an easy way to check for specific key presses and react accordingly every frame. An iteration of the render loop is more commonly called a frame.

```
// render loop
while (!glfwWindowShouldClose(window))
{
     processInput(window);
     glfwSwapBuffers(window);
    glfwPollEvents();
}
```

First rendering

Example2.cpp

All the rendering commands should be added in the render loop, in order to execute them at each frame.



First rendering

Rendering commands

To test if this approach works, it is possible to look at simple rendering functions. For instance, the OpenGL functions `glClear()` can be leveraged to clear the screen with a color of our choice. In particular, this function makes the application passes to the buffer bits mode to specify the buffer to be cleared. The possible bits that can be set are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` and `GL_STENCIL_BUFFER_BIT`. For this example, the color buffer bit is used.

Note that the color used to clear the screen has to be specified using the `glClearColor()`. Whenever the `glClear()` is called to clear the color buffer, the entire color buffer will be filled with the color configured by the `glClearColor()` function.

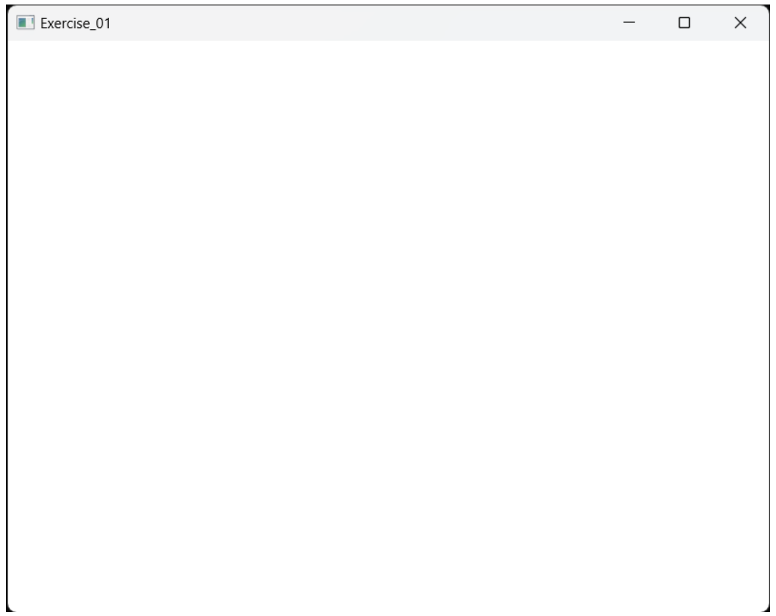
First rendering

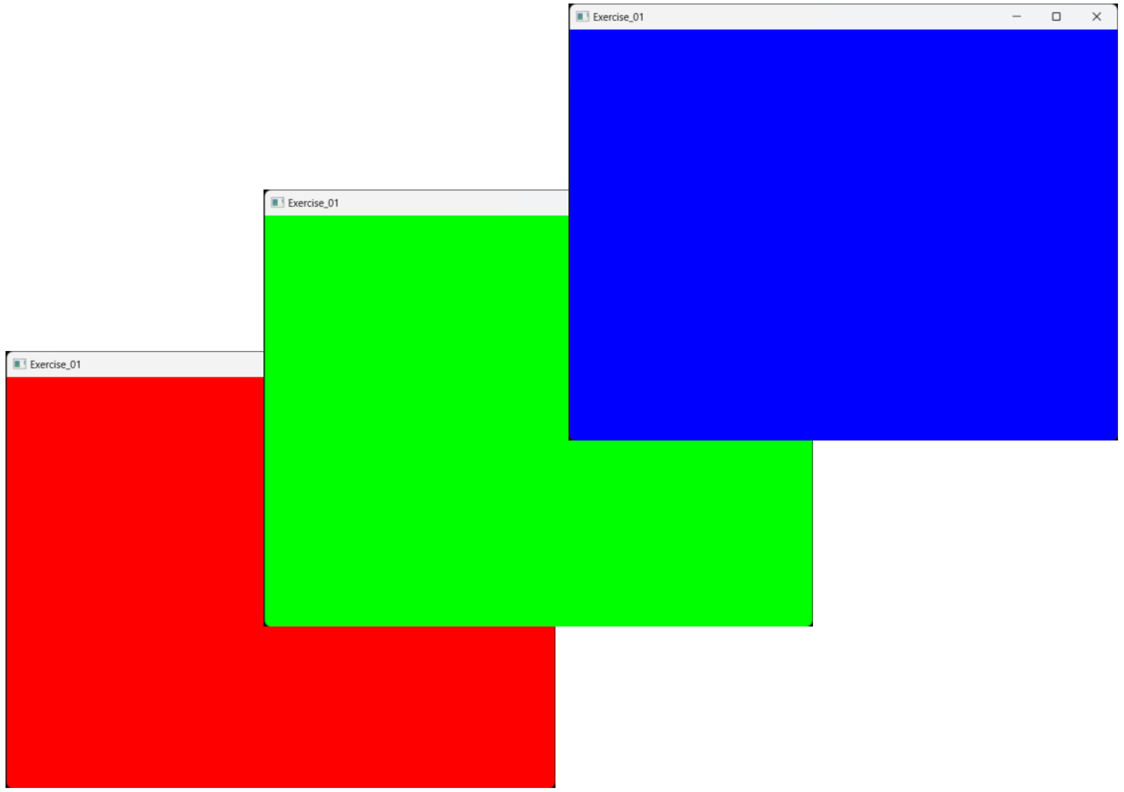
Example2.cpp

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

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

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

Exercise #1

Modify the `processInput()` function to change the background color using the keys: R(red), G(green), B(blue).

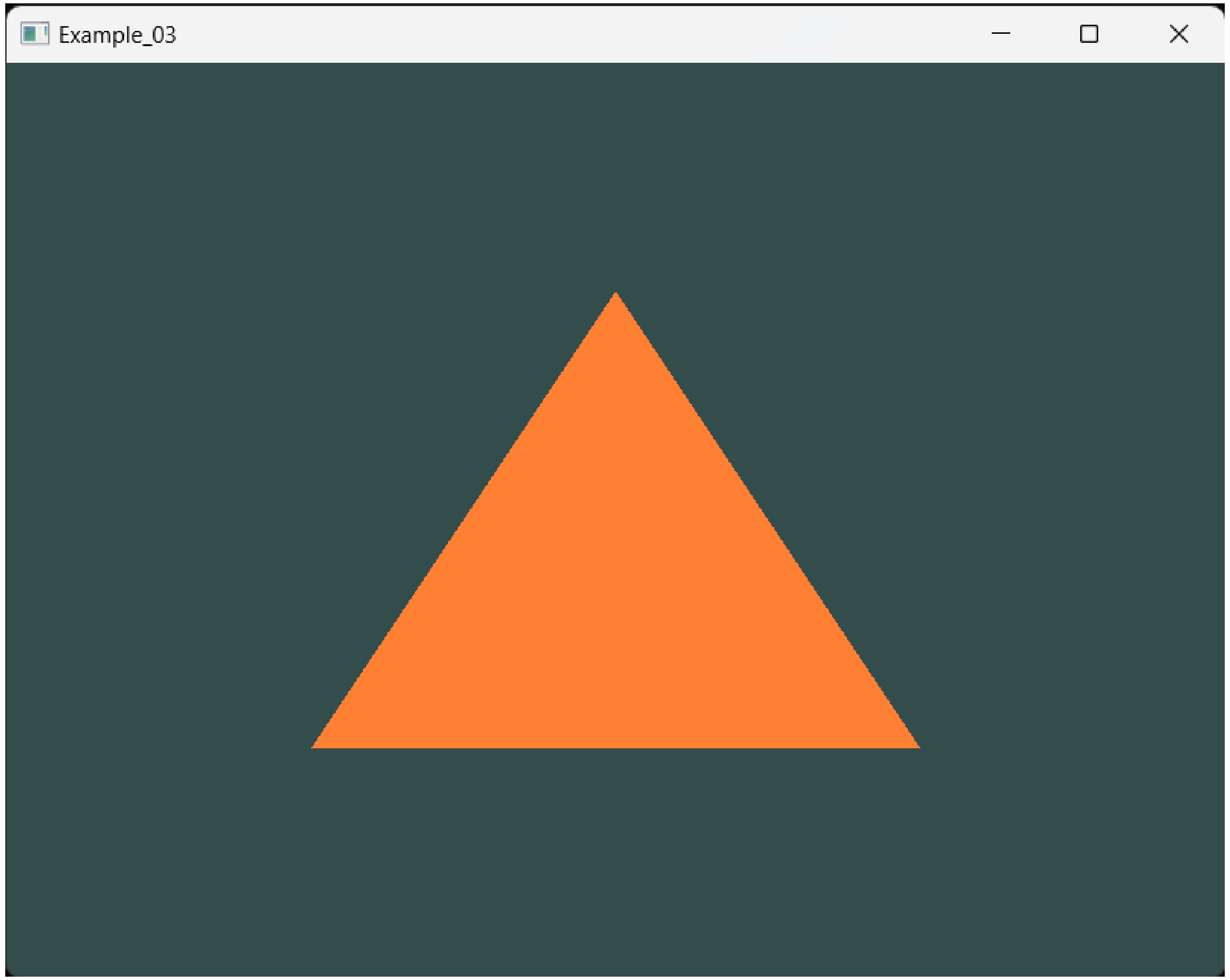




Draw a triangle

Example3.cpp

Before drawing a triangle, it is necessary to know more about low-level graphic programming.



OpenGL graphics pipeline

In OpenGL everything is in 3D space, but the screen is a 2D array of pixels. For this reason, it is necessary to transform all the 3D coordinates to 2D pixels that fit the screen. This process is managed by the **graphics pipeline** of OpenGL. The graphics pipeline takes as input a set of 3D coordinates and transforms these to colored 2D pixels on screen

OpenGL graphics pipeline

The graphics pipeline can be divided into several steps where each step requires the output of the previous step as its input. All of these steps are highly specialized and, usually, they can easily be executed in parallel. Because of their parallel nature, graphics cards of today have thousands of small processing cores to quickly process data within the graphics pipeline. The processing cores run small programs on the GPU for each step of the pipeline. These small programs are called **shaders**.

Some of these shaders are configurable by the developer, who can override the existing default shaders with own custom shader.

This approach gives fine-grained control over specific parts of the pipeline.

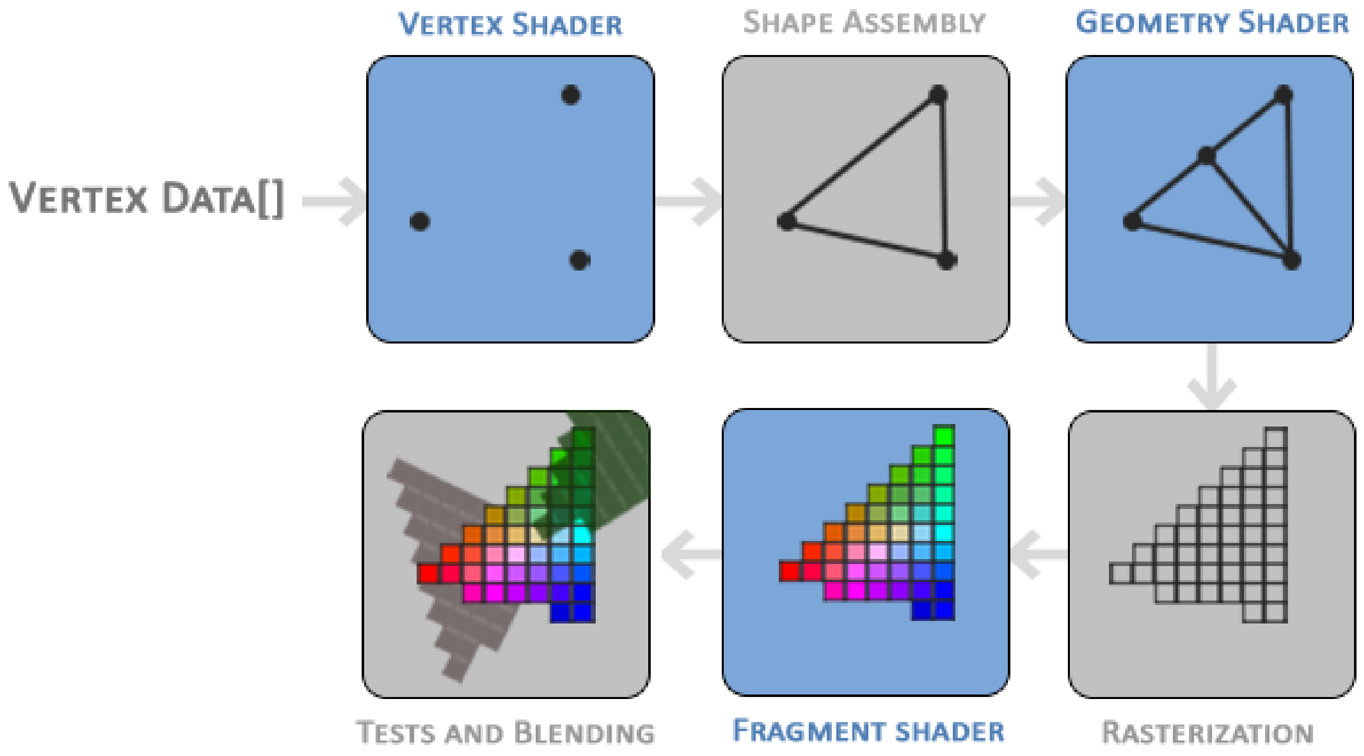
Shaders are written in the **OpenGL Shading Language** (GLSL).

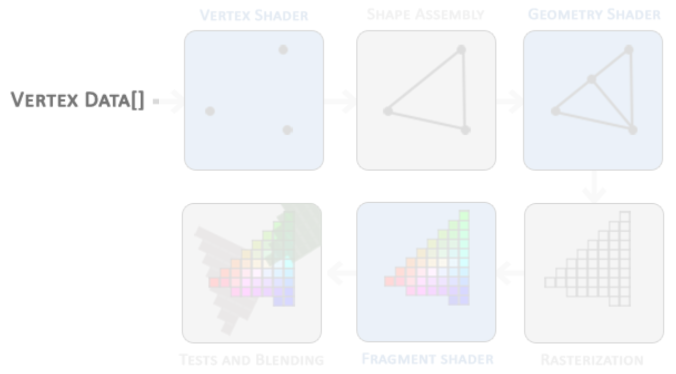
OpenGL Shading Language (GLSL): https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language

OpenGL graphics pipeline

The image shows an abstract representation of all the stages of the graphics pipeline. The blue sections are those in which it is possible to inject custom shaders.

Each section handles a specific part of the process that converts vertex data to a fully rendered pixel.



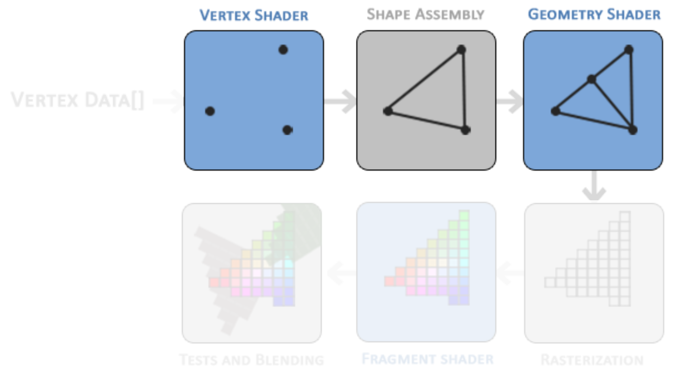


OpenGL graphics pipeline

A list of 3D coordinates is passed as input to the graphics pipeline. In this example, the list contains the three vertices forming a triangle. The vertices are passed in an array called **Vertex Data**. The vertex data is a collection of vertices, i.e., the available data for each 3D coordinate. The vertex's data contains vertex attributes such as the 3D positions or color values.

In order for OpenGL to know what to make of coordinates and color values, OpenGL has to be instructed on what kind of render type should be activated on the provided data. For instance, data can be rendered as a collection of points, triangles, or a line. These instructions or hints are called **primitives** and are given to OpenGL while calling the drawing commands. Some of these hints are `GL_POINTS`, `GL_TRIANGLES`, and `GL_LINE_STRIP`.

OpenGL graphics pipeline

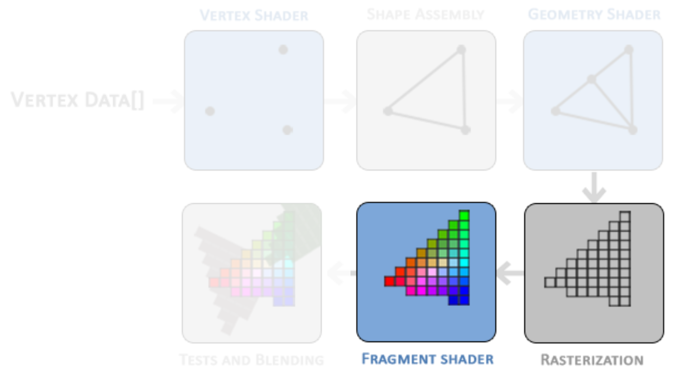


The first part of the pipeline is the **vertex shader**. The Vertex Shader is the programmable shader stage in the rendering pipeline, that handles the processing of individual vertices. A vertex shader receives a single vertex from the vertex stream and generates a single vertex to the output vertex stream.

The output of the vertex shader stage is optionally passed to the **geometry shader**. The geometry shader takes as input a collection of vertices that form a primitive and generates 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.

The **primitive/shape assembly** stage takes as input all the vertices from the vertex (or geometry) shader that form one or more primitives and assembles all the point(s) in the primitive shape given; in this case a triangle.

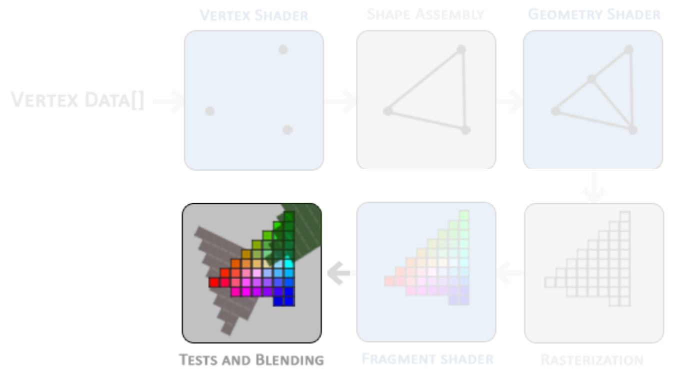
OpenGL graphics pipeline



The output of the geometry shader is then passed on to the **rasterization stage.** During this stage the resulting primitive(s) is(are) mapped to the corresponding pixels on the final screen, resulting in fragments, i.e., all the data required for OpenGL to render a single pixel. Fragments are then used by the Fragment shader. Before the fragment shaders run, clipping is performed. Clipping discards all fragments that are outside the view, increasing performance.

The main purpose of the **fragment shader** is to calculate the final color of a pixel. The fragment shader usually contains data about the 3D scene (e.g., lights, shadows, color of the light, etc.) that are used to calculate the final pixel color.

OpenGL graphics pipeline

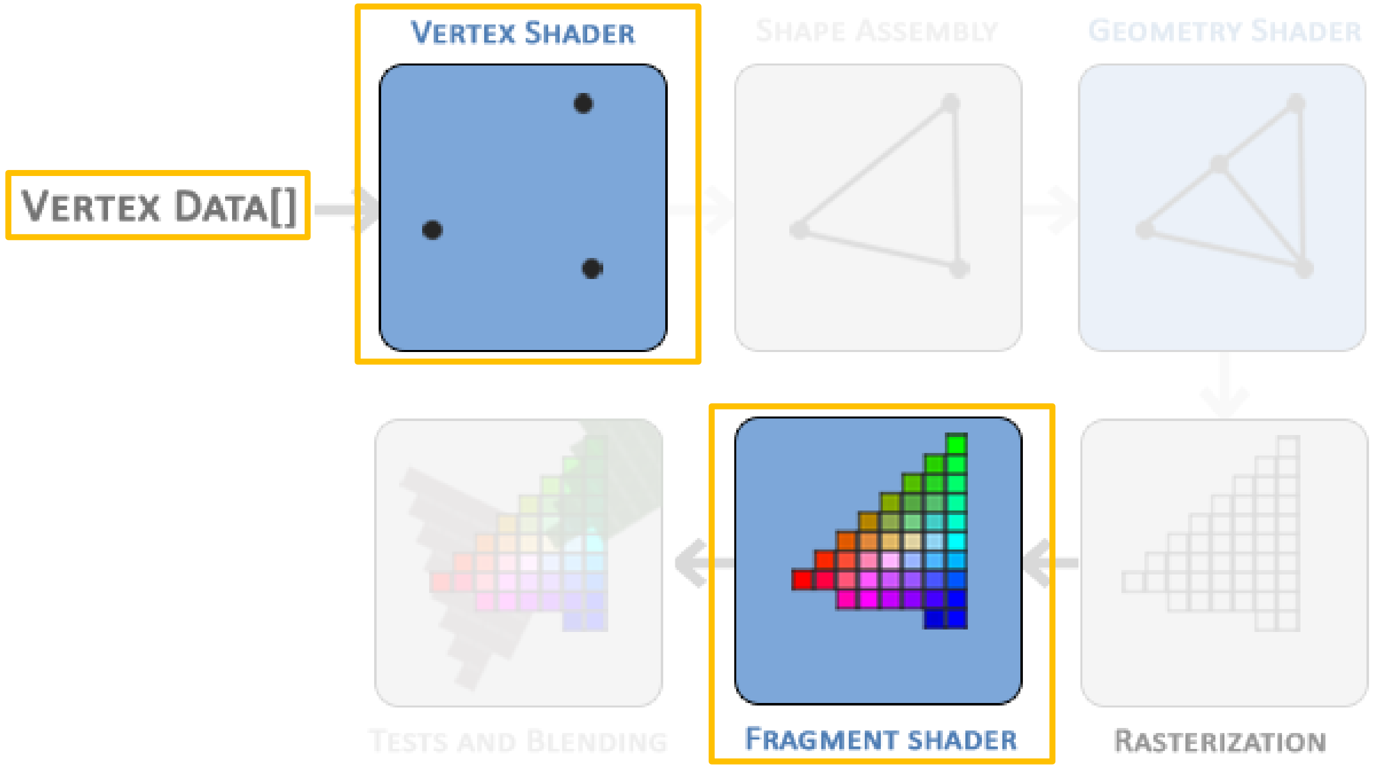


The object then pass through the **last stage** in which the alpha test and blending are performed. This stage checks the corresponding depth (and stencil) value of the fragment and uses those values to check if the resulting fragment is in front or behind other objects and should be discarded accordingly.

The stage also checks for alpha values (alpha values define the opacity of an object) and blends the objects accordingly. So even if a pixel output color is calculated in the fragment shader, the final pixel color could still be something entirely different when rendering multiple triangles.

OpenGL graphics pipeline

The use of modern OpenGL requires to define at least a vertex and fragment shader of our own, as there are no default vertex/fragment shaders on the GPU.



Draw a triangle

Vertex input

To start drawing something, some input vertex data is needed. OpenGL is a 3D graphics library, as a result all coordinates are specified in 3D (x, y and z coordinate). OpenGL processes 3D coordinates specified on a range between -1.0 and 1.0 for all the 3 axes. All coordinates within this so-called **normalized device coordinates** range will end up visible on screen (and all coordinates outside this region won't).

Because the goal is to render a single triangle, a total of three vertices has to be specified, with each vertex having a 3D position. The triangle is defined in normalized device coordinates (the visible region of OpenGL) using a float array:

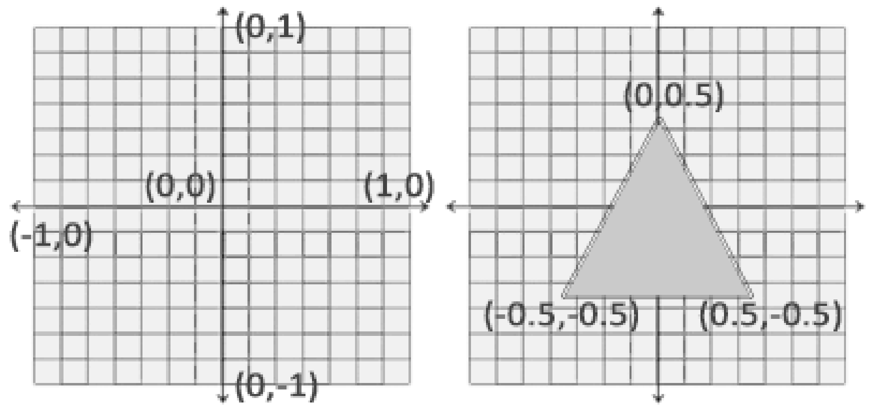
```
// set up vertex data (and buffer(s)) and configure vertex attributes
float vertices[] = {
    -0.5f, -0.5f, 0.0f, // left
     0.5f, -0.5f, 0.0f, // right
     0.0f,  0.5f, 0.0f  // top
};
```

Draw a triangle

Normalized Device Coordinates (NDC)

The normalized device coordinates are a small space where the x, y, and z values vary from -1.0 to 1.0. Any coordinates that fall outside this range will be discarded/clipped and won't be visible on the screen.

Unlike usual screen coordinates, the positive y-axis points in the up-direction and the (0,0) coordinates are at the center of the graph, instead of top-left. The NDC coordinates will then be transformed to screen-space coordinates via the **viewport transform** using the data provided with `glViewport`. The resulting screen-space coordinates are then transformed to fragments as inputs to the fragment shader.



➡ **Viewport transform** ➡ Screen-space coordinates

NDC Coordinates

Draw a triangle

Vertex input

After defining the vertex data, it has to be sent as input to the first process of the graphics pipeline, i.e., the vertex shader. This is done by:

- creating memory 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.

The vertex shader then processes the vertices stored in its memory. The memory is managed via the so-called **vertex buffer objects** (VBO). The advantage of using buffer objects is that large batches of data can be sent all at once to the graphics card, and kept it there if there's enough memory left, without having to send data one vertex at a time. Sending data to the graphics card from the CPU is relatively slow, so is better to send as much data as possible at once. Once the data is in the graphics card's memory the vertex shader has almost instant access to the vertices making it extremely fast.

Draw a triangle

Vertex input

A vertex buffer object is an OpenGL object, and it should have a unique ID corresponding to that buffer. To generate a buffer ID:

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

OpenGL has many types of buffer objects. The buffer type for a vertex buffer object is `GL_ARRAY_BUFFER`. OpenGL allows developers to bind to several buffers at once as long as they have a different buffer type. To bind the newly created buffer to the `GL_ARRAY_BUFFER` target:

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

Draw a triangle

Vertex input

After the binding, any buffer calls made (on the `GL_ARRAY_BUFFER` target) will be used to configure the currently bound buffer, which is VBO. Then, the `glBufferData()` function can be called to copy the previously defined vertex data into the buffer's memory:

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

glBufferData()

This function creates and initializes a buffer object's data store for the buffer object currently bound. Any pre-existing data store is deleted. The new data store is created with the specified size in bytes and usage.

Usage: `void glBufferData(Glenum target,Glsizeiptr size, const GLvoid * data,GLenum usage);`

Parameters:

- `target`: Specifies the target buffer object. The symbolic constant must be `GL_ARRAY_BUFFER`, `GL_COPY_READ_BUFFER`, `GL_COPY_WRITE_BUFFER`, `GL_ELEMENT_ARRAY_BUFFER`, `GL_PIXEL_PACK_BUFFER`, `GL_PIXEL_UNPACK_BUFFER`, `GL_TEXTURE_BUFFER`, `GL_TRANSFORM_FEEDBACK_BUFFER`, or `GL_UNIFORM_BUFFER`.
- `size`: Specifies the size in bytes of the buffer object's new data store.

glBufferData(): https://docs.gl/gl3/glBufferData

`glBufferData()`

Parameters:

- `data`: Specifies a pointer to data that will be copied into the data store for initialization, or NULL if no data is to be copied.
- `usage`: Specifies the expected usage pattern of the data store. The symbolic constant must be GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY, GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, or GL_DYNAMIC_COPY.

glBufferData(): https://docs.gl/gl3/glBufferData

Draw a triangle
Vertex shader

Until now the vertex data are stored within memory on the graphics card and managed with a vertex buffer object named VBO. Next, a vertex and fragment shader have to be created to actually processes this data. Modern OpenGL requires to configure at least a vertex and fragment shader to do some rendering.

Starting from the vertex shader, the first thing to do is to write the shader in GLSL, and then compile it to make it usable by the OpenGL application. The code below implements a 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);
}
```

Draw a triangle

Vertex shader

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

Each shader begins with a declaration of its version. Since OpenGL 3.3 and higher the version numbers of GLSL match the version of OpenGL (GLSL version 420 corresponds to OpenGL version 4.2 for example). It is also possible to explicitly mention that the core profile functionality are used.

Draw a triangle

Vertex shader

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

Next, all the input vertex attributes are declared in the vertex shader with the `in` keyword. In this example, only the position data are considered, hence a single vertex attribute is enough. GLSL has a vector datatype `vec–`. 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` respectively. Note that the `vec.w` component is not used as a position in space but is used for perspective division. The number of floats contains is specified on the postfix digit. Since each vertex has a 3D coordinate a `vec3` input variable is created with the name `aPos`. The command `location = 0` is used to specify the location of the input variable.

Draw a triangle

Vertex shader

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

To set the output of the vertex shader, it is necessary to assign the position data to the predefined `gl_Position` variable. Since the input (`vec3`) is a vector of size 3, a cast is needed to a vector of size 4. This is done by inserting the vec3 values inside the constructor of vec4 and set its w component to 1.0f.

The current vertex shader is very simple, but in real applications the input data is usually not already in normalized device coordinates, so the input data are transformed to coordinates that fall within OpenGL visible region.

Draw a triangle

Compiling a shader

The source code of the vertex shader is first stored in a const C string.

```
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";
```

In order for OpenGL to use the shader, it has to dynamically compile it at run-time from its source code. To this aim, a shader object is created, again referenced by an ID, by using the `glCreateShader()` function and by storing it as an unsigned int.

```
ụŋṣîĝŋêđ îŋʧ ŵêsʧêỵŞḥǻđês   ĝľCsêǻʧêŞḥǻđês ĞĽ ΛÉŖŢÉŶ ŞȞADÉŖ
```

The type of shader to be created (in this example a vertex shader) is passed as an argument to the `glCreateShader()` function.

Draw a triangle

Compiling a shader

The next step is attaching the shader source code to the shader object and compile the shader.

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

To check if compilation was successful after the call to `glCompileShader` or to read the error code if the compilation fails

```
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;
}
```

Draw a triangle

Fragment shader

The fragment shader is in charge of calculating the color output of pixels. Colors in computer graphics are represented as an array of 4 values: the red, green, blue and alpha (opacity) component, commonly abbreviated to RGBA. When defining a color in OpenGL or GLSL the strength of each component is set with a value between 0.0 and 1.0. The code below implements a basic fragment shader that always output an orange-ish color.

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

Draw a triangle

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. The output is a vector of size 4 that defines the final color output. The output values are declared with the `out` keyword, that, in this example is named `FragColor`.

Draw a triangle

Fragment shader

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

Next, the vec4 is assigned to the color output as an orange color with an alpha value of 1.0 (1.0 being completely opaque).

Draw a triangle

Compiling the fragment shader

The process for compiling a fragment shader is similar to the vertex shader, although this time we use the `GL_FRAGMENT_SHADER` constant as the shader type:

```
// fragment shader
unsigned 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;
}
```

Draw a triangle

Shader program

Both the shaders are now compiled, and the only thing left to do is link both shader objects into a **shader program** that can be used for rendering. A shader program object is the final linked version of multiple shaders combined. To use the recently compiled shaders, it is necessary to link them to a shader program object and then activate this shader program when rendering objects. The activated program shaders will be used when render calls are issued.

When linking the shaders into a program, the outputs of each shader are linked to the inputs of the next shader. Linking errors can occur if the outputs and inputs do not match.

Draw a triangle

Shader program

The code below create a program object:

```
unsigned int shaderProgram = glCreateProgram();
```

The `glCreateProgram` function creates a program and returns the ID reference to the newly created program object. Now it is time to attach the previously compiled shaders to the program object and then link them with `glLinkProgram`:

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Draw a triangle

Shader program

To check if linking a shader program failed and retrieve the corresponding log:

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

The result is a program object that can be activated (in the render loop) by calling `glUseProgram` with the newly created program object as its argument:

```
glUseProgram(shaderProgram);
```

Draw a triangle

Shader program

Once linked into the program, the vertex and fragment shaders are no longer needed, so they can be deleted:

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

Right now, the input vertex data are sent to the GPU and the GPU is instructed on how it should process the vertex data within a vertex and fragment shader.

However, 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.

Draw a triangle

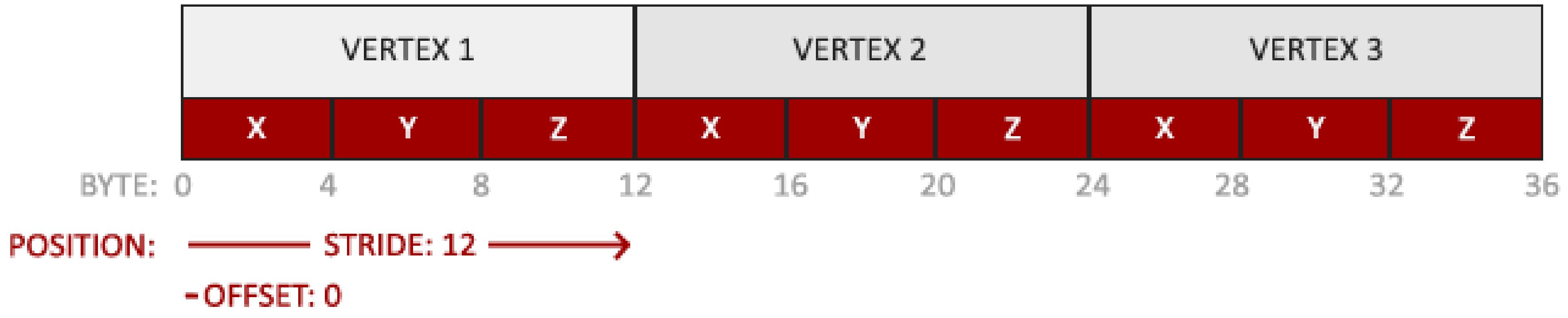
Linking Vertex Attributes

The vertex shader allows developers to specify any input they want in the form of vertex attributes. Although this approach allows for great flexibility, it does mean that it is necessary to manually specify what part of input data goes to which vertex attribute in the vertex shader. In other words, developers need to specify how OpenGL should interpret the vertex data before rendering.

Draw a triangle

Linking Vertex Attributes

The vertex buffer data is formatted as follows:



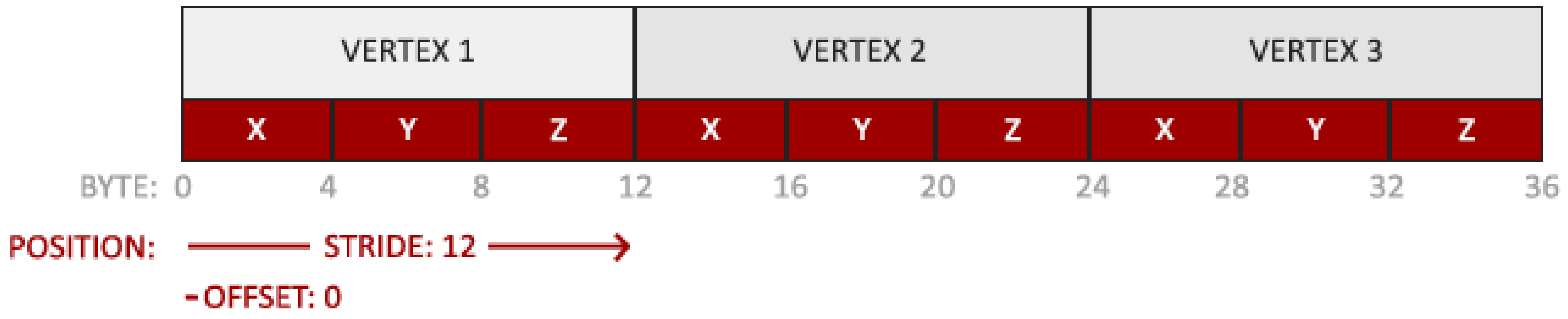
- The position data is stored as 32-bit (4 byte) floating point values.
- Each position is composed of 3 of those values.
- There is no space (or other values) between each set of 3 values. The values are tightly packed in the array
- The first value in the data is at the beginning of the buffer.

Draw a triangle

Linking Vertex Attributes

Knowing this structure, it is possible to tell OpenGL how it should interpret the vertex data (per vertex attribute) using `glVertexAttribPointer`:

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



glVertexAttribPointer()

This function specify how OpenGL should interpret the vertex buffer data whenever a drawing call is made. define an array of generic vertex attribute data

Usage: `void glVertexAttribPointer(Gluint index, Glint size, Glenum type,Glboolean normalized, Glsizei stride, const GLvoid * pointer);`

Parameters:

- `index`: Specifies the index of the generic vertex attribute to be modified.
- `size`: Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. The initial value is 4.

glVertexAttribPointer(): https://docs.gl/gl3/glVertexAttribPointer

glVertexAttribPointer()

Parameters:

- `type`: Specifies the data type of each component in the array. The symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, and `GL_UNSIGNED_INT`. The initial value is `GL_FLOAT`.
- `normalized`: Specifies whether fixed-point data values should be normalized (`GL_TRUE`) or converted directly as fixed-point values (`GL_FALSE`) when they are accessed.
- `stride`: Specifies the byte offset between consecutive generic vertex attributes. If stride is 0, the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.
- `pointer`: Specifies an offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the `GL_ARRAY_BUFFER` target. The initial value is 0.

glVertexAttribPointer(): https://docs.gl/gl3/glVertexAttribPointer

Draw a triangle

Linking Vertex Attributes

After instructing OpenGL on how to interpret the vertex data, the vertex attributes have to be enabled with `glEnableVertexAttribArray` giving the vertex attribute location as its argument (by default, vertex attributes are disabled).

Draw a triangle

Linking Vertex Attributes

In summary, the vertex data are initialized into a buffer using a vertex buffer object, a vertex and fragment shader are set up and OpenGL is instructed on 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 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. set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0); glEnableVertexAttribArray(0);
// 2. use shader program to render an object
glUseProgram(shaderProgram);
// 3. draw the object
OpenGLDrawingFunction();
```

Draw a triangle

Linking Vertex Attributes

The above process should be repeated every time an object has to be draw. Binding the appropriate buffer objects and configuring all vertex attributes for each of those objects quickly becomes a cumbersome process. For this reason, there is a way to store all these state configurations into an object and simply bind this object to restore its state.

Draw a triangle

Vertex Array Object (VAO)

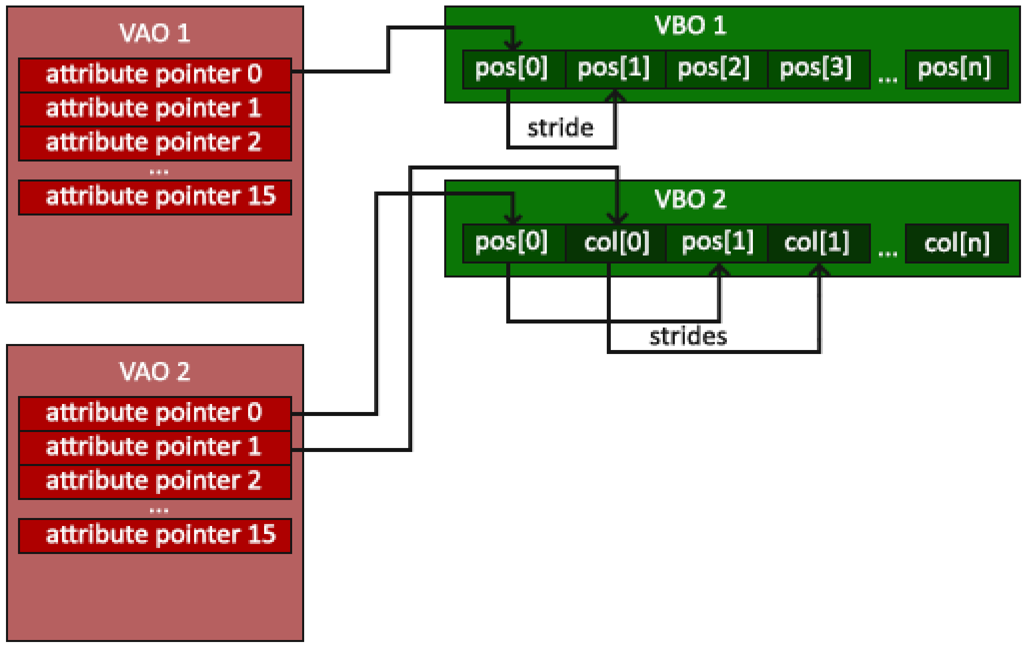
A **Vertex Array Object** (also known as VAO) can be bound just like a vertex buffer object and any subsequent vertex attribute calls from that point on will be stored inside the VAO. This has the advantage that when configuring vertex attribute pointers developer only has to make those calls once and whenever the object has to be draw. This makes switching between different vertex data and attribute configurations as easy as binding a different VAO. All the state is stored inside the VAO.

Draw a triangle

Vertex Array Object (VAO)

A vertex array object stores the following:

- Calls to `glEnableVertexAttribArray` or `glDisableVertexAttribArray`.
- Vertex attribute configurations via `glVertexAttribPointer`.
- Vertex buffer objects associated with vertex attributes by calls to `glVertexAttribPointer`.



Draw a triangle

Linking Vertex Attributes

The process to generate a VAO looks similar to that of a VBO:

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

To use a VAO it is necessary to bind it using `glBindVertexArray`. From that point on developers should bind/configure the corresponding VBO(s) and attribute pointer(s) and then unbind the VAO for later use. As soon as an object has to be drawn, the VAO is bound with the preferred settings.

Draw a triangle

Linking Vertex Attributes

The code of the above process looks like:

```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 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);

[...]

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

Draw a triangle

Linking Vertex Attributes

In summary, a VAO is created that stores vertex attribute configuration and which VBO to use. Usually, when multiple objects have to be drawn, first it is necessary to generate/configure all the VAOs (and thus the required VBO and attribute pointers) and store those for later use. When one of the objects has to be drawn, the corresponding VAO is taken, bonded, the object drawn and the VAO unbound again.

Draw a triangle

OpenGL drawing function

To draw the objects, OpenGL provides developers with the `glDrawArrays` function that draws primitives using the currently active shader, the previously defined vertex attribute configuration and with the VBO's vertex data (indirectly bound via the VAO).

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

glVertexAttribPointer()

This function renders primitives form array data

Usage: `void glDrawArrays( Glenum mode, Glint first, Glsizei count);`

Parameters:

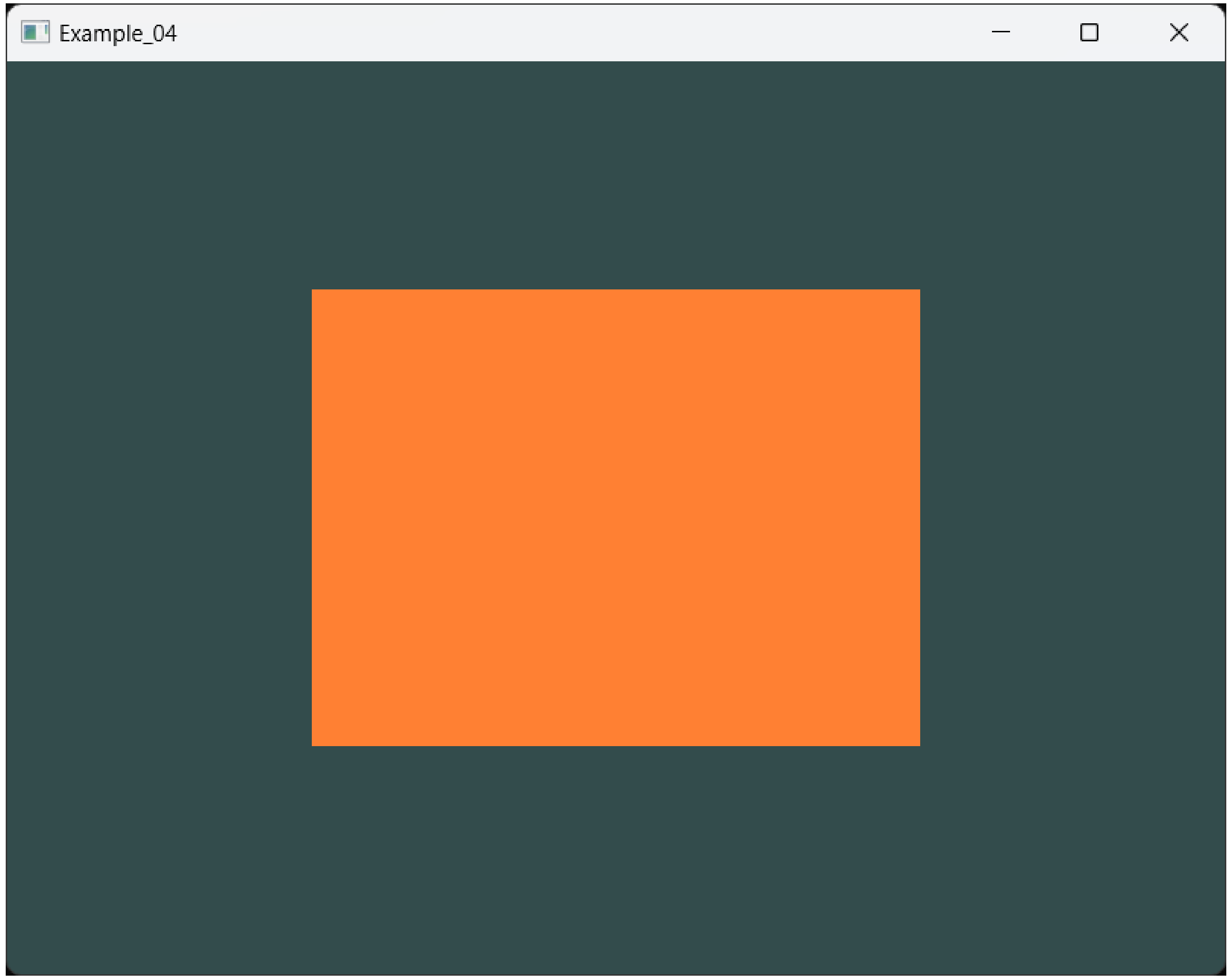
- `mode`: Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_LINE_STRIP_ADJACENCY`, `GL_LINES_ADJACENCY`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP_ADJACENCY` and `GL_TRIANGLES_ADJACENCY` are accepted.
- `first`: Specifies the starting index in the enabled arrays.
- `count`: Specifies the number of indices to be rendered.

glVertexAttribPointer(): https://docs.gl/gl3/glVertexAttribPointer

Draw a rectangle

Example4.cpp

Before drawing it is necessary to introduce the Element Buffer object



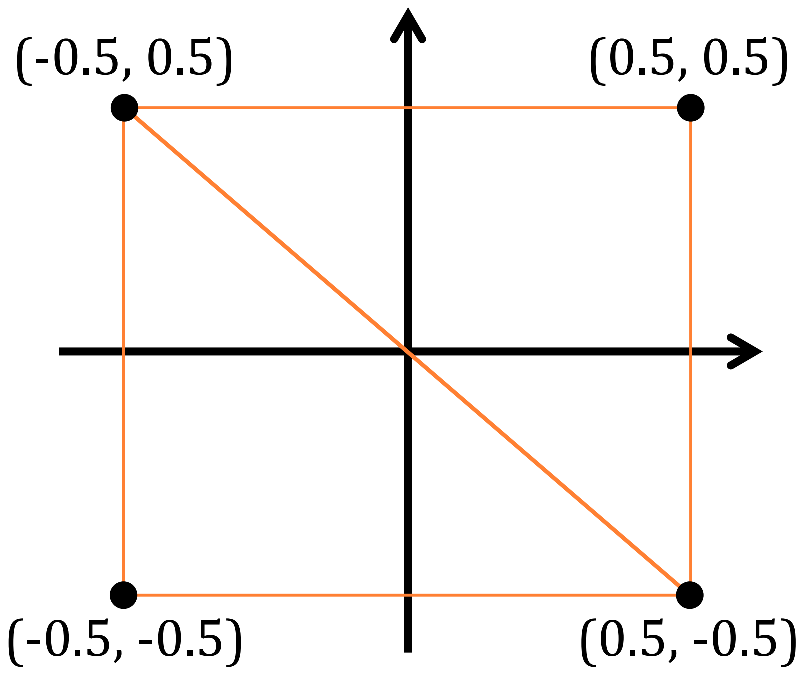
Draw a rectangle

Element Buffer Objects

A rectangle can be drawn using two triangles (OpenGL mainly works with triangles).

The code below will generate the required 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
};
```

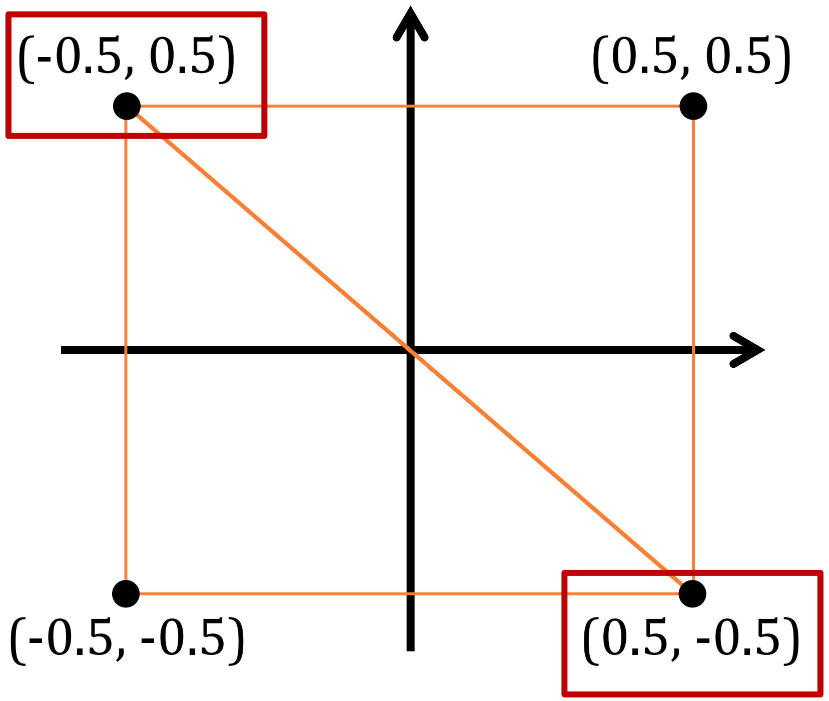


Draw a rectangle

Element Buffer Objects

There is some overlap on the vertices specified. The same rectangle could also be specified with only 4 vertices, instead of 6 (overhead of 50%).

```
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
};
```



Draw a rectangle

Element Buffer Objects

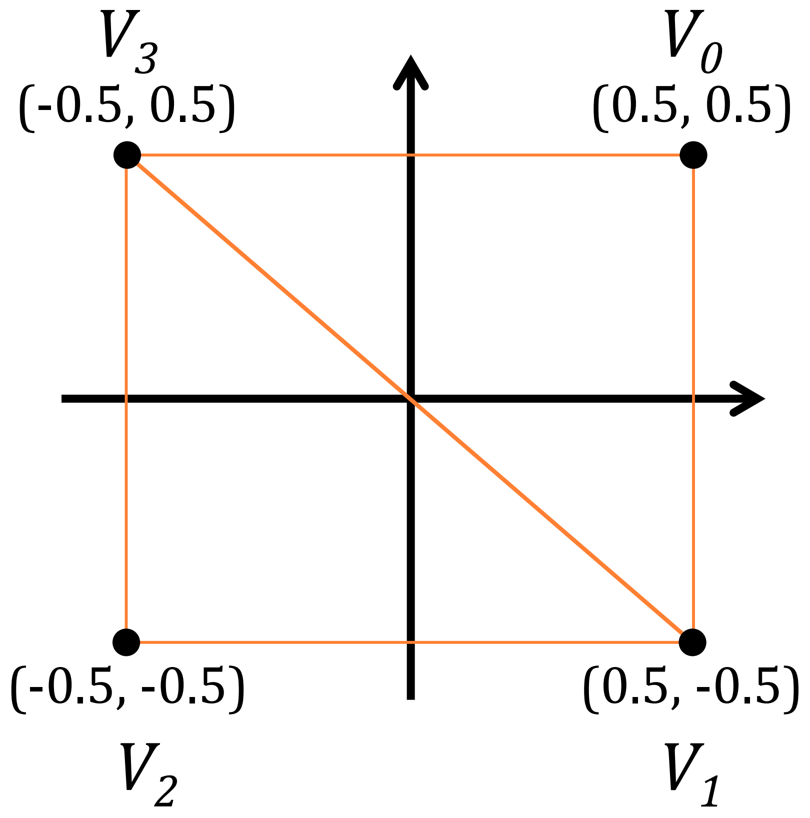
A better solution is to store only the unique vertices and then specify the order in which the vertices have to be drawn. The **Element Buffer Objects** (EBO) provide exactly this feature. An EBO is a buffer, just like a vertex buffer object, that stores indices that OpenGL uses to decide what vertices to draw. This so-called **indexed drawing** is exactly the solution to the problem.

Draw a rectangle

Element Buffer Objects

To get started, first the (unique) vertices and the indices to be drawn as a rectangle have to be specified:

```
// set up vertex data (and buffer(s)) and
configure vertex attributes
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[] = {
    0, 1, 3,  // first Triangle
    1, 2, 3   // second Triangle
};
```



Draw a rectangle

Element Buffer Objects

Similar to the VBO buffer, it is necessary to bind the EBO and copy the indices into the buffer with the `glBufferData`. As previously, the call for copying data in the buffer should be done between a bind and an unbind call. However, this time the bind is made with the `GL_ELEMENT_ARRAY_BUFFER` as the buffer type.

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Draw a rectangle

Element Buffer Objects

The last thing left to do is replacing the `glDrawArrays` call with `glDrawElements` (in the render loop) to indicate that the triangles have to be rendered from an index buffer. When using `glDrawElements` the indices provided in the element buffer object currently bound will be used for drawing:

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

glDrawElements()

This function renders primitives form array data

Usage: `void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);`

Parameters:

- `mode`: Specifies what kind of primitives to render. Symbolic constants `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, and `GL_TRIANGLES` are accepted.
- `count`: Specifies the number of indices to be rendered.
- `type`: Specifies the type of the values in indices. Must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`.
- `indices`: Specifies a byte offset (cast to a pointer type) into the buffer bound to `GL_ELEMENT_ARRAY_BUFFER` to start reading indices from. If no buffer is bound, specifies a pointer to the location where the indices are stored.

glDrawElements(): https://docs.gl/es3/glDrawElements

Draw a rectangle

Element Buffer Objects

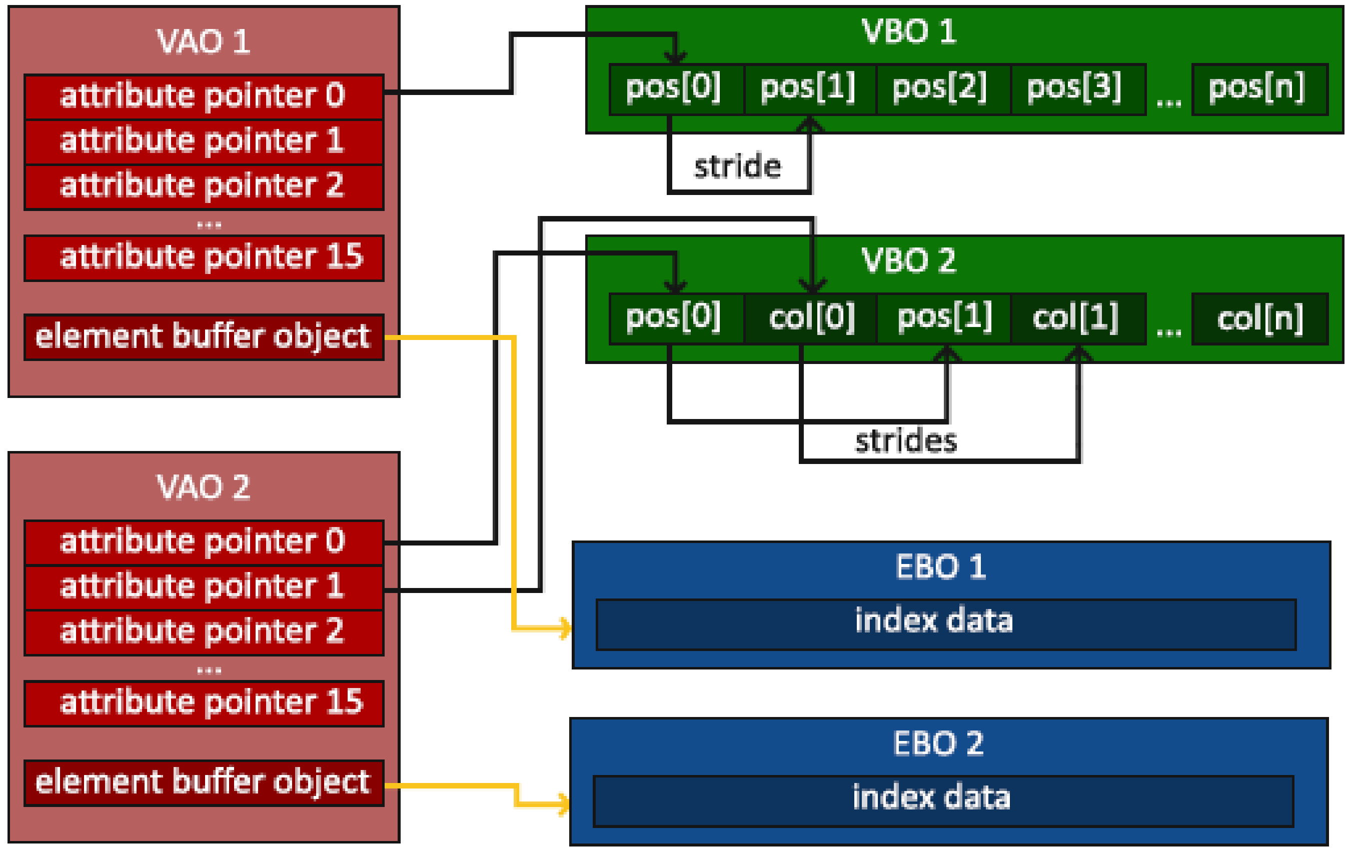
The `glDrawElements` function takes its indices from the EBO currently bound to the `GL_ELEMENT_ARRAY_BUFFER` target. This means that a bind to the corresponding EBO should be done each time an object has to be rendered with indices. This can be a bit cumbersome.

To cope with this issue, a vertex array object also keeps track of element buffer object bindings. The last element buffer object that gets bound while a VAO is bound, is stored as the VAO's element buffer object. Binding to a VAO then also automatically binds that EBO.

A VAO stores the `glBindBuffer` calls when the target is `GL_ELEMENT_ARRAY_BUFFER`. This also means it stores its unbind calls so make sure to not unbind the element array buffer before unbinding VAO, otherwise it doesn't have an EBO configured.

Draw a rectangle

Element Buffer Objects



Draw a rectangle

Element Buffer Objects

The resulting initialization code now looks something like this:

```
// 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);
```

Draw a rectangle

Element Buffer Objects

The resulting code for drawing now looks something like this:

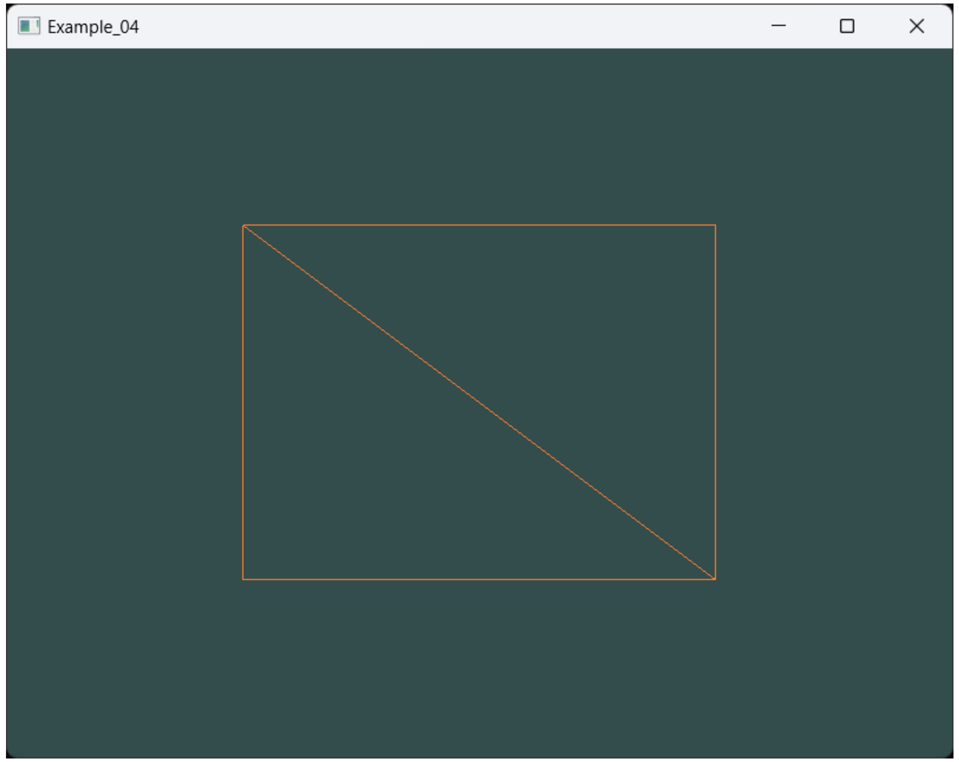
```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
// glBindVertexArray(0); // no need to unbind it every time
```

Draw a rectangle

Activate the Wireframe mode

To draw the triangles in wireframe mode, it is possible to configure how OpenGL draws its primitives via `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. The first argument says that the configuring draw mode has to be applied to the front and back of all triangles. The second parameters specifies that triangles have to be drawn as lines.

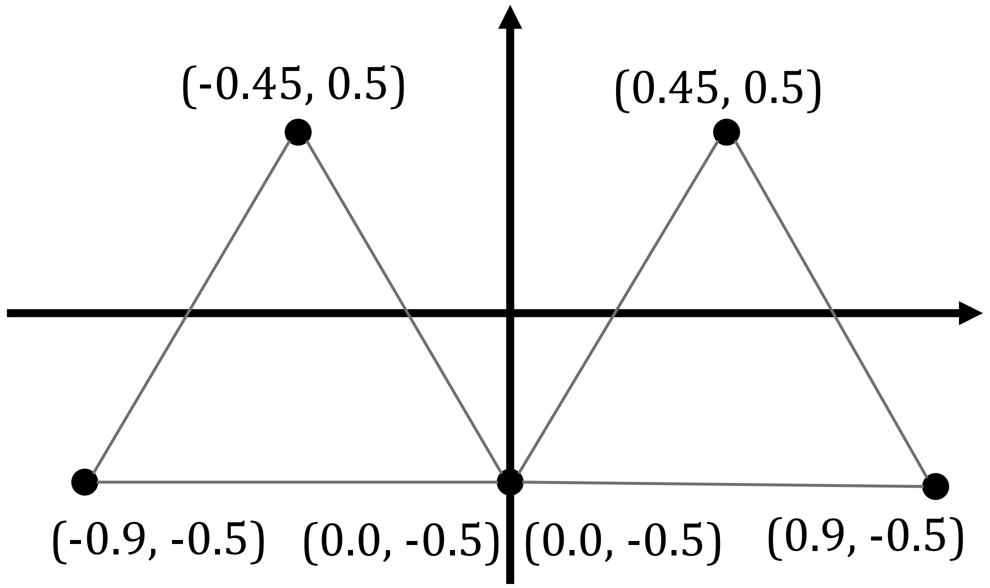
Any subsequent drawing calls will render the triangles in wireframe mode until another draw mode is set, e.g., by setting it to its default using `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)`.



Exercise #2

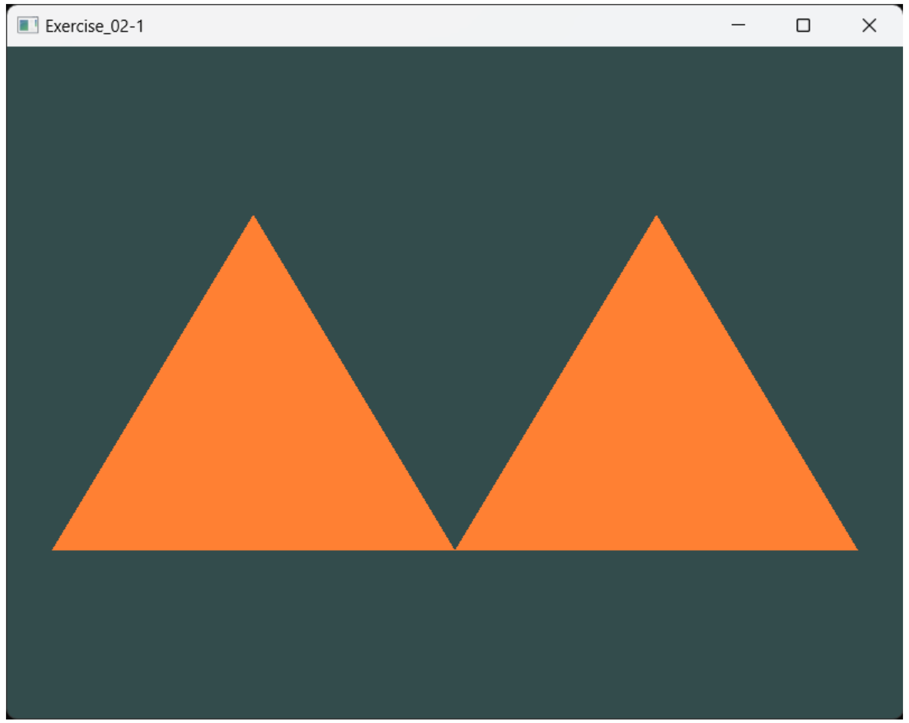
- 2.1) Draw 2 triangles next to each other using `glDrawArrays` by adding more vertices to your data.
  - Start from Example#3 and extents it to complete the exercise
  - Consider the coordinates reported in the figure
- 2.2) Create the same 2 triangles using two different VAOs and VBOs for their data
  - Multiple VAOs and VBOs can be created with

```
unsigned int VBOs[2], VAOs[2];
glGenVertexArrays(2, VAOs);
glGenBuffers(2, VBOs);
```

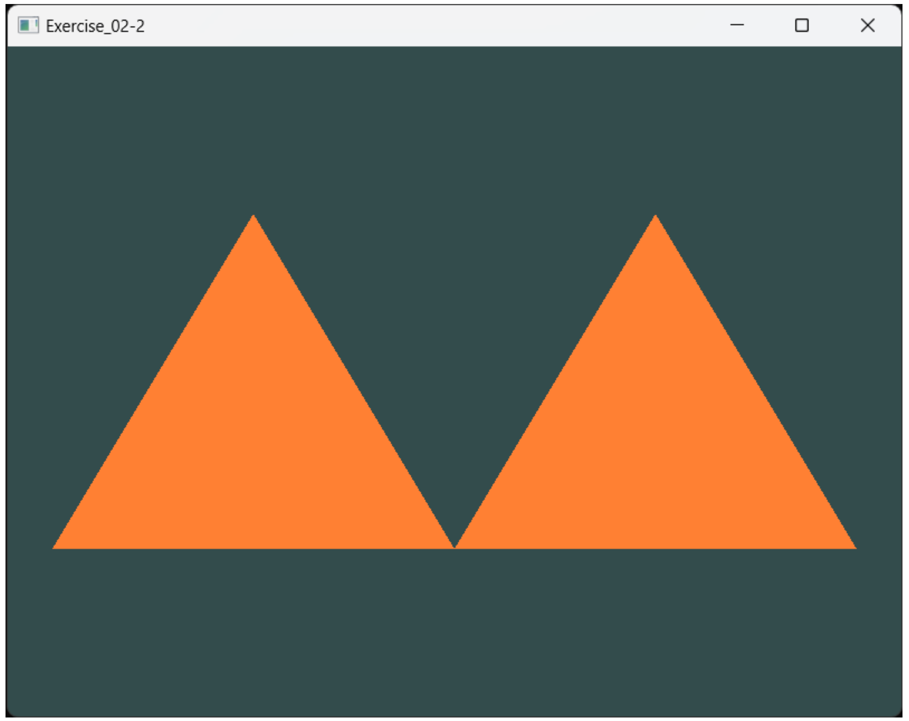


Exercise #2

- Exercise 2.1

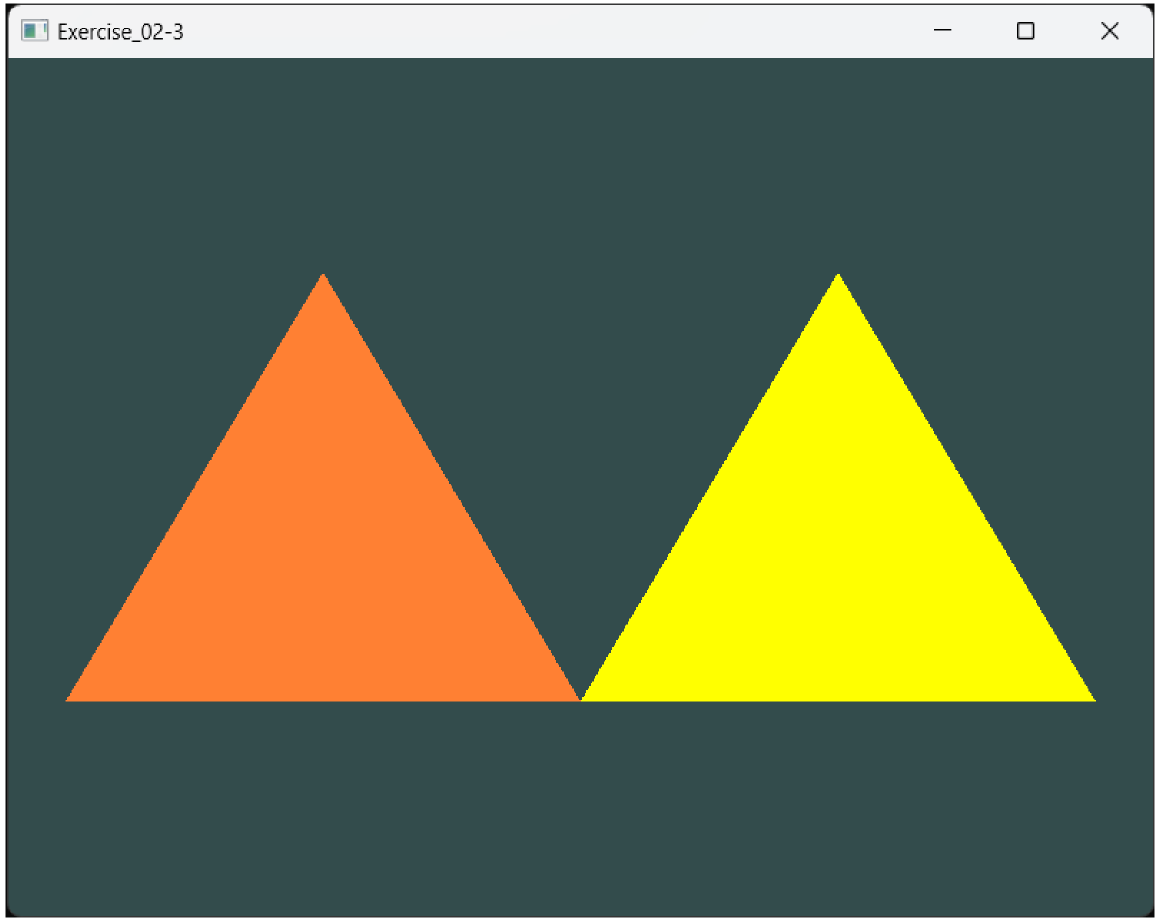


- Exercise 2.2



Exercise #2

- 2.3) Create two shader programs where the second program uses a different fragment shader that outputs the color yellow; draw both triangles again where one outputs the color yellow.


Exercise_02-3