```
//declare a vector in OpenGL
glm::vec4 newVector(1.0f, 2.0f, 3.0f, 1.0f);
glm::vec4 newVector2(1.0f, 3.0f, 2.0f, 4.0f);

//you can do operations directly with scalars or between vectors

float length = glm::length(newVector); //length of a vector
float dotProduct = glm::dot(newVector, newVector1);

//same for cross product

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

glm::mat2x3 newMatrix(1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f);

glm::mat4 identity4Matrix(1.0f); //declaration of identity matrix


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
glm::mat4 newMatrix(1.0f);
glm::mat4 newMatrix(2.0f);
glm::mat4 newMatrix3;

newMatrix3 = 2.0f + newMatrix1; //addition or subtraction of a scalar newMatrix3 = 2.0f *
newMatrix1; //multiplication with a scalar newMatrix3 = newMatrix1 + newMatrix2; //matrix
addition or subtraction newMatrix3 = -newMatrix1; //matrix negation
newMatrix3 = newMatrix1 * newMatrix2; //matrix multiplication


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

glm::vec4 newVector1(1.0f, 2.0f, 3.0f, 1.0f);
glm::mat4 newMatrix1(1.0f);
glm::vec4 newVector3;

newVector3 = newMatrix1 * newVector1; //matrix-vector multiplication


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

glm::mat4 oldTransformationMatrix;
glm::mat4 newTransformationMatrix;
newTransformationMatrix = glm::translate(oldTransformationMatrix, glm::vec3(1.0f, 2.0f, 3.0f));
 //generate a translation matrix with (1.0f, 2.0f, 3.0f)

newTransformationMatrix = glm::scale(oldTransformationMatrix, glm::vec3(2.0f, 1.0f, 3.0f));
//generate a scale matrix with (2.0f, 1.0f, 3.0f)

newTransformationMatrix = glm::rotate(oldTransformationMatrix, glm::radians(45.0f),
glm::vec3(0.0f, 1.0f, 0.0f));
//generate a rotation matrix with 45.0 degrees around the (0.0f, 1.0f, 0.0f) axis (Y axis)
```

```
newTransformationMatrix = glm::lookAt(glm::vec3(0.0f, 0.0f, 7.0f), glm::vec3(0.0f, 0.0f, - 10.0f), glm::vec3(0.0f, 1.0f, 0.0f));
 //generate a camera transformation (camera position, reference point, up vector)

newTransformationMatrix = glm::perspective(45.0f, (GLfloat)screen_width / (GLfloat)screen_height, 1.0f, 1000.0f);
//generate a perspective projection matrix (vertical fov, aspect ratio, zNear, zFar)


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
glm::mat4 translationScaleRotation(1.0f); //start with the identity matrix
translationScaleRotation = glm::rotate(translationScaleRotation, glm::radians(45.0f), glm::vec3(0.0f, 1.0f, 0.0f));
//generate a rotation matrix with 45.0 degrees around the (0.0f, 1.0f, 0.0f) axis (Y axis)

translationScaleRotation = glm::scale(translationScaleRotation, glm::vec3(2.0f, 1.0f, 3.0f));
//generate a scale matrix with (2.0f, 1.0f, 3.0f)

translationScaleRotation = glm::translate(translationScaleRotation, glm::vec3(1.0f, 2.0f, 3.0f));
//generate a translation matrix with (1.0f, 2.0f, 3.0f)



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

A Vertex Buffer Object (VBO) is a buffer used to hold an array of vertex attributes (such as position, color, normal vector, etc.).
A Vertex Array Object (VAO) groups together multiple VBOs. In order to use VBOs we need to:
 generate VBO names (IDs): glGenBuffers(…)
 bind (select) a specific VBO for initialization: glBindBuffer(GL_ARRAY_BUFFER, …)
 load data into the VBO: glBufferData(GL_ARRAY_BUFFER, ...)

generate VAO names (IDs): glGenVertexArrays(…)
 bind (select) a specific VAO for initialization: glBindVertexArray(…)
 update VBOs associated with this VAO: glBindBuffer(…) and glVertexAttribPointer(…)
 bind VAO for use in rendering: glDrawArrays(…)

//
In order to be able to render 3D objects, we need to write at least a vertex shader and a fragment shader. For this we need to:
 generate a shader object (referenced by its unique ID): glCreateShader(…)
 attach the shader source code to the shader object: glShaderSource(…)
 compile the shader: glCompileShader(…)

After we have compiled the vertex and fragment shaders we need to combine them in a so called shader program object.
The steps for this are the following:

 create a shader program: glCreateProgram(…)
 attach the previous compiled shaders: glAttachShader(…)
 link them in the shader program: glLinkProgram(…)

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//vertex coordinates in normalized device coordinates
 GLfloat vertexCoordinates[] = {     0.0f, 0.5f, 0.0f,     0.5f, -0.5f, 0.0f, -0.5f, -0.5f, 0.0f };

 GLuint verticesVBO;
 GLuint triangleVAO;

  void initObjects() {
        //generate a unique ID corresponding to the verticesVBO
        glGenBuffers(1, &verticesVBO);


        //bind the verticesVBO buffer to the GL_ARRAY_BUFFER target,
        //any further buffer call made to GL_ARRAY_BUFFER will configure the
        //currently bound buffer, which is verticesVBO
        glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);


        //copy data into the currently bound buffer, the first argument specify
        //the type of the buffer, the second argument specify the size (in bytes) of data,
        //the third argument is the actual data we want to send,
        //the last argument specify how should the graphic card manage the data
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertexCoordinates), vertexCoordinates,
GL_STATIC_DRAW);


        //generate a unique ID corresponding to the triangleVAO
        glGenVertexArrays(1, &triangleVAO);
        glBindVertexArray(triangleVAO);
        glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);


        //set the vertex attributes pointers
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
        glEnableVertexAttribArray(0);


        //unbind the triangleVAO
        glBindVertexArray(0);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//VERTEX SHADER
 layout(location = 0) in vec3 vertex_position;
 out vec3 colour;

 void main() {
       //specify the vertex color
```

```
	colour = vec3(1.0, 0.0, 0.0);
	gl_Position = vec4(vertex_position, 1.0);
 }

//FRAGMENT SHADER
 in vec3 colour;
 out vec4 frag_colour;

 void main()
 {
	frag_colour = vec4 (colour, 1.0);

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Read, compile and link the shader programs
GLuint initBasicShader(std::string vertexShaderFileName, std::string fragmentShaderFileName)
{

	//read, parse and compile the vertex shader
	std::string v = readShaderFile(vertexShaderFileName);
	const GLchar* vertexShaderString = v.c_str();
	GLuint vertexShader;
	vertexShader = glCreateShader(GL_VERTEX_SHADER);

	glShaderSource(vertexShader, 1, &vertexShaderString, NULL);
	glCompileShader(vertexShader);

	//check compilation status
	shaderCompileLog(vertexShader);

	//read, parse and compile the vertex shader
	std::string f = readShaderFile(fragmentShaderFileName);
	const GLchar* fragmentShaderString = f.c_str();
	GLuint fragmentShader;
	fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
	glShaderSource(fragmentShader, 1, &fragmentShaderString, NULL);
	glCompileShader(fragmentShader);

	//check compilation status
	shaderCompileLog(fragmentShader);

	//attach and link the shader programs
	shaderProgram = glCreateProgram();
	glAttachShader(shaderProgram, vertexShader);
	glAttachShader(shaderProgram, fragmentShader);
	glLinkProgram(shaderProgram);

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

```
        //check linking info
        shaderLinkLog(shaderProgram);
        return shaderProgram;

}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//Finally, we need to specify which primitive OpenGL should render. For this purpose, we use the glDrawArrays function, which
// employs the current active shader, VAO and VBOs in order to draw the specified primitive.

void renderScene() {

        //clear the color and depth buffer before rendering the current frame
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        //specify the background color
        glClearColor(0.8, 0.8, 0.8, 1.0);

        //specify the viewport location and dimension
        glViewport (0, 0, retina_width, retina_height);

        //process the keyboard inputs
        if (glfwGetKey(glWindow, GLFW_KEY_A)) {
                //TODO
        }

        if (glfwGetKey(glWindow, GLFW_KEY_D)) {
                //TODO
        }

        //bind the shader program, any further rendering call
        //will use this shader program

        glUseProgram(shaderProgram);

        //bind the VAO
        glBindVertexArray(triangleVAO);

        //specify the type of primitive, the starting index and
        //the number of indices to be rendered
        glDrawArrays(GL_TRIANGLES, 0, 3);
}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//The typical structure of a shader is the following:
 //OpenGL version
 #version versionNumber
```

```
//list of input variables
in type inputVariableName;

//list of output variables
out type outputVariableName;

//list of uniforms
uniform type uniformName;

void main() {
        //implements the functionality and output the variables
        outputVariableName = processVariable();
}
```

//

vecn: vector of n floats
- bvecn: vector of n booleans
- ivecn: vector of n integers
- uvecn: vector of n unsigned integers
- dvecn: vector of n double components

If the name of an input variable (from one shader) matches the name
of an output variable (from another shader) then the data passes from one shader to the other.

In order to match the vertex attributes specified at the CPU level with the ones defined at the GPU level,
we use the following statement: layout (location = locationID).

//

Another way of transferring data from our application (which runs on the CPU) to
our shaders (which run on the GPU) is by UNIFORMS. They are global (meaning unique for a shader program object),
and can be accessed by any shader of the shader program. In addition, their value will remain the same until we modify it.

//

```
//declare a uniform
uniform type uniformName;

//In order to update the uniform, we need (at application level) to get the uniform location and then we can change its value.

glUseProgram(shaderProgram);
```

```
GLint uniformLocation = glGetUniformLocation(shaderProgram, "uniformName");
glUniform3f(uniformLocation, 0.15f, 0.0f, 0.84);
```

//

```
//In order to get the color value, we need to modify the vertex shader => set corresponding location
layout(location = locationID1) in vec3 vertexPosition;
layout(location = locationID2) in vec3 vertexColor;

//We also need to change the way in which we specify the vertices attributes in our program

//vertex position attribute
glVertexAttribPointer(locationID1, size, type, normalized, stride, pointer);
glEnableVertexAttrivArray(locationID1);

//vertex colour attribute
glVertexAttribPointer(locationID2, size, type, normalized, stride, pointer);
glEnableVertexAttribArray(locationID2);
```

//

```
//In order to avoid duplicating (re-defining) some vertices, we can use EBOs to store indices.
//Indices will be used by OpenGL to decide what vertices to choose from when forming polygons.
//The procedure of creating EBOs is very similar to the one used to define VBOs:

glGenBuffers(1, &verticesEBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices,
GL_STATIC_DRAW);

//When calling glDrawElements, we draw using the indices provided within the EBO currently
bound.
//The first parameter specifies the type of the primitive, the second represents the number of
elements we draw (vertices),
//the third parameter represents the type of indices (typically GL_UNSIGNED_INT) and the last
one is an offset.

glDrawElements(GL_TRIANGLES, size, type, offset);
```

//

3.1 Transfering data from vertex to fragment shader

Start from the solution that you find on the website and modify the vertex and fragment shaders.
Remember that if the name of an input variable (from one shader) matches the name of
an output variable (from another shader), then the data passes from one shader to the other one.

The code from vertex shader:

```
#version 400
```

```
layout(location = 0) in vec3 vertexPosition;

//the color variable that we will send to the fragment shader
out vec3 colour;

void main() {
	colour = vec3(0.74, 0.16, 0.0);
	gl_Position = vec4(vertexPosition, 1.0);
}
```

The code from fragment shader:

```
#version 400
//the color variable that we received from the vertex shader
in vec3 colour;
out vec4 fragmentColour;
void main()
{
	fragmentColour = vec4(colour, 1.0);
}
```

//

glGetUniformLocation returns an integer that represents the location of a specific uniform variable within a program object.

//

```
//Draw a rectangle composed of two triangles. EBO will be used to save storage space.
//Define, globally, the vertex data (representing position and color) and the buffers that will be used for it.
GLfloat vertexData[] = {
//vertex position and vertex color
	-0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f,
	-0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
	0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f,
	0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f };


GLuint vertexIndices[] = {
	0, 1, 2,
	0, 2, 3 };


GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;

 //We need to update the initialization of our object. The difference is that, in this case,
```

```
 //we have also indices and a new attribute for each vertex.
 //For details about the parameters needed for each function.

 void initObjects() {
        glGenVertexArrays(1, &objectVAO);
        glBindVertexArray(objectVAO);

        glGenBuffers(1, &verticesVBO);
        glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData,
GL_STATIC_DRAW);

        glGenBuffers(1, &verticesEBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices,
GL_STATIC_DRAW);

        //vertex position attribute
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
        glEnableVertexAttribArray(0);


        //vertex colour attribute
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 *
sizeof(float)));
    glEnableVertexAttribArray(1);


    glBindVertexArray(0);
}

    //In the renderScene() function, we need to update the drawing call to glDrawElements.

glBindVertexArray(objectVAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

//Vertex shader:

#version 400
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;
out vec3 colour;

void main() {
        colour = vertexColour;
        gl_Position = vec4(vertexPosition, 1.0);
}

//Fragment shader:
#version 400

in vec3 colour;
```

```
out vec4 fragmentColour;

void main() {
	fragmentColour = vec4(colour, 1.0);
}
```