CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

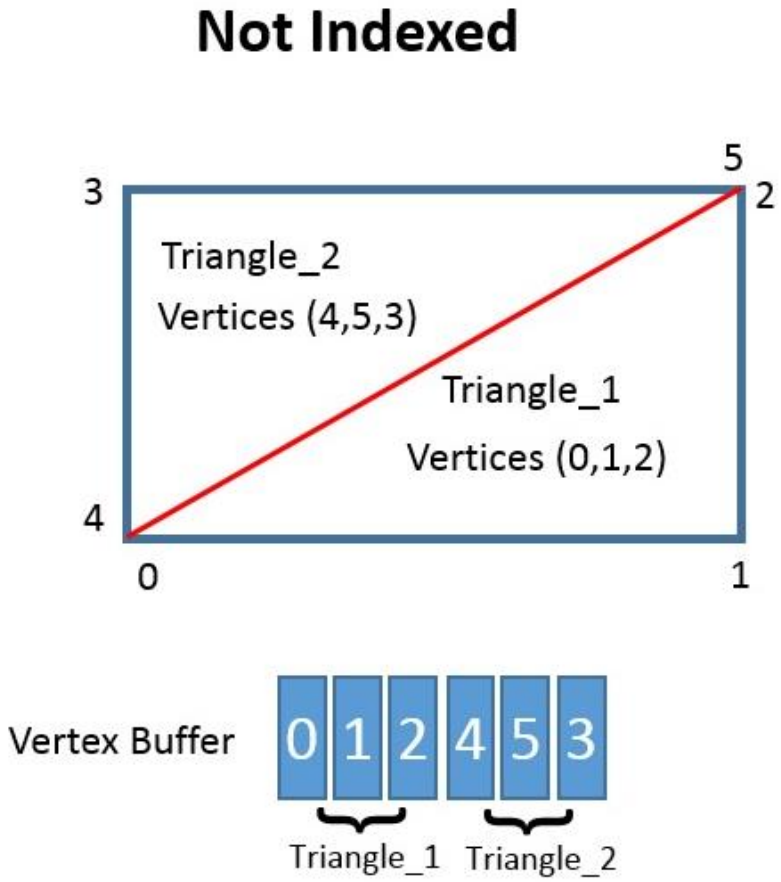
2.2 Transformaciones Geométricas en 2D

2.2.1 OpenGL 2D Programming

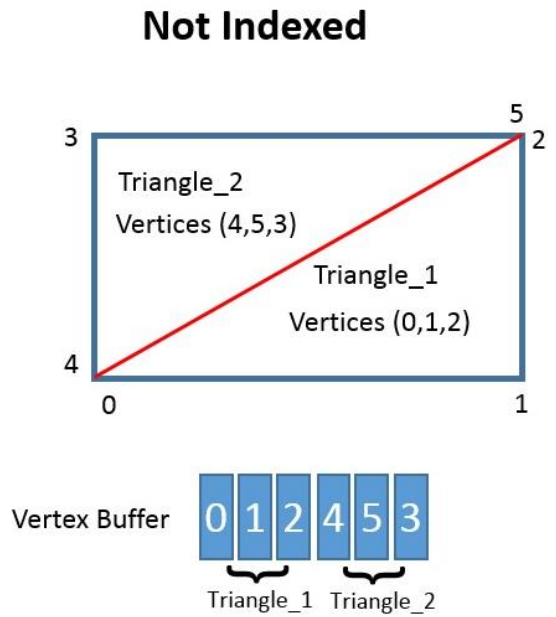
Element Buffer Objects

Suppose we want to draw a rectangle instead of a triangle.
We can draw a rectangle using two triangles (OpenGL mainly works with triangles).

```
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



As you can see, there is some overlap on the vertices specified. We specify bottom right and top left twice!

This is an overhead of 50% since the same rectangle could also be specified with only 4 vertices, instead of 6.

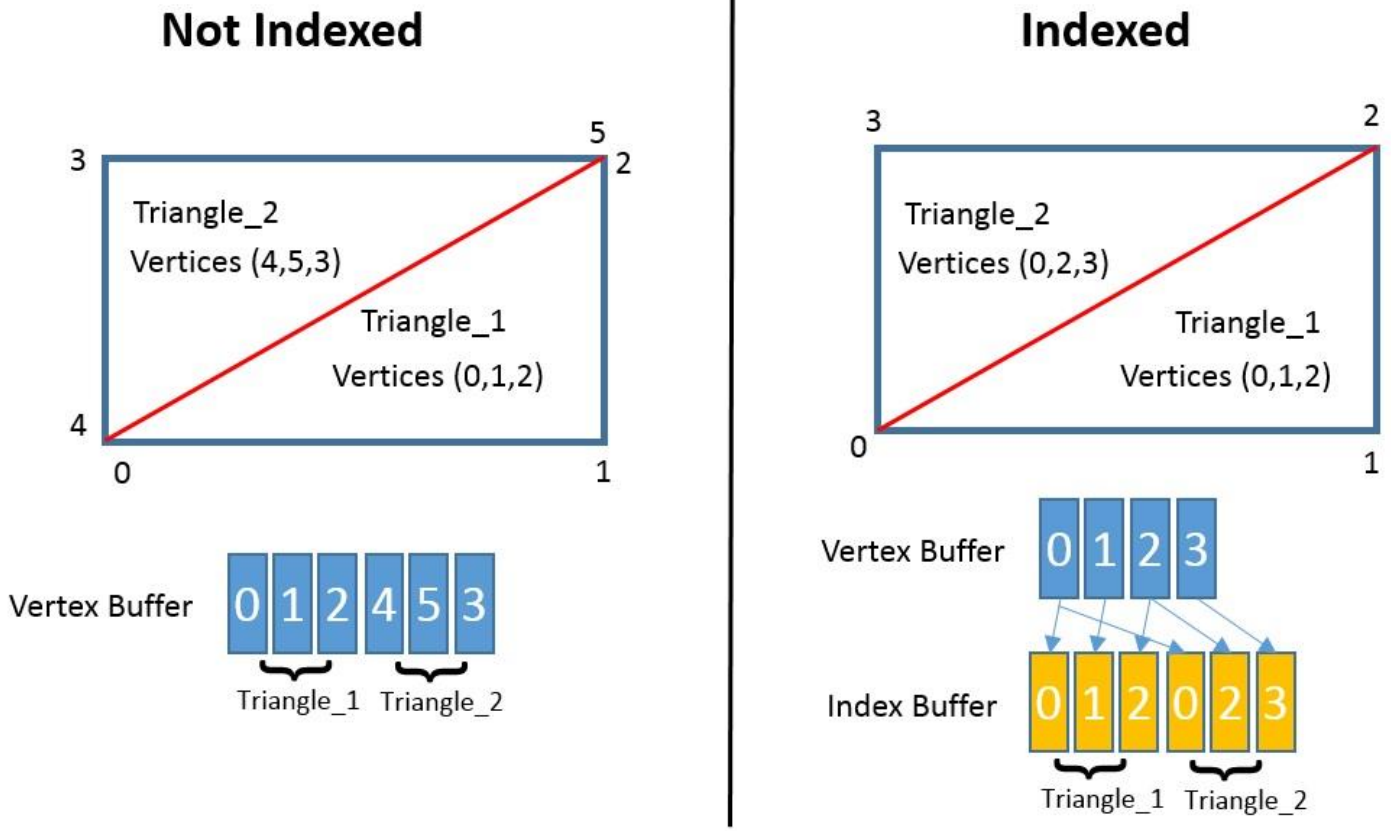
This will only get worse as soon as we have more complex models that have over 1000s of triangles where there will be large chunks that overlap.

```
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

A better solution is to store only the unique vertices and then specify the order at which we want to draw these vertices in.

In that case we would only have to store 4 vertices for the rectangle, and then just specify at which order we'd like to draw them.



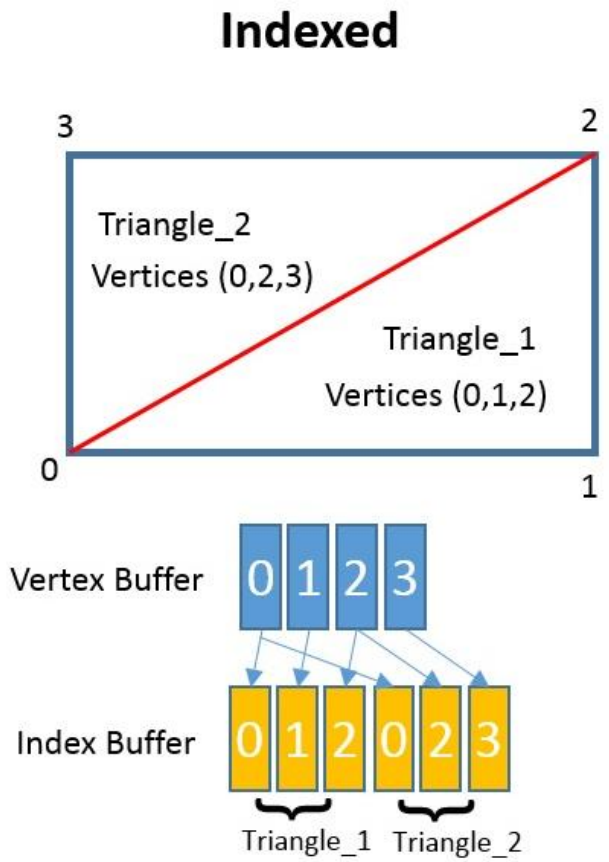
Element Buffer Objects

An Element Buffer Object EBO is a buffer, just like a vertex buffer object, that stores indices that OpenGL uses to decide what vertices to draw.

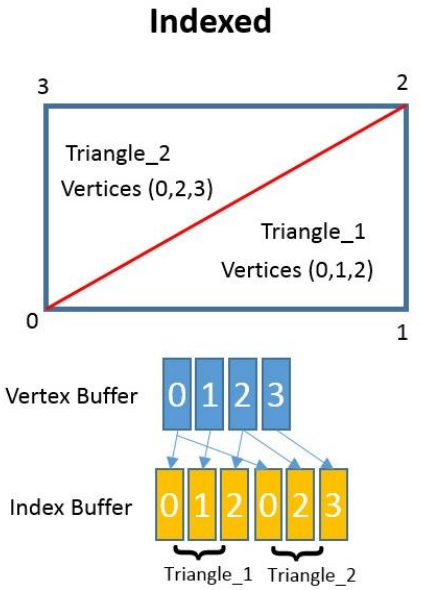
This so called indexed drawing is exactly the solution to our problem.

We first have to specify the (unique) vertices and the indices to draw them as a 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



Process:

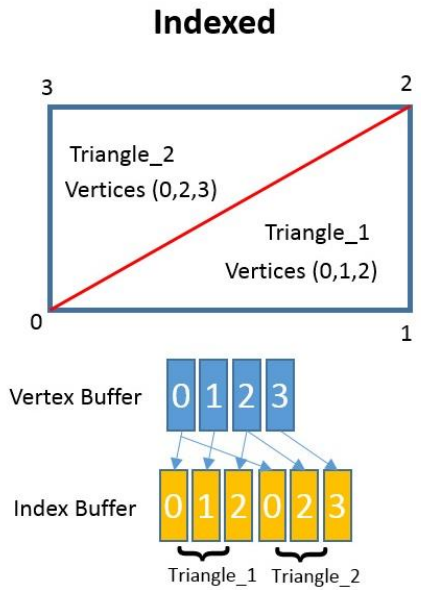
- Create the element buffer object.

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

- Bind the EBO and copy the indices into the buffer with glBufferData.
- Also, just like the VBO we want to place those calls between a bind and an unbind call, although this time we specify 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);
```

Element Buffer Objects



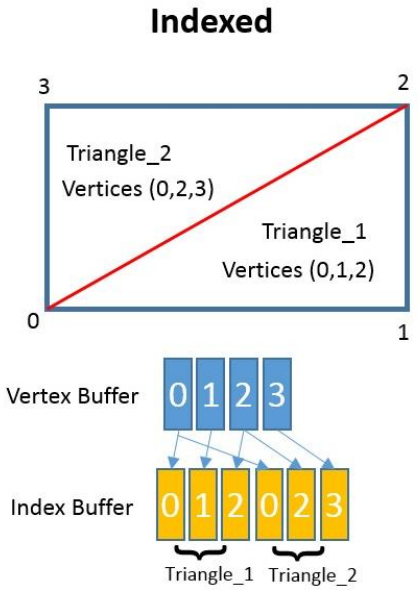
Process:

- The last thing left to do is replace the glDrawArrays call with glDrawElements to indicate we want to render the triangles from an index buffer.

- When using glDrawElements we're going to draw using indices provided in the element buffer object currently bound:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

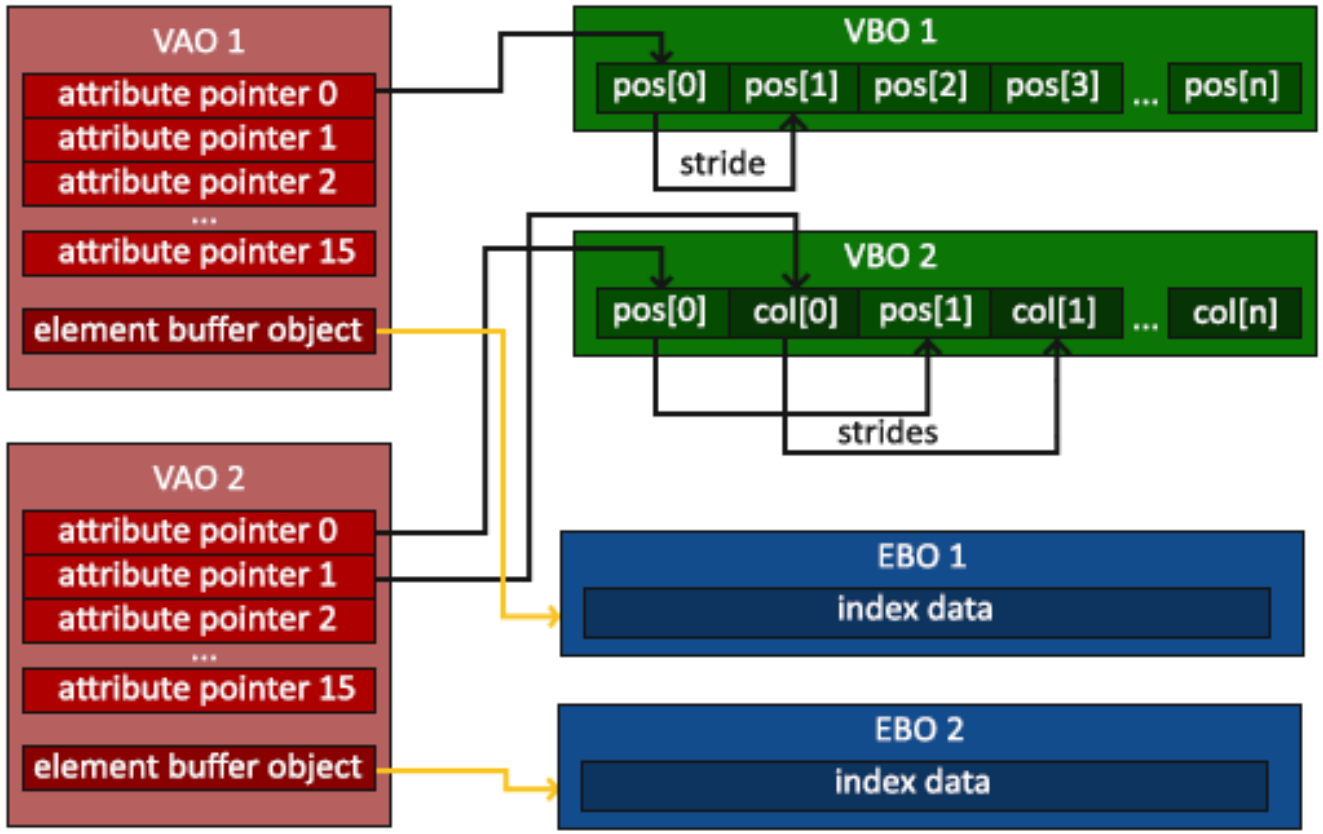
Element Buffer Objects

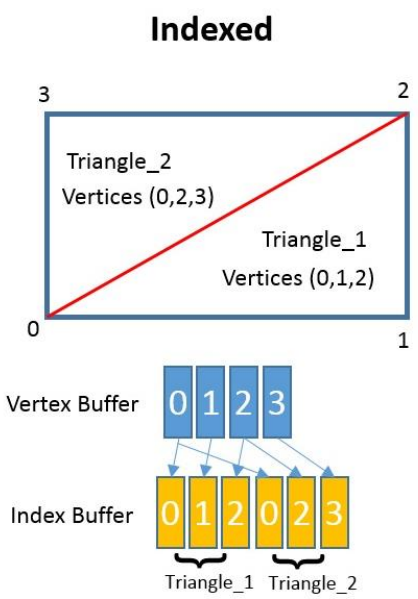



```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

- **The first argument** specifies the mode we want to draw in, similar to glDrawArrays.
- **The second argument** is the count or number of elements we'd like to draw. We specified 6 indices so we want to draw 6 vertices in total.
- **The third argument** is the type of the indices which is of type GL_UNSIGNED_INT.
- **The last argument** allows us to specify an offset in the EBO (or pass in an index array, but that is when you're not using element buffer objects), but we're just going to leave this at 0.

Element Buffer Objects



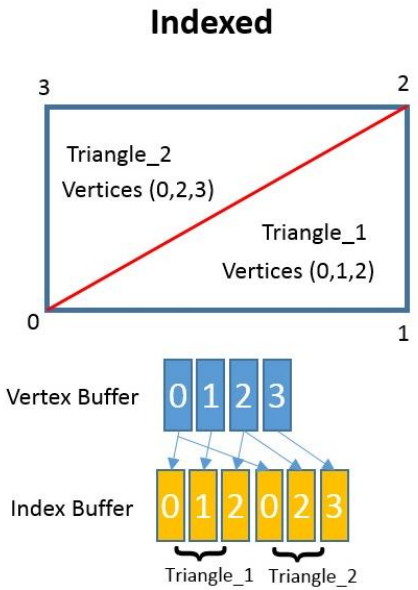


The glDrawElements function takes its indices from the EBO currently bound to the GL_ELEMENT_ARRAY_BUFFER target. This means we have to bind the corresponding EBO each time we want to render an object with indices which again is a bit cumbersome. It just so happens that a vertex array object also keeps 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.

Element Buffer Objects

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 you don't unbind the element array buffer before unbinding your VAO, otherwise it doesn't have an EBO configured.



The resulting initialization and drawing code:

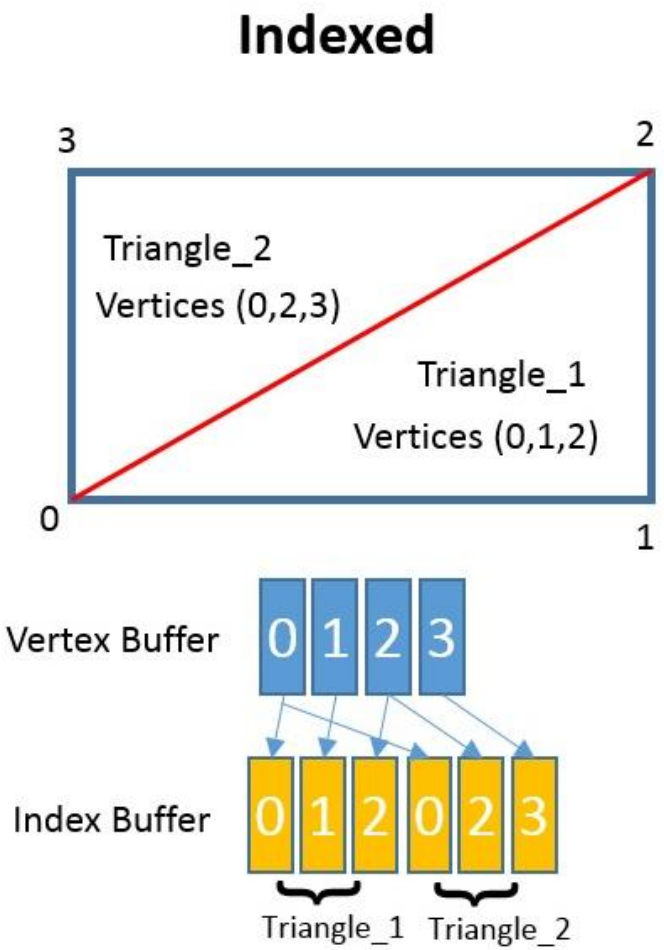
```
// ..:: Initialization code :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in 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);

[...]
```

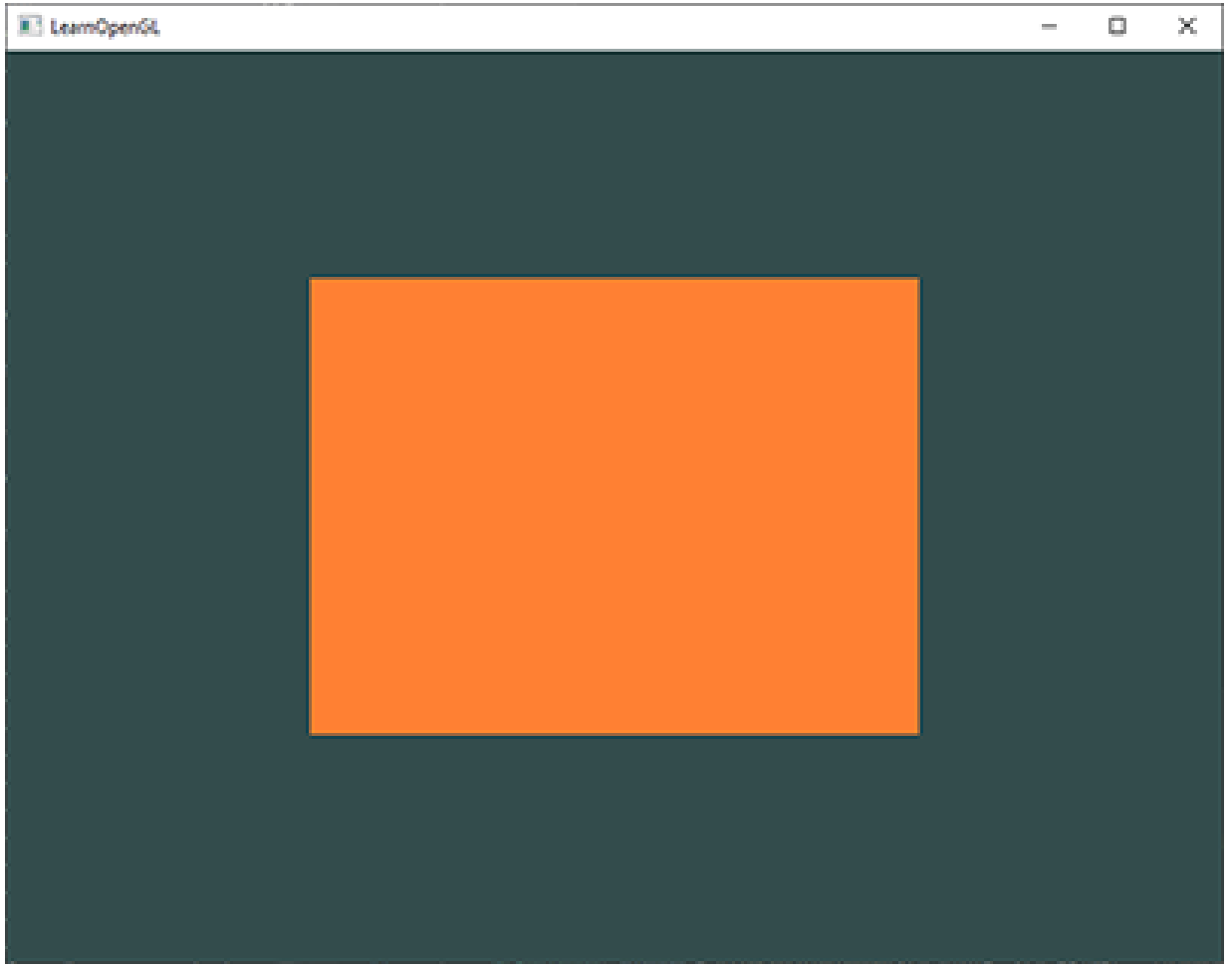
```
[...]

// ..:: Drawing code (in render loop) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
```

glBindVertexArray(0) breaks the existing vertex array object binding

Element Buffer Objects

Running the program:



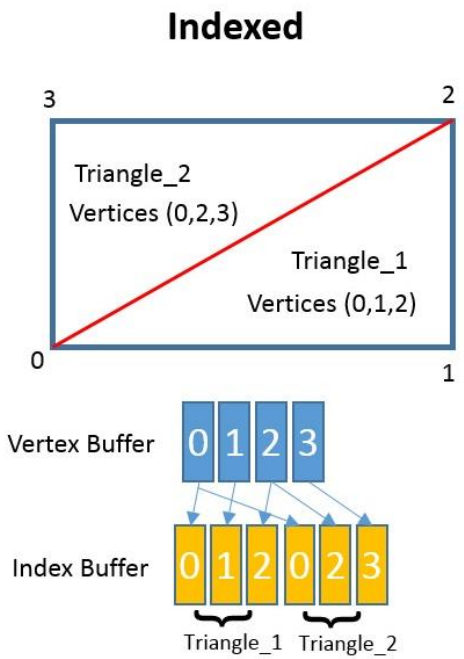


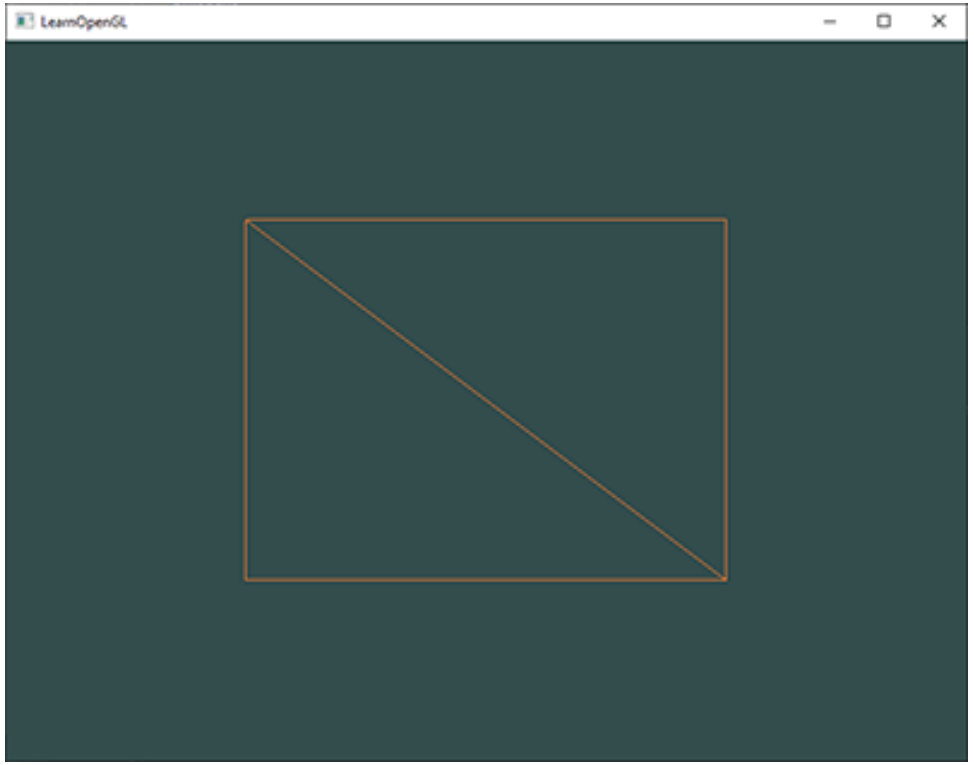
B1C2_2D_Square_EBO.cpp

Element Buffer Objects

Wireframe mode

To draw your triangles in wireframe mode, you can configure how OpenGL draws its primitives via glPolygonMode(GL_FRONT_AND_BACK, GL_LINE). The first argument says we want to apply it to the front and back of all triangles and the second line tells us to draw them as lines. Any subsequent drawing calls will render the triangles in wireframe mode until we set it back to its default using glPolygonMode(GL_FRONT_AND_BACK, GL_FILL).





B1C2_2D_Sqare_EBO_wireframe.cpp

Additional Exercises

Ejercicio 1:

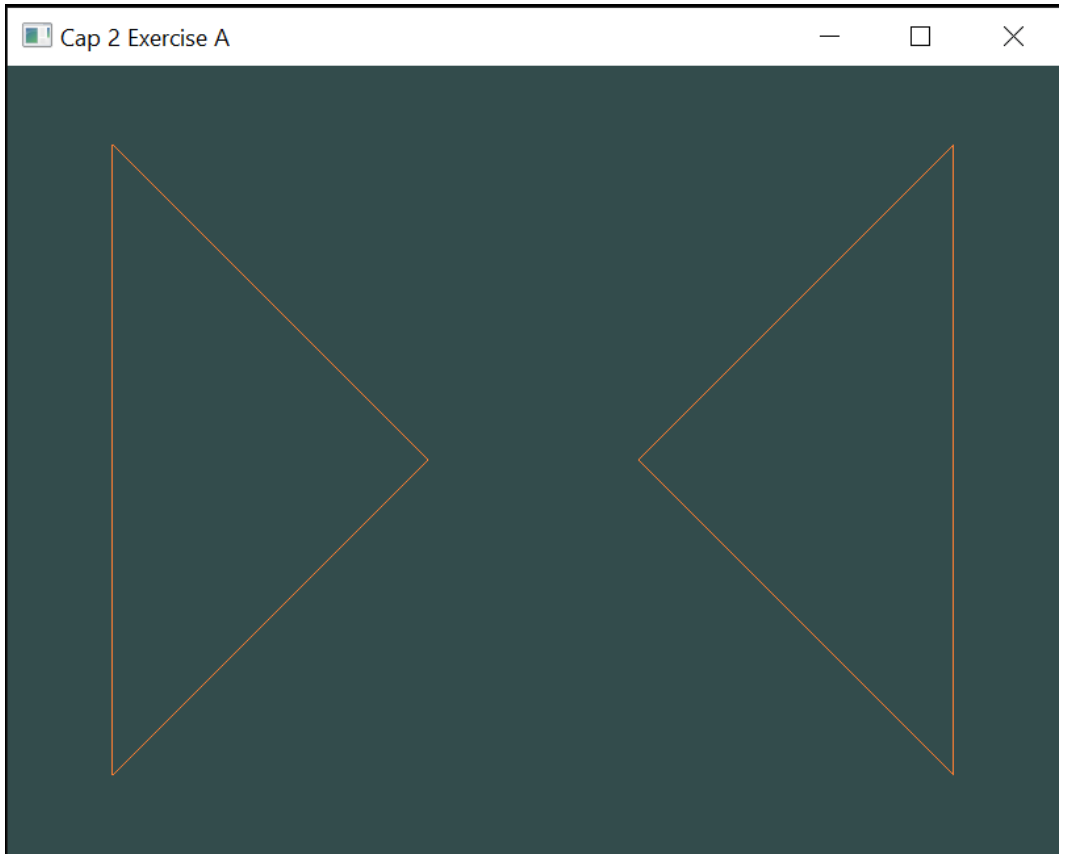
- Try to draw 2 triangles next to each other using glDrawArrays by adding more vertices to your data:

Ejercicio 1:

- Try to draw 2 triangles next to each other using glDrawArrays by adding more vertices to your data:

```
float vertices[] = {
      -0.2f, 0.0f, 0.0f, // left  1
      -0.8f, 0.8f, 0.0f, // right 1
      -0.8f, -0.8f, 0.0f,  // top  1
      0.2f, 0.0f, 0.0f, // left  2
      0.8f, 0.8f, 0.0f, // right 2
      0.8f, -0.8f, 0.0f  // top  2
  };
```

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```



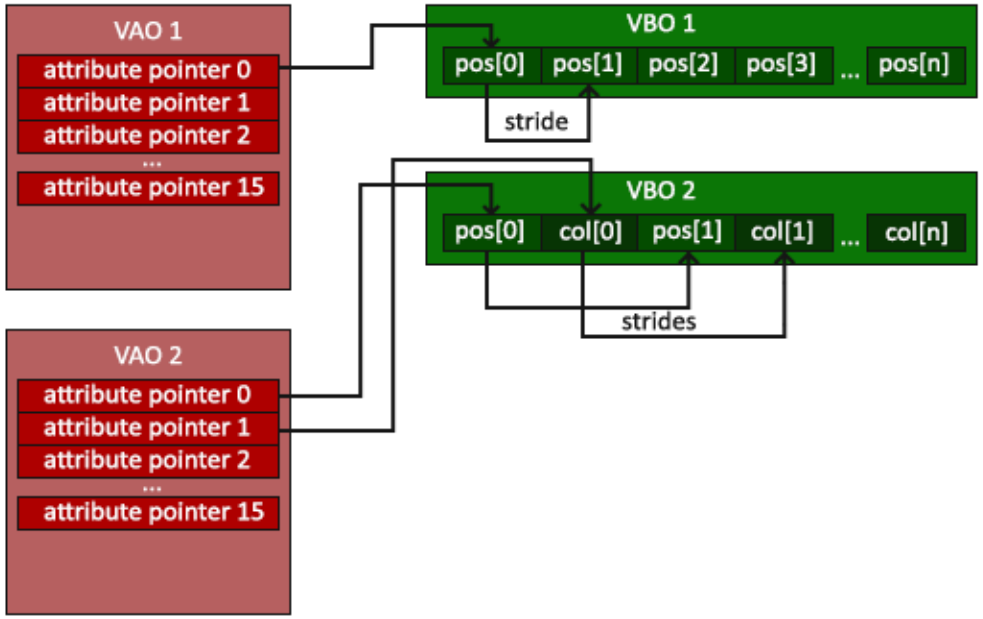
Note: In this case, we do not use EBOs

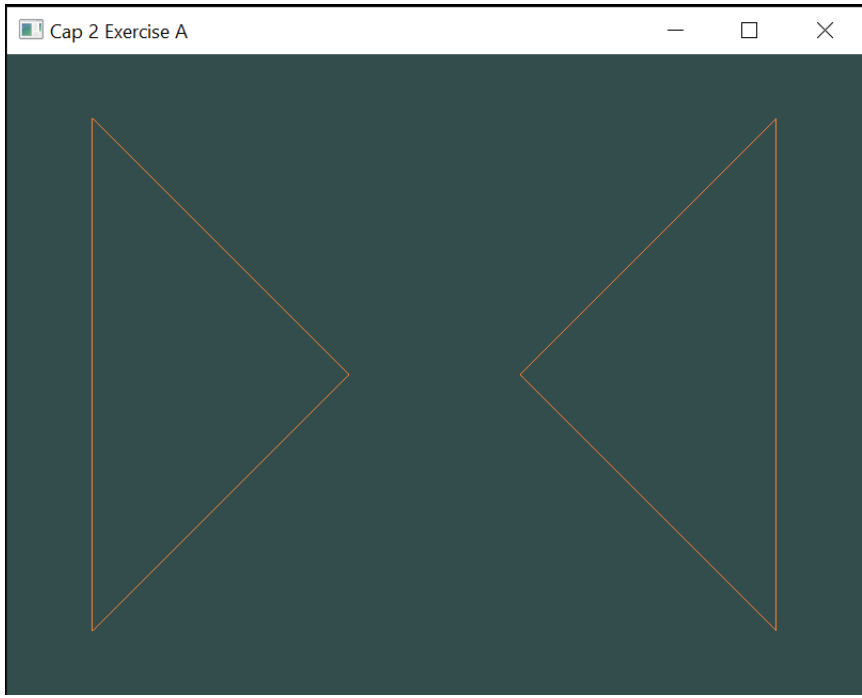
```
// glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
 //glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

Ejercicio 2:

- Create the same 2 triangles using two different VAOs and VBOs for their data:

VERTEX DATA [] →



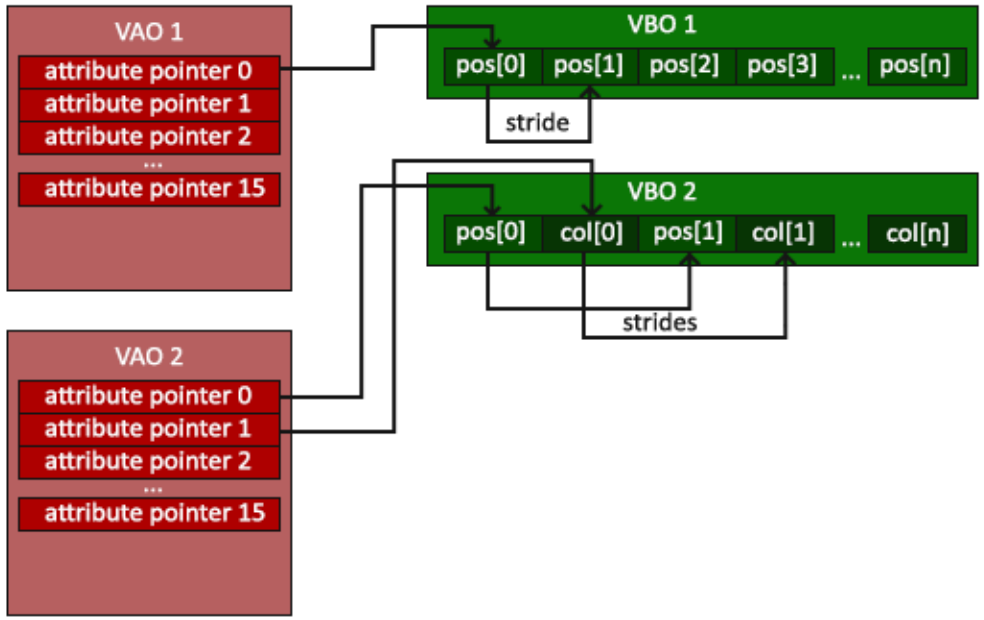


Ejercicio 2:

- Create the same 2 triangles using two different VAOs and VBOs for their data:

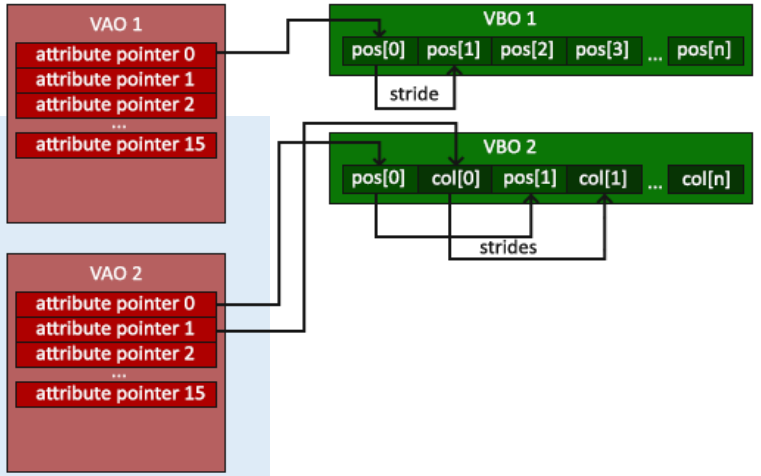
VERTEX DATA [] →

```
float firstTriangle[] = {
    -0.2f, 0.0f, 0.0f, // left  1
     -0.8f, 0.8f, 0.0f, // right 1
     -0.8f, -0.8f, 0.0f,  // top  1
 };

 float secondTriangle[] = {
     0.2f, 0.0f, 0.0f, // left  2
    0.8f, 0.8f, 0.0f, // right 2
    0.8f, -0.8f, 0.0f  // top  2
 };
```



Ejercicio 2:

- Create the same 2 triangles using two different VAOs and VBOs for their data:

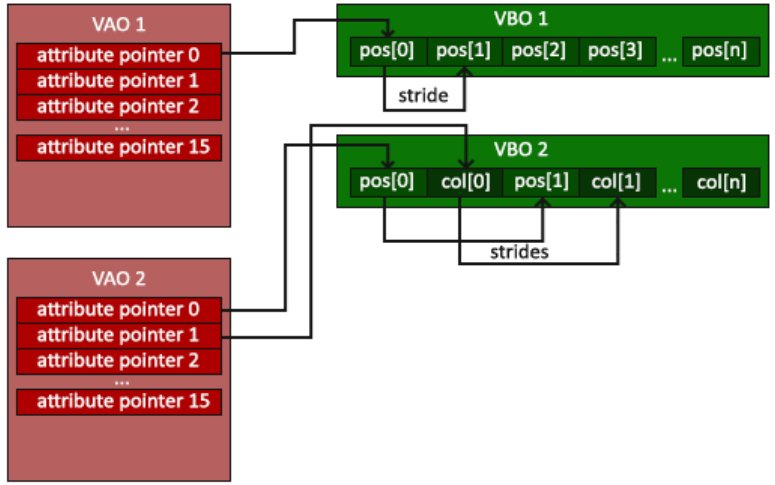


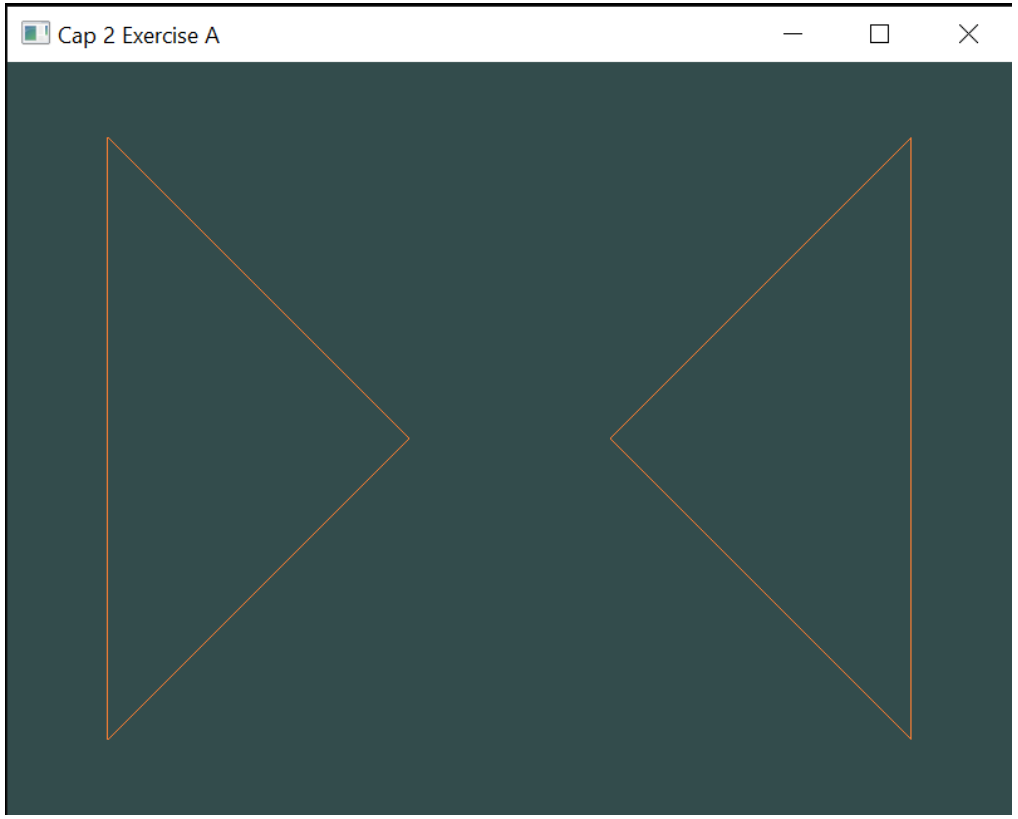

```
unsigned int VBOs[2], VAOs[2];
glGenVertexArrays(2, VAOs); // we can also generate multiple VAOs or buffers at the same time
glGenBuffers(2, VBOs);
// first triangle setup
// --------------------
glBindVertexArray(VAOs[0]);
glBindBuffer(GL_ARRAY_BUFFER, VBOs[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(firstTriangle), firstTriangle, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0); // Vertex attributes stay the same
glEnableVertexAttribArray(0);
// glBindVertexArray(0); // no need to unbind at all as we directly bind a different VAO the next few lines
// second triangle setup
// --------------------
glBindVertexArray(VAOs[1]);     // note that we bind to a different VAO now
glBindBuffer(GL_ARRAY_BUFFER, VBOs[1]);                // and a different VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(secondTriangle), secondTriangle, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0); // because the vertex data is tightly packed we
can also specify 0 as the vertex attribute's stride to let OpenGL figure it out
glEnableVertexAttribArray(0);
```

Ejercicio 2:

- Create the same 2 triangles using two different VAOs and VBOs for their data:

VERTEX DATA [] →

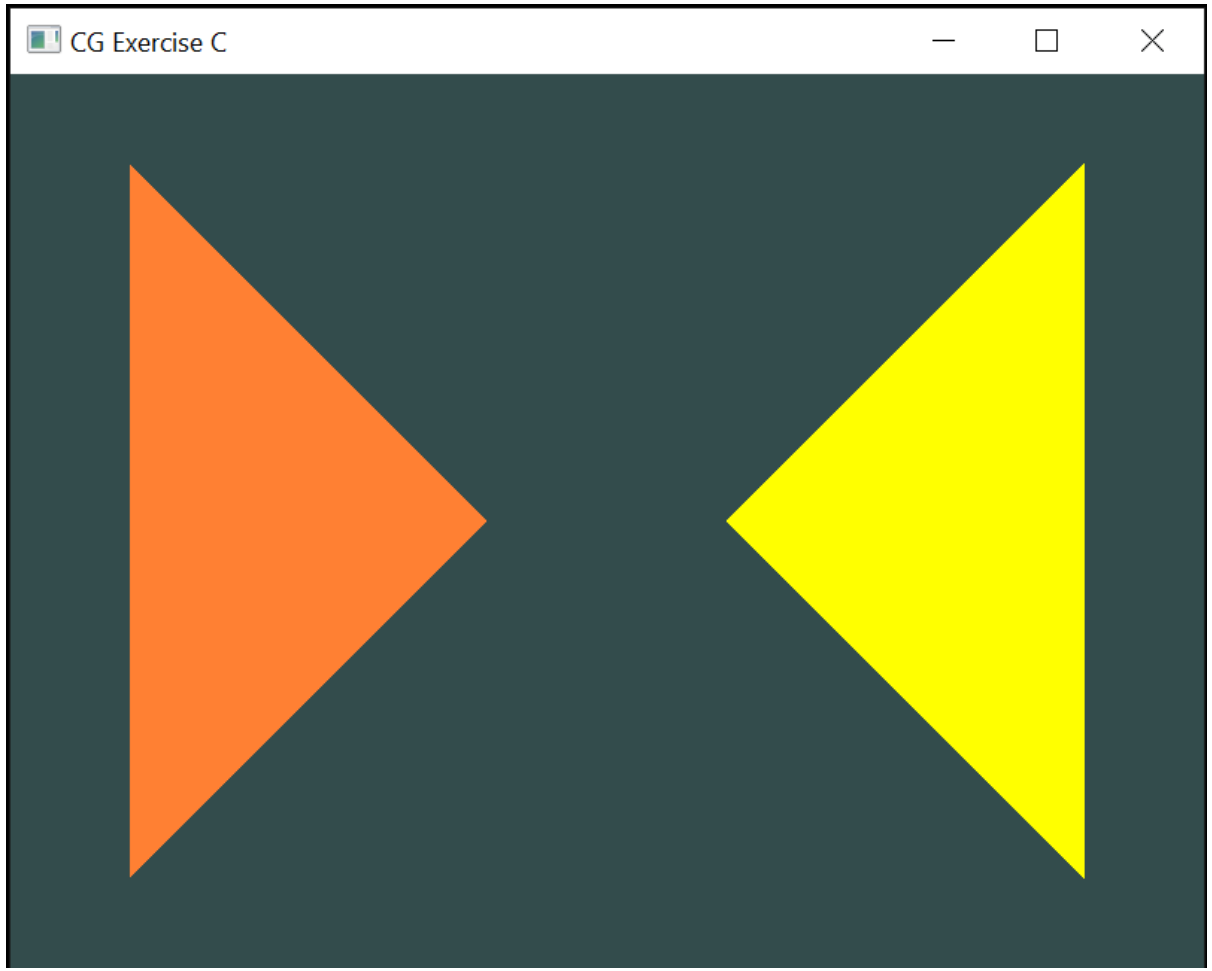





```
// in render loop...
// draw first triangle using the data from the first VAO
glBindVertexArray(VAOs[0]);
glDrawArrays(GL_TRIANGLES, 0, 3);
// then we draw the second triangle using the data from the second VAO
glBindVertexArray(VAOs[1]);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Ejercicio 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:



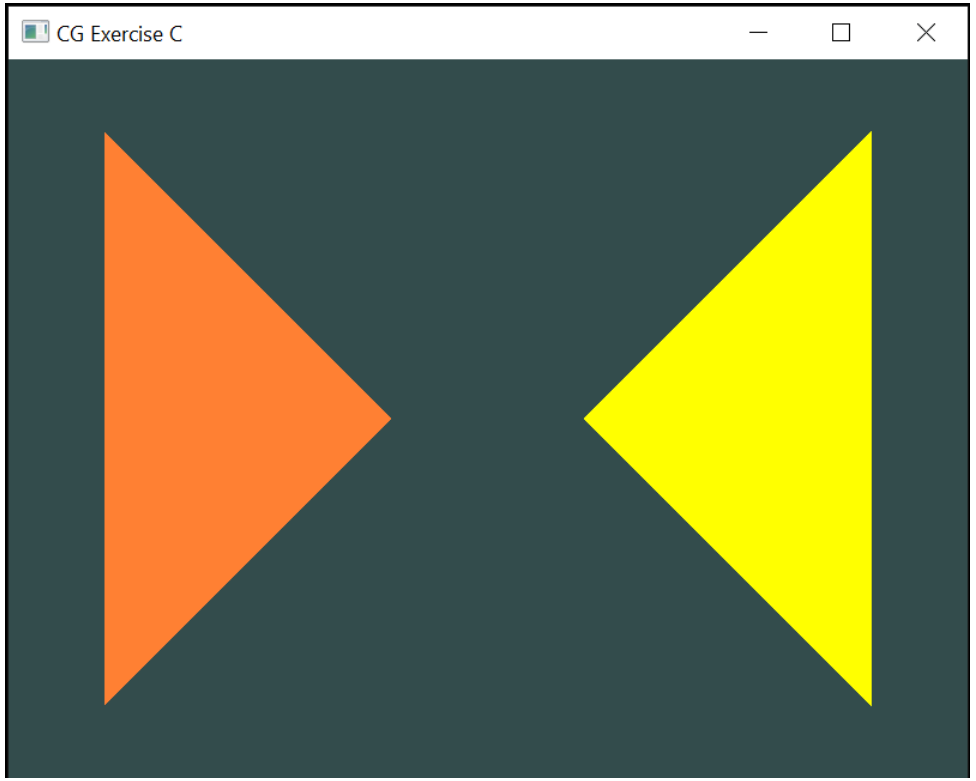
Ejercicio 3:

- 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:

Fragment shader thread 1

Fragment shader thread 2

```
 const char* fragmentShader1Source = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n\0";
const char* fragmentShader2Source = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 1.0f, 0.0f, 1.0f);\n"
"}\n\0";
```



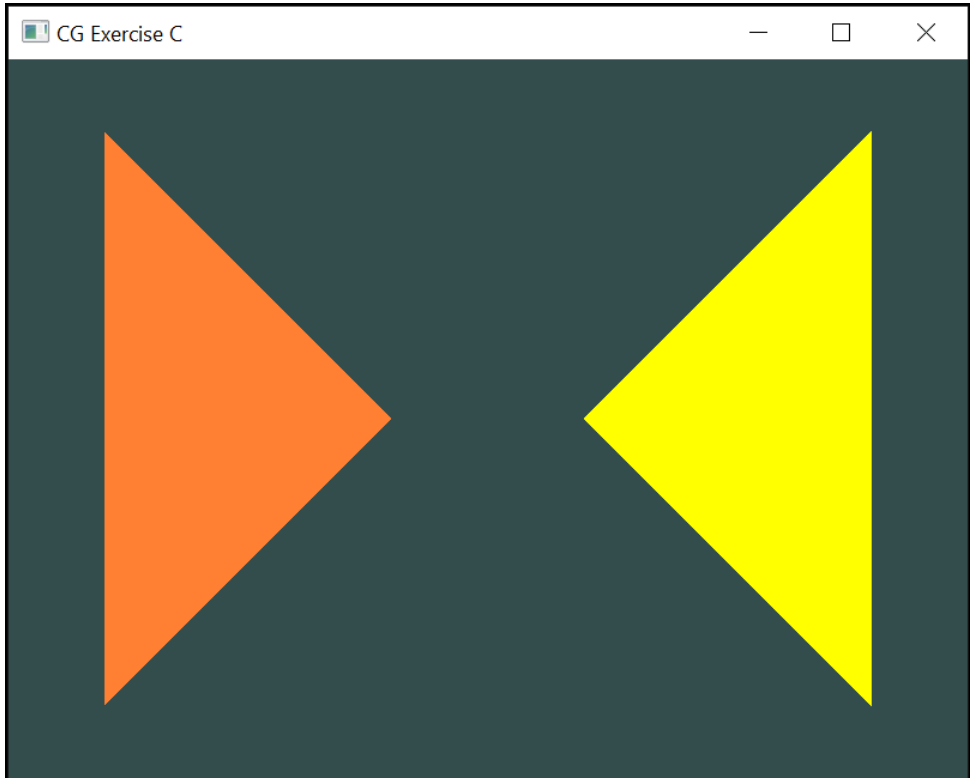
Ejercicio 3:

- 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:

Fragment shader thread 1

Fragment shader thread 2

```
unsigned int fragmentShaderYellow = glCreateShader(GL_FRAGMENT_SHADER);
...
unsigned int shaderProgramYellow = glCreateProgram();
glShaderSource(fragmentShaderYellow, 1, &fragmentShader2Source, NULL);
glCompileShader(fragmentShaderYellow);
...
glAttachShader(shaderProgramYellow, vertexShader);
glAttachShader(shaderProgramYellow, fragmentShaderYellow);
glLinkProgram(shaderProgramYellow);
...
```



Ejercicio 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:

```
 // now when we draw the triangle we first use the vertex and orange fragment shader from the first program
      glUseProgram(shaderProgramOrange);
      // draw the first triangle using the data from our first VAO
      glBindVertexArray(VAOs[0]);
      glDrawArrays(GL_TRIANGLES, 0, 3); //this call should output an orange triangle
      // then we draw the second triangle using the data from the second VAO
      // when we draw the second triangle we want to use a different shader program so we switch to the shader program with our yellow fragment shader.
      glUseProgram(shaderProgramYellow);
      glBindVertexArray(VAOs[1]);
      glDrawArrays(GL_TRIANGLES, 0, 3);// this call should output a yellow triangle
```