CAPITULO 2

OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

2.2 Transformaciones Geométricas en 2D

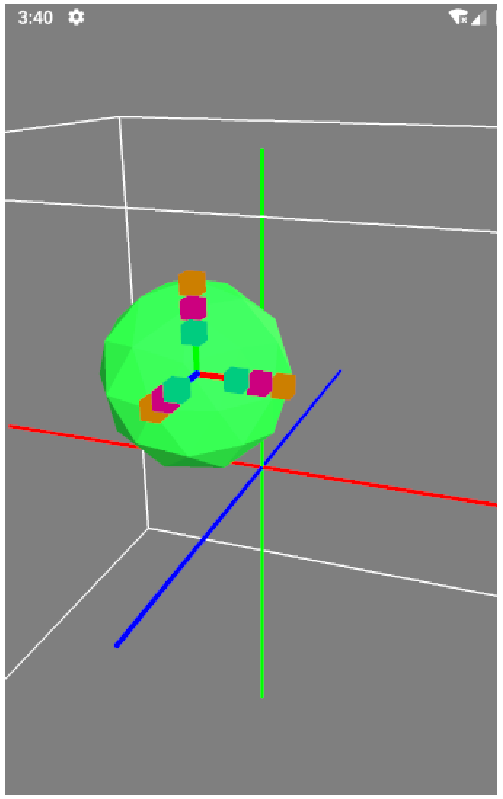
2.2.4 Core Concepts on Transformations 2D/3D

T2D/3D – In Practice

OpenGL does not have any form of matrix or vector knowledge built in, so we have to define our own mathematics classes and functions.

We would rather abstract from all the tiny mathematical details and simply use pre-made mathematics libraries.

Luckily, there is an easy-to-use and tailored-for-**OpenGL mathematics library** called **GLM**.



T2D/3D – In Practice - GLM

- GLM stands for OpenGL Mathematics and is a header-only library, which means that we only have to include the proper header files and we're done; no linking and compiling necessary.
- Download Link: https://github.com/g-truc/glm
- Copy the root directory of the header files into your includes folder and let's get rolling.

Most of GLM's functionality that we need can be found in 3 headers files that we'll include as follows:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

T2D/3D – In Practice - GLM

- Let's see if we can put our transformation knowledge to good use by translating a vector of **(1,0,0)** by **(1,1,0).**
- **N**ote that we define it as a glm::vec4 with its homogenous coordinate set to 1.0:

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

T2D/3D – In Practice

Exercise 10 Task 1:
Implement the example and test the output of the transformation matrix.

T2D/3D – In Practice

Exercise 10 Task 1:
Implement the example and test the output of the transformation matrix.

- Download the lastest versión of GLM

  https://github.com/g-truc/glm

- Unzip and copy the /glm/ folder to your OpenGL include folder: OpenGL_Stuff\include
- Include the GLM library in your OPENGL program.

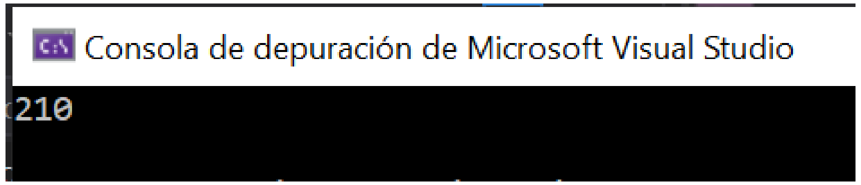
```
[…]
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
[…]
```

T2D/3D – In Practice

Exercise 10 Task 1:
Implement the example and test the output of the transformation matrix.

- Program and test your transformation matrix

```
[...]
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
[...]

int main () {
   glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
   glm::mat4 trans = glm::mat4(1.0f);
   trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
   vec = trans * vec;
   std::cout << vec.x << vec.y << vec.z << std::endl;
}
```


Consola de depuración de Microsoft Visual Studio

```
210
```

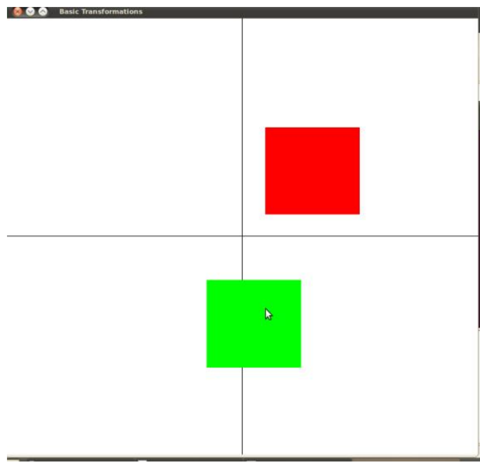
T2D/3D – In Practice - GLM

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

- We first define a vector named vec using GLM's built-in vector class.

- Next we define a mat4 and explicitly initialize it to the identity matrix by initializing the matrix's diagonals to **1.0**;



> if we do not initialize it to the identity matrix the matrix would be a **null matrix (all elements 0)** and all subsequent matrix operations would end up a null matrix as well.

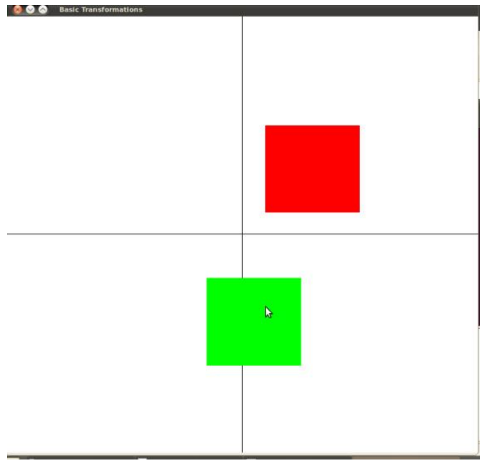
- The next step is to create a **transformation matrix** by passing our **identity matrix** to the glm::translate function, together with a **translation vector** (the given matrix is then multiplied with a translation matrix and the resulting matrix is returned).

T2D/3D – In Practice - GLM

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
std::cout << vec.x << vec.y << vec.z << std::endl;
```

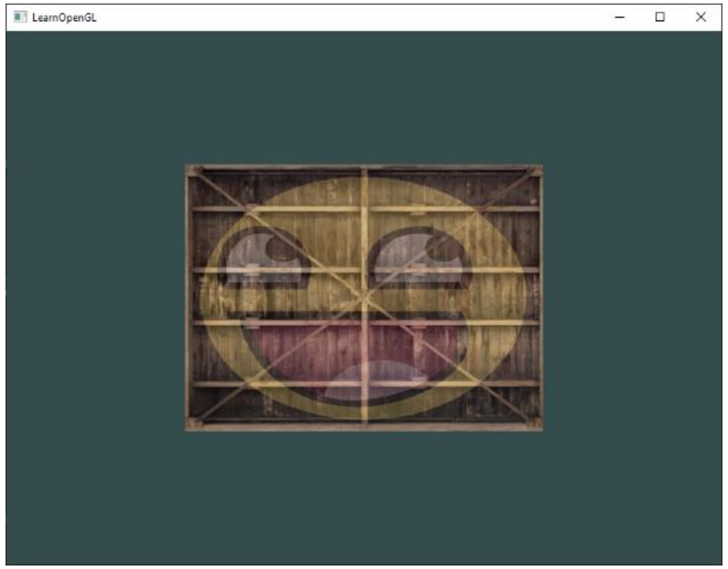
- Then we multiply our vector by the transformation matrix and output the result.
- If we still remember how matrix translation works then the resulting vector should be **(1+1,0+1,0+0)** which is (2,1,0).
- This snippet of code outputs 210 so the translation matrix did its job.



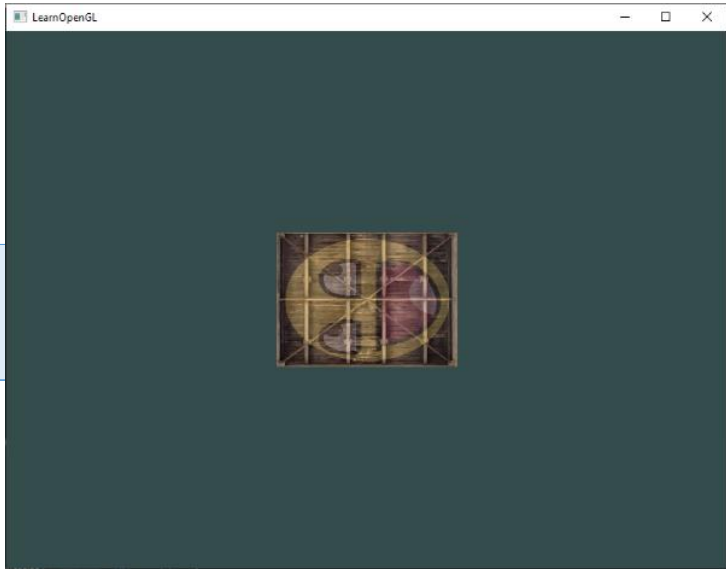
T2D/3D – In Practice - GLM

Example: Scale and rotate the container object from the previous chapter:

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```



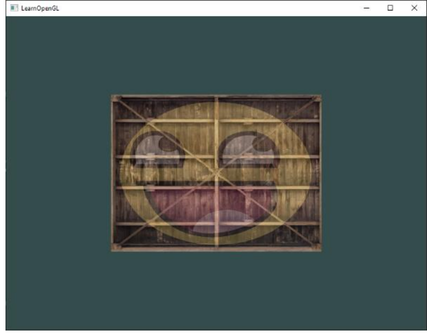
- First, we **scale** the container by **0.5 on each axis** and then **rotate the container 90 degrees** around the Z-axis. Note that the textured rectangle is on the XY plane so we want to rotate around the Z-axis.
- GLM expects its angles in radians so we convert the degrees to radians using glm::**radians**.



Keep in mind that the axis that we rotate around should be a **unit vector**, so be sure to <u>normalize the vector first if you're not rotating around the X, Y, or Z axis</u>

- Because we pass the matrix to each of GLM's functions, GLM automatically multiples the matrices together, resulting in a transformation matrix that combines all the transformations.

T2D/3D – In Practice - GLM



How do we get the transformation matrix to the shaders?

GLSL has a **mat4** type.

- So we'll adapt the **vertex shader** to accept a mat4 uniform variable and multiply the position vector by the matrix uniform:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
   gl_Position = transform * vec4(aPos, 1.0f);
   TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

We added the uniform and multiplied the position vector with the transformation matrix before passing it to gl_Position.

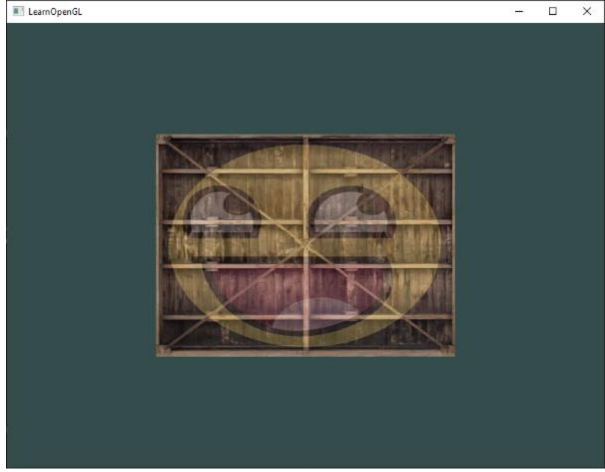
GLSL also has `mat2` and `mat3` types that allow for swizzling-like operations just like vectors. All the aforementioned math operations (like scalar-matrix multiplication, matrix-vector multiplication and matrix-matrix multiplication) are allowed on the matrix types. Wherever special matrix operations are used we'll be sure to explain what's happening.

T2D/3D – In Practice - GLM

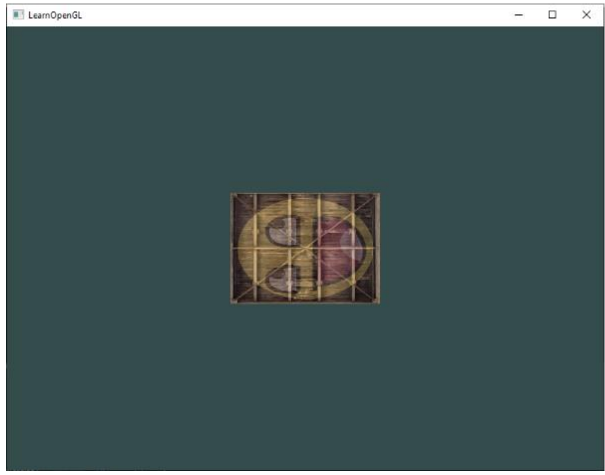
How do we get the transformation matrix to the shaders?

We query the location of the uniform variable and then send the matrix data to the shaders using **glUniform** with **Matrix4fv** as its postfix.

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```



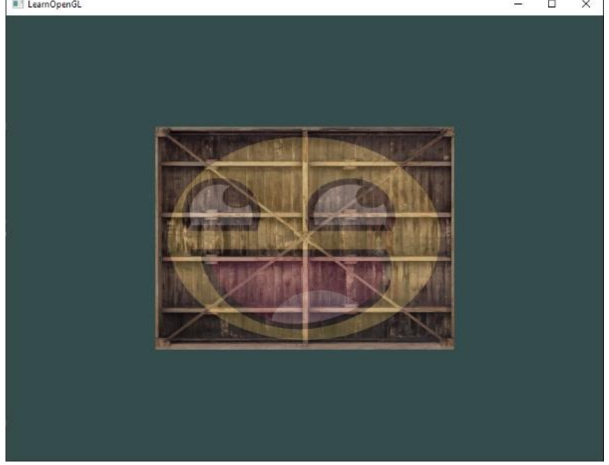
- The **first argument** should be familiar by now which is the uniform's location.
- The **second argument** tells OpenGL how many matrices we'd like to send, which is 1.
- The **third argument** asks us if we want to **transpose** our matrix, that is to swap the columns and rows. OpenGL developers often use an internal matrix layout called column-major ordering which is the default matrix layout in GLM so there is no need to transpose the matrices; we can keep it at GL_FALSE.
- The **last parameter** is the actual matrix data, but GLM stores their matrices' data in a way that doesn't always match OpenGL's expectations, so we first convert the data with GLM's built-in function **value_ptr**.



T2D/3D – In Practice - GLM

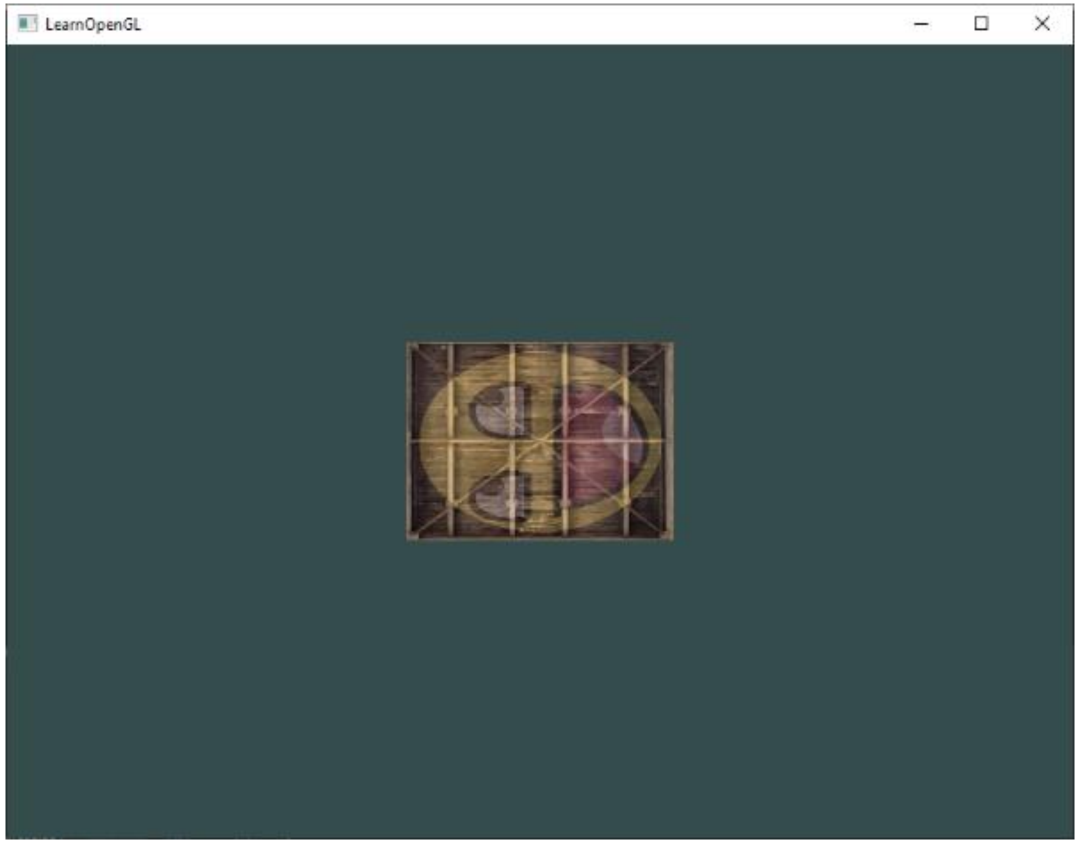
How do we get the transformation matrix to the shaders?

We created a transformation matrix, declared a uniform in the vertex shader and sent the matrix to the shaders where we transform our vertex coordinates.



transformation matrix

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```



vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 transform;

void main()
{
   gl_Position = transform * vec4(aPos, 1.0f);
   TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

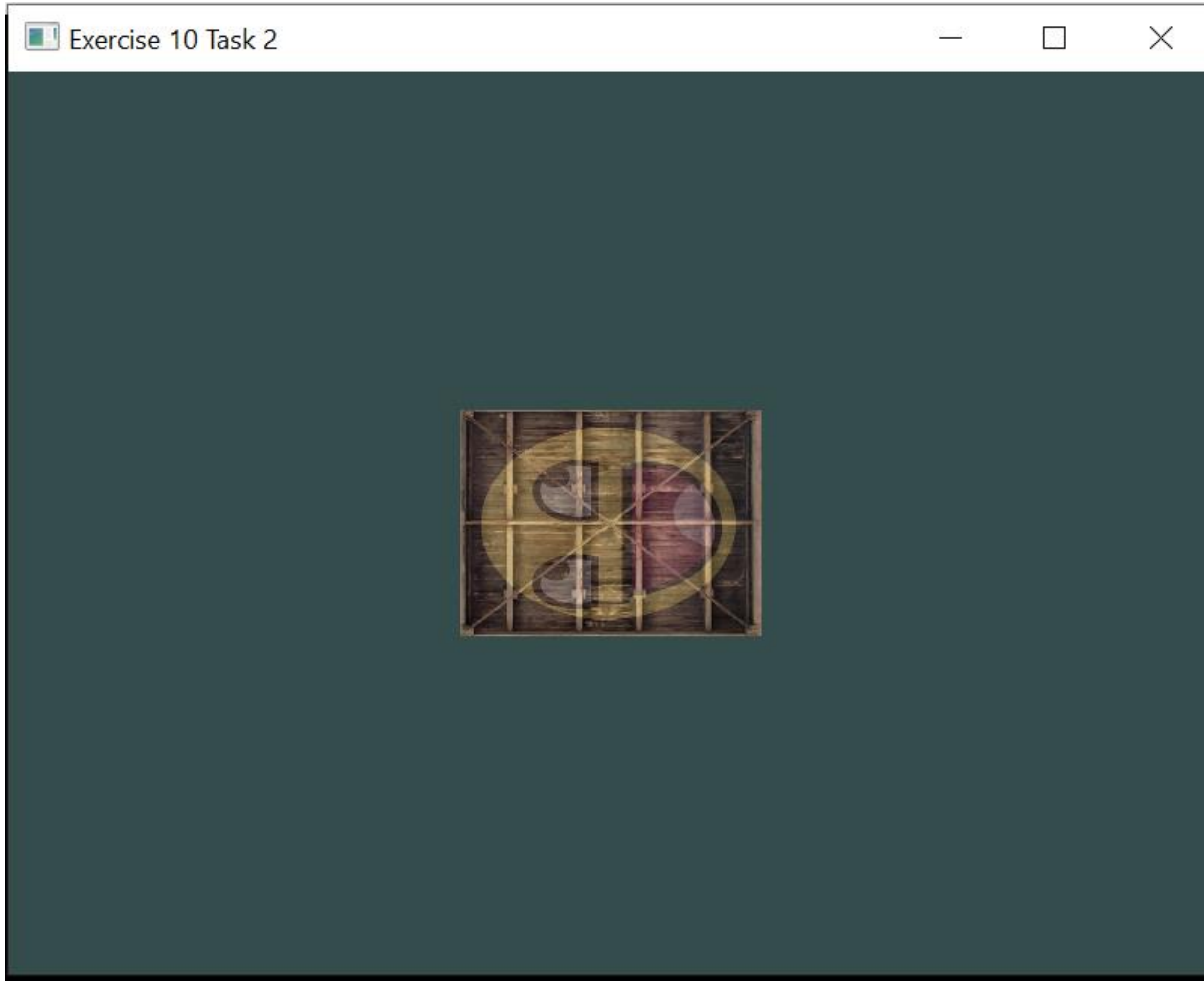
sent the matrix to the shaders

```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

T2D/3D – In Practice - GLM

Exercise 10 Task 2:

- Scale and rotate the container object from the previous chapter (C2_Exercise_10_T2D_Task2)

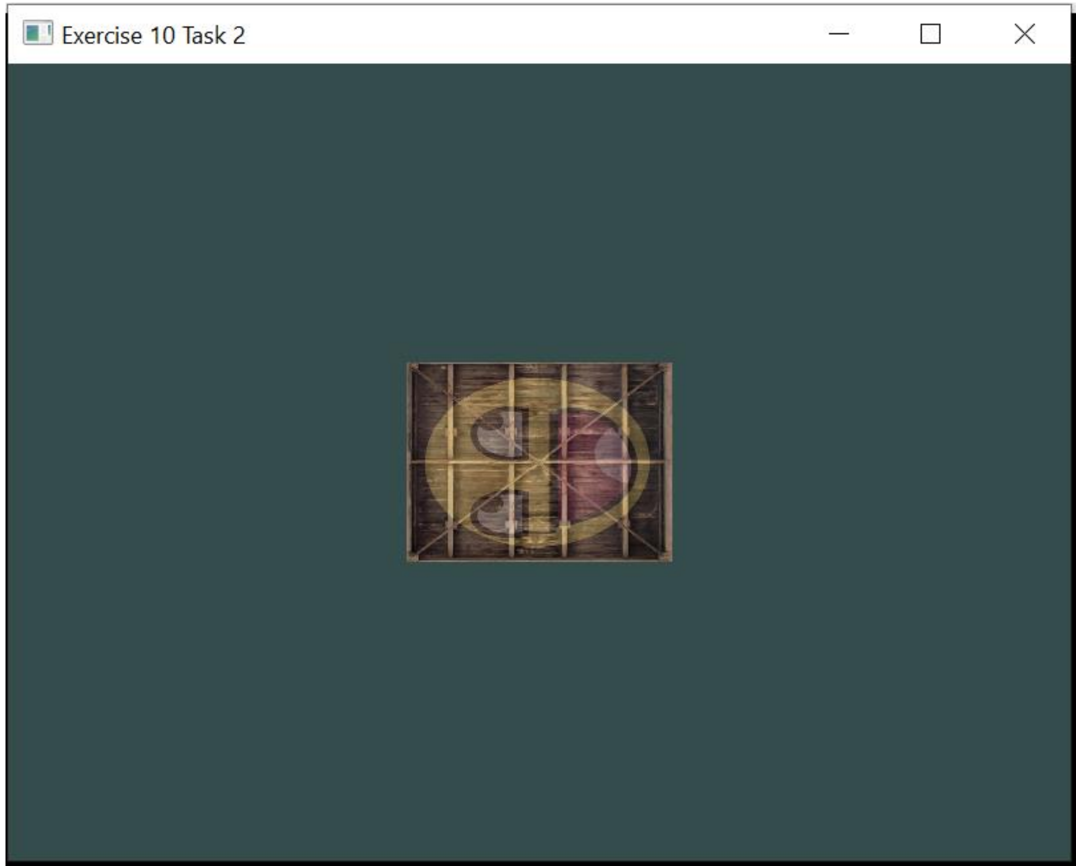


T2D/3D – In Practice - GLM

Exercise 10 Task 2:

- Scale and rotate the container object from the previous chapter (C2_Exercise_10_T2D_Task2)

```
[...]
// create transformations
   glm::mat4 trans = glm::mat4(1.0f);
   trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
   trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));

   // get matrix's uniform location and set matrix
   ourShader.use();
   unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
   glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
[...]
//render loop
[...]
```



T2D/3D – In Practice - GLM

Let's get a little more funky and see if we can **rotate the container over time**, and for fun we'll also **reposition the container at the bottom-right side of the window**.

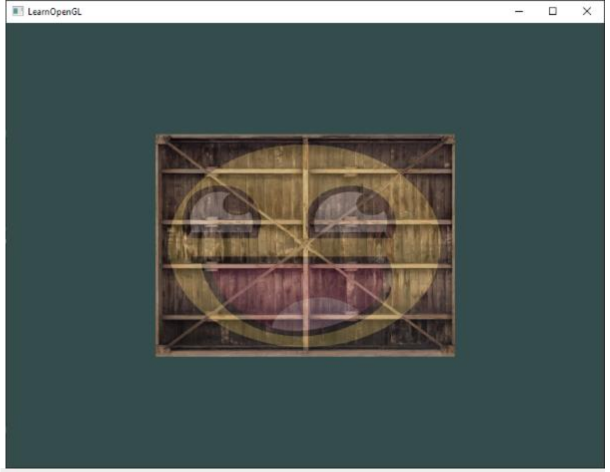
To rotate the container over time we have to update the transformation matrix in the render loop because it needs to update each frame.

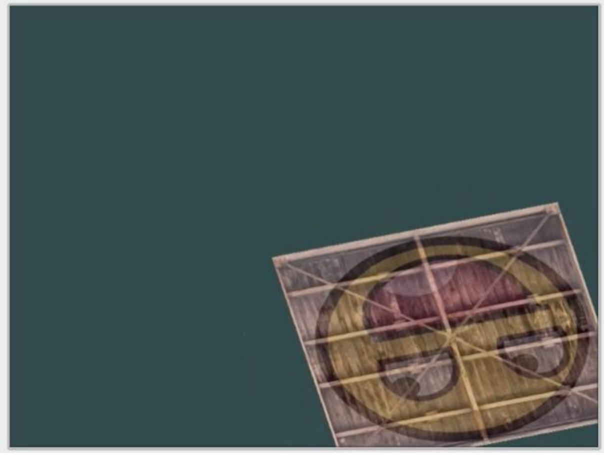
We use GLFW's time function to get an angle over time:

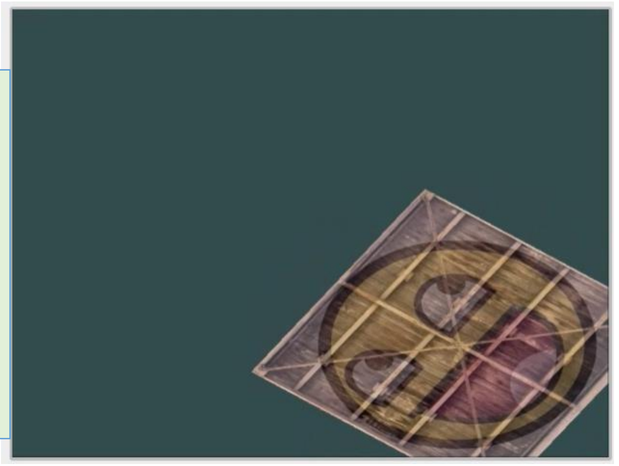
```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

Keep in mind that in the previous case we could declare the transformation matrix anywhere, but **now we have to create it every iteration to continuously update the rotation**. This means we have to re-create the transformation matrix in each iteration of the **render loop**.

Usually when rendering scenes we have several transformation matrices that are re-created with new values each frame.







T2D/3D – In Practice - GLM

Let's get a little more funky and see if we can **rotate the container over time**, and for fun we'll also **reposition the container at the bottom-right side of the window**.

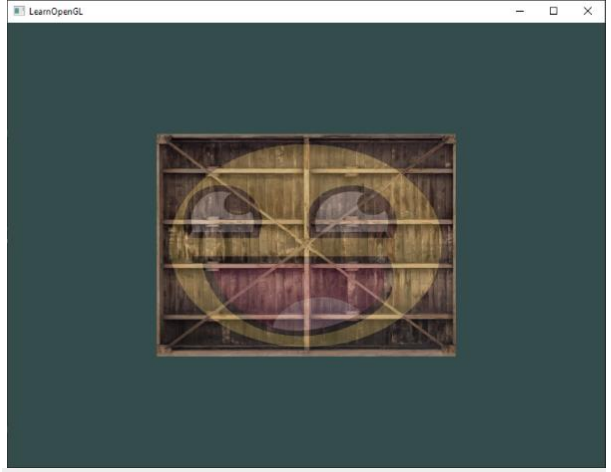
```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

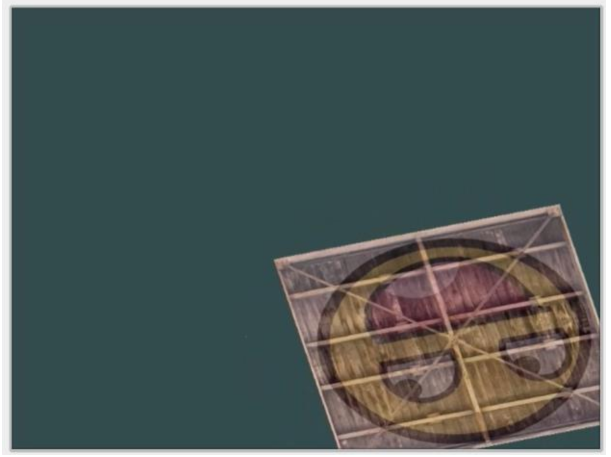
- Here we first rotate the container around the origin (0,0,0) and once it's rotated, we translate its rotated version to the bottom-right corner of the screen.

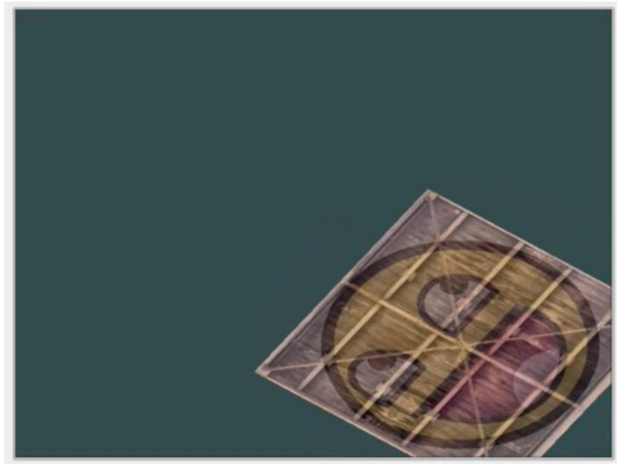
- Remember that the actual transformation order should be read in reverse: even though in code we first translate and then later rotate, the actual transformations first apply a rotation and then a translation.

- Understanding all these combinations of transformations and how they apply to objects is difficult to understand.

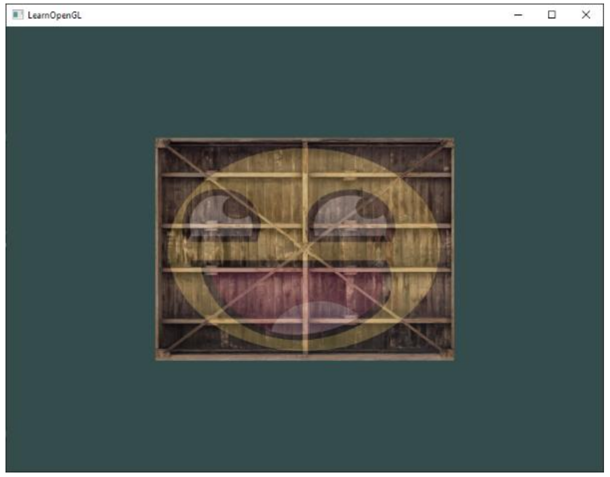
- **Try and experiment with transformations like these and you'll quickly get a grasp of it.**

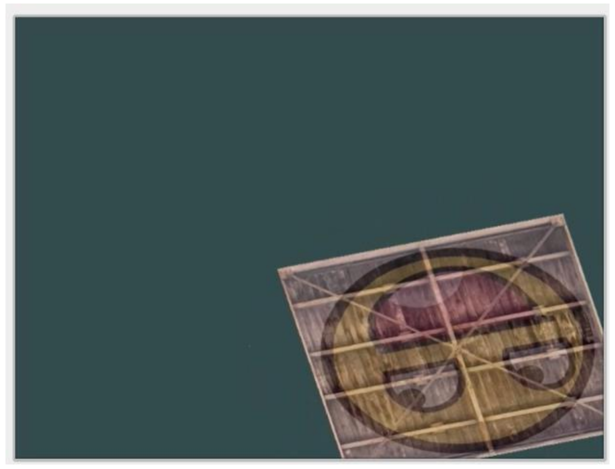


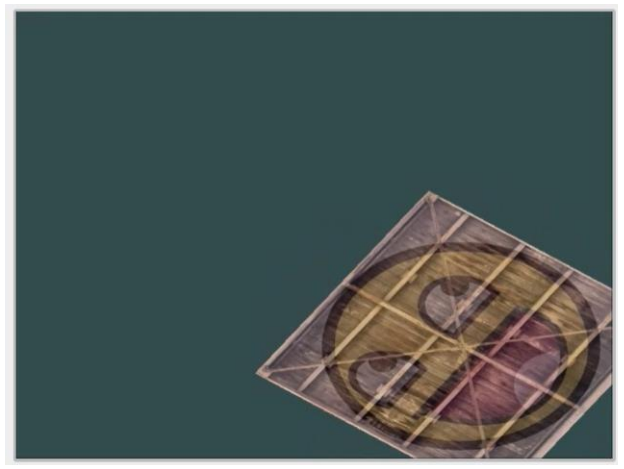




T2D/3D – In Practice - GLM







A translated container that's rotated over time, all done by a single transformation matrix! Now you can see why matrices are such a powerful construct in graphics land.

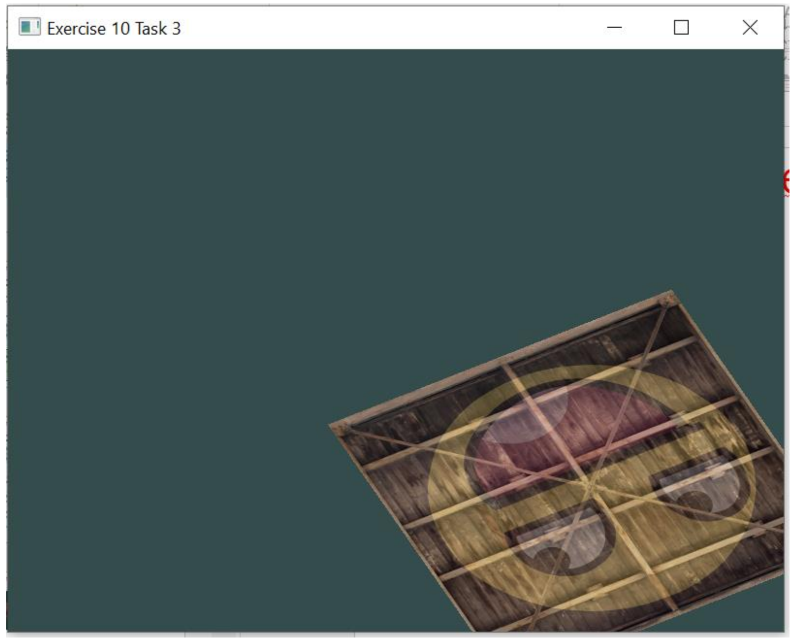
We can define an infinite amount of transformations and combine them all in a single matrix that we can re-use as often as we'd like.

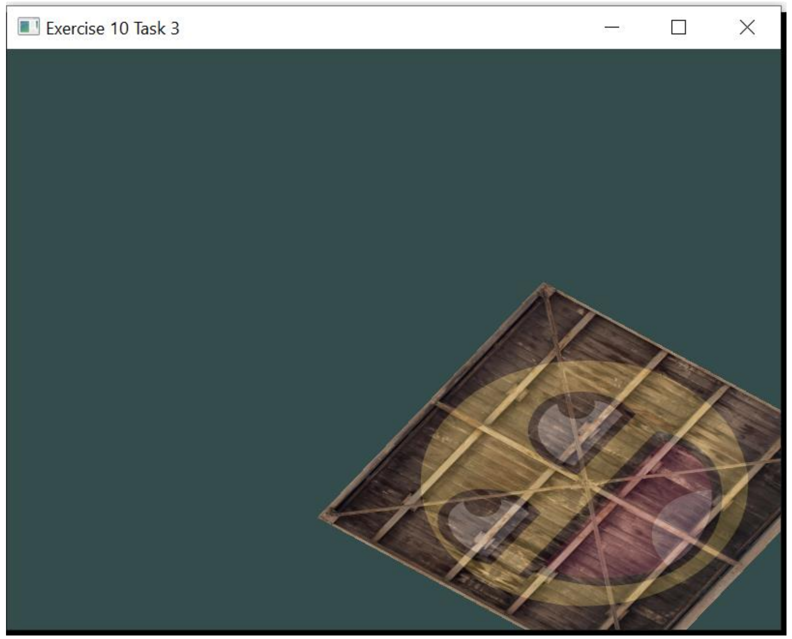
Using transformations like this in the vertex shader saves us the effort of re-defining the vertex data and saves us some processing time as well, since we don't have to re-send our data all the time (which is quite slow); all we need to do is update the transformation uniform.

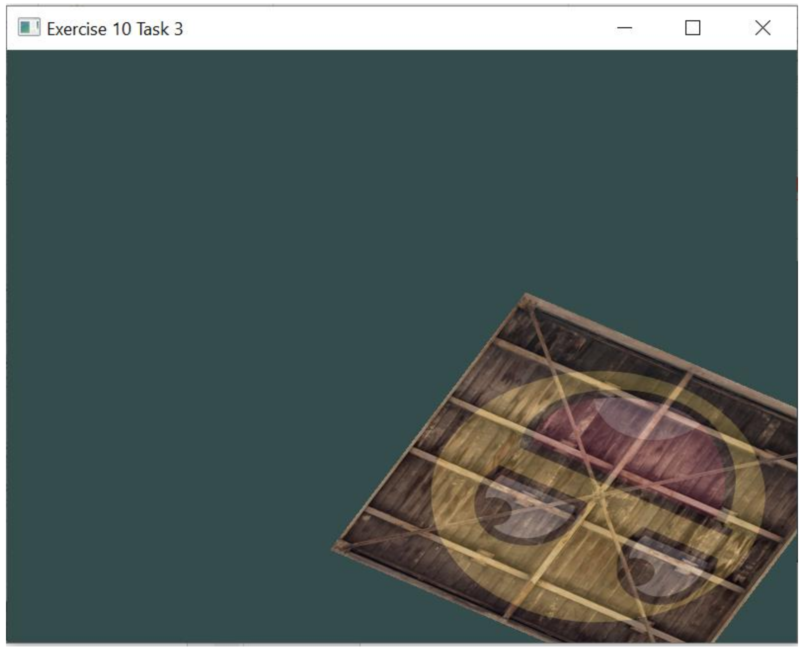
T2D/3D – In Practice - GLM

Exercise 10 Task 3:

- Rotate the container over time (C2_Exercise_10_T2D_Task3)





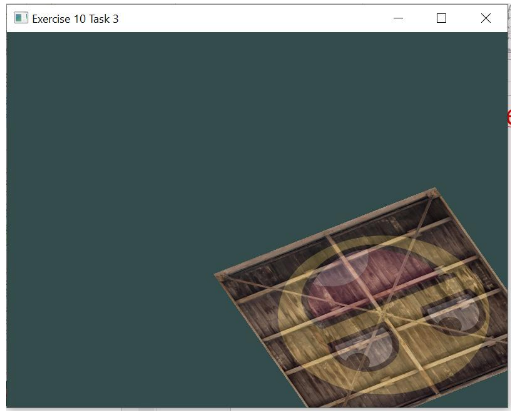


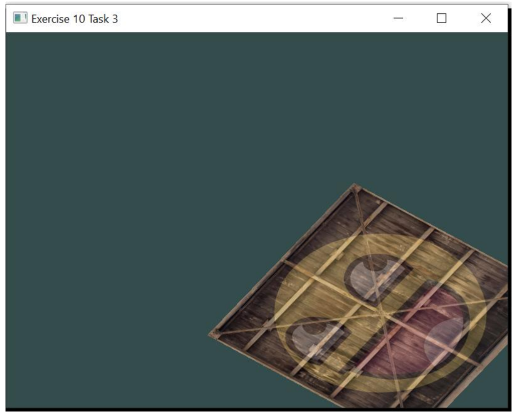
T2D/3D – In Practice - GLM

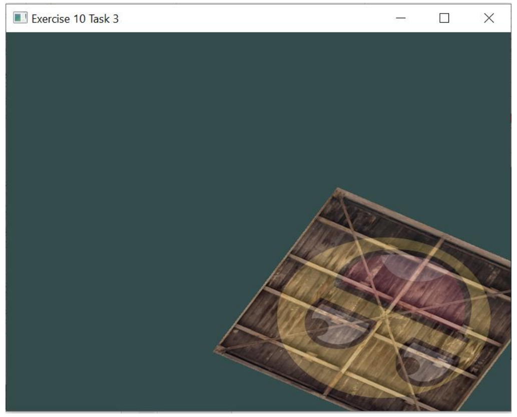
Exercise 10 Task 3:

- Rotate the container over time (C2_Exercise_10_T2D_Task3)

```
[...]
//In render loop
while (!glfwWindowShouldClose(window))
   {
    [...]
      //Exercise 10 Task3
     // create transformations
     glm::mat4 transform = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
     transform = glm::translate(transform, glm::vec3(0.5f, -0.5f, 0.0f));
     transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));

     // get matrix's uniform location and set matrix
     ourShader.use();
     unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
     glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform));
[...]
}
[...]
```







T2D/3D – In Practice

Exercise 10 Task 4:

Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens:

T2D/3D – In Practice

Exercise 10 Task 4:

Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens:

Option A

```
glm::mat4 transform = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
transform = glm::translate(transform, glm::vec3(0.5f, -0.5f, 0.0f));
transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

⇩

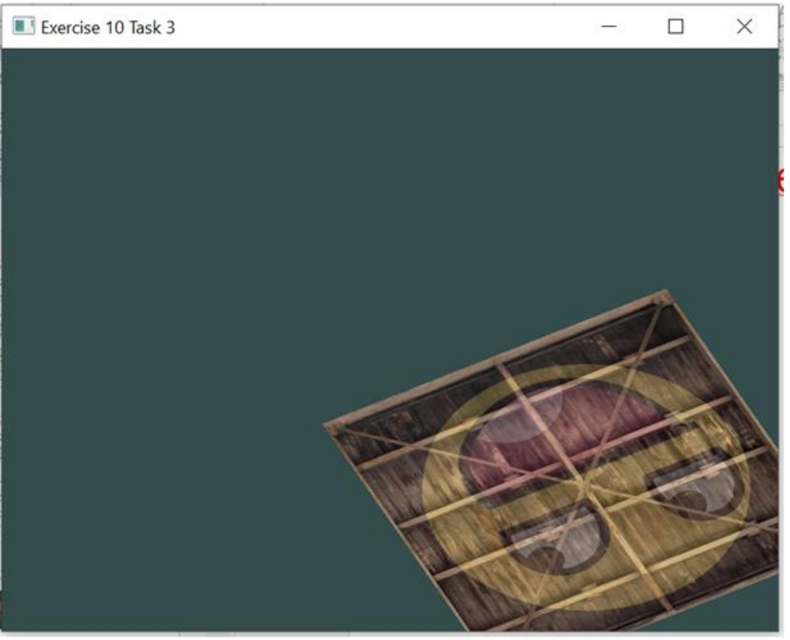
Option B

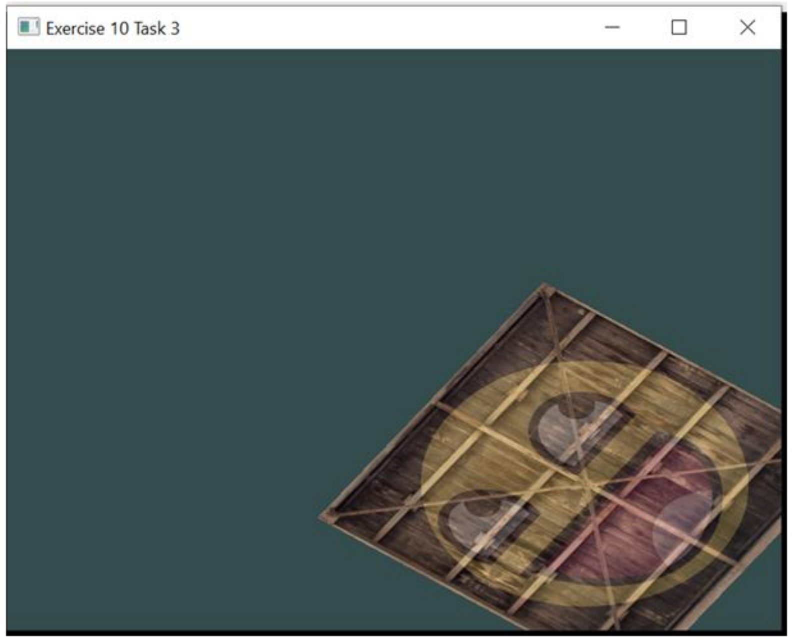
```
glm::mat4 transform = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
transform = glm::translate(transform, glm::vec3(0.5f, -0.5f, 0.0f));
```

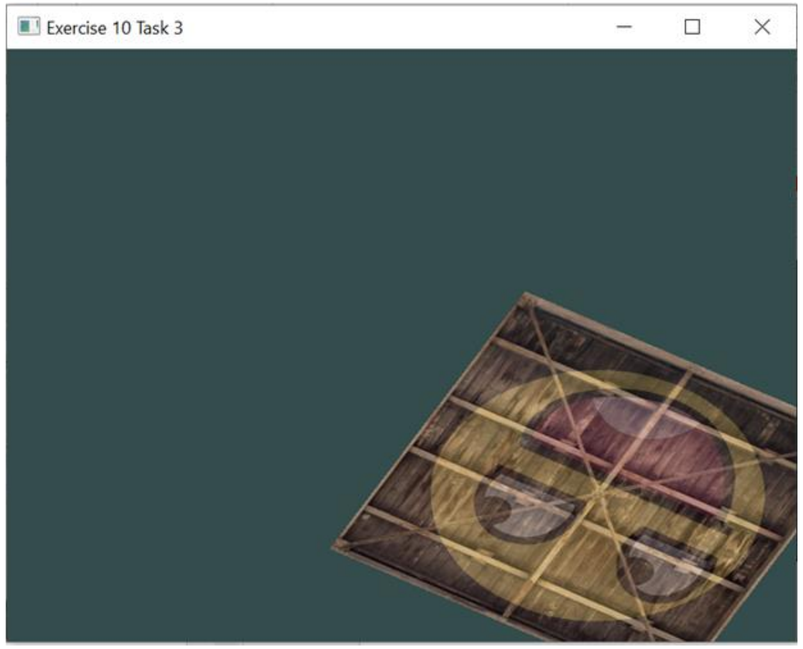
T2D/3D – In Practice

Exercise 10 Task 4:
Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens:

Option A





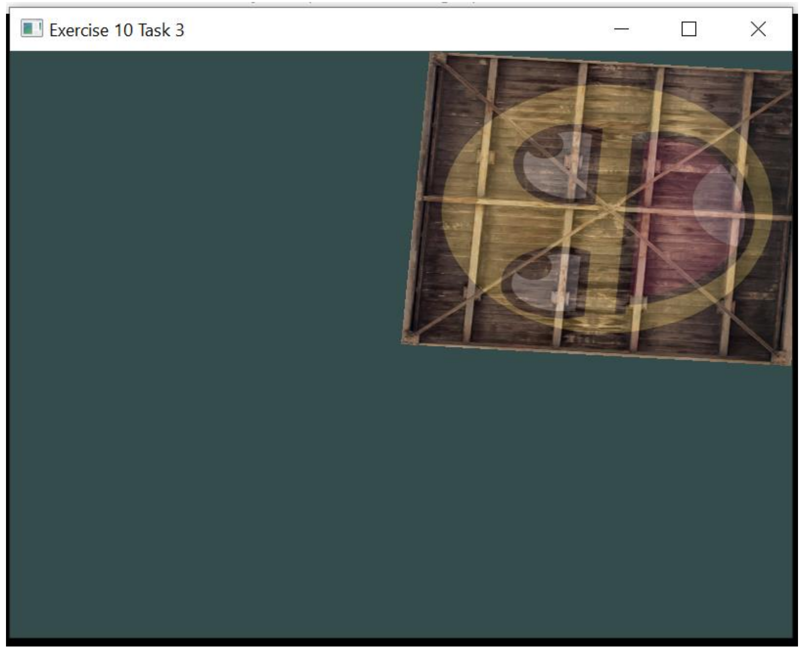


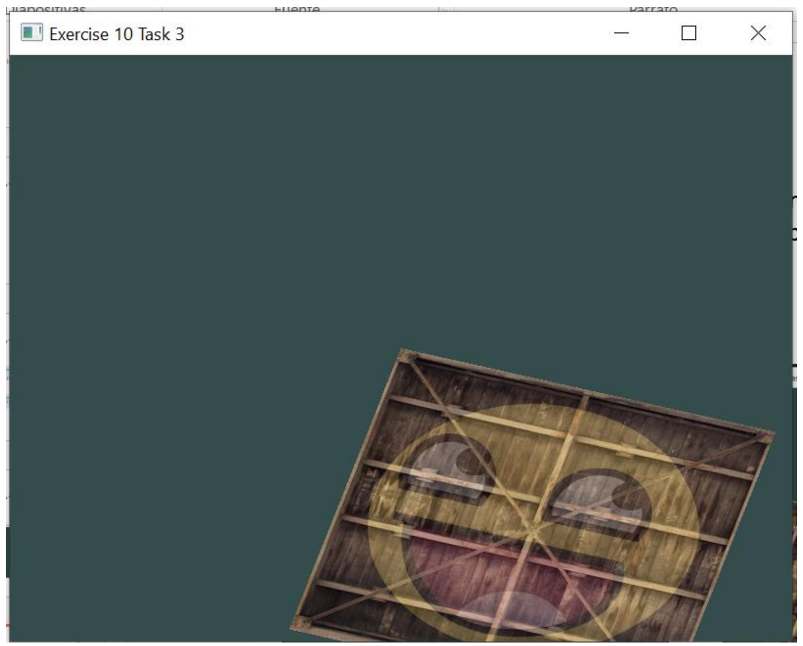
T2D/3D – In Practice

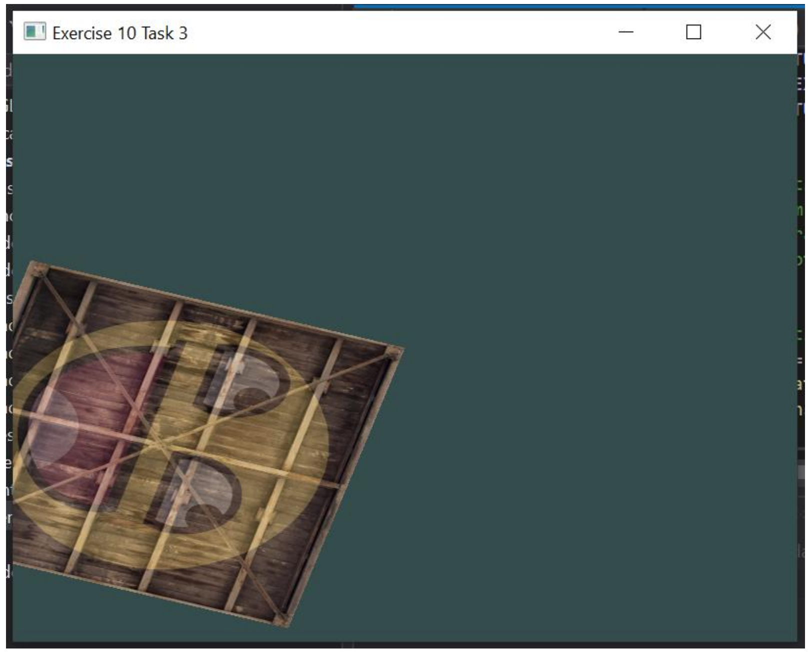
Exercise 10 Task 4:

Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens:

Option B



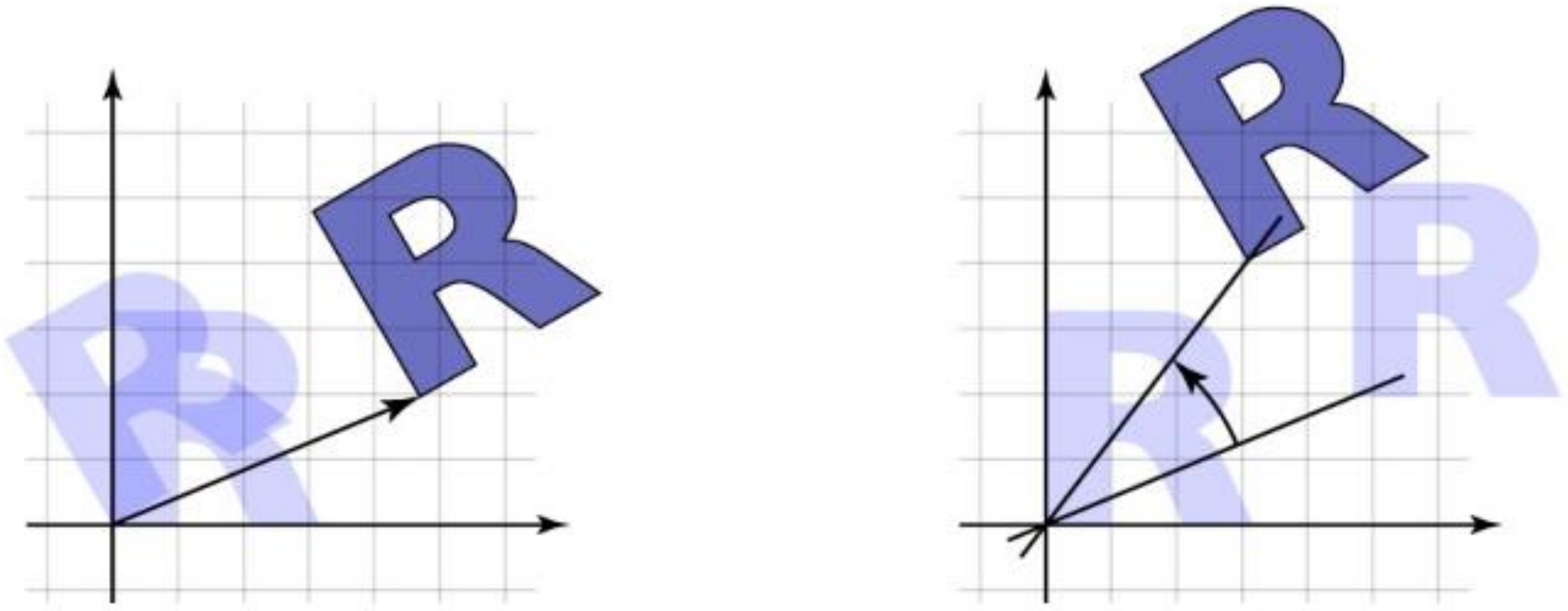




T2D/3D – In Practice

Exercise 10 Task 4:
Using the last transformation on the container, try switching the order around by first rotating and then translating. See what happens and try to reason why this happens:

- In general **not** commutative: order matters!


rotate, then translate   translate, then rotate

T2D/3D – In Practice

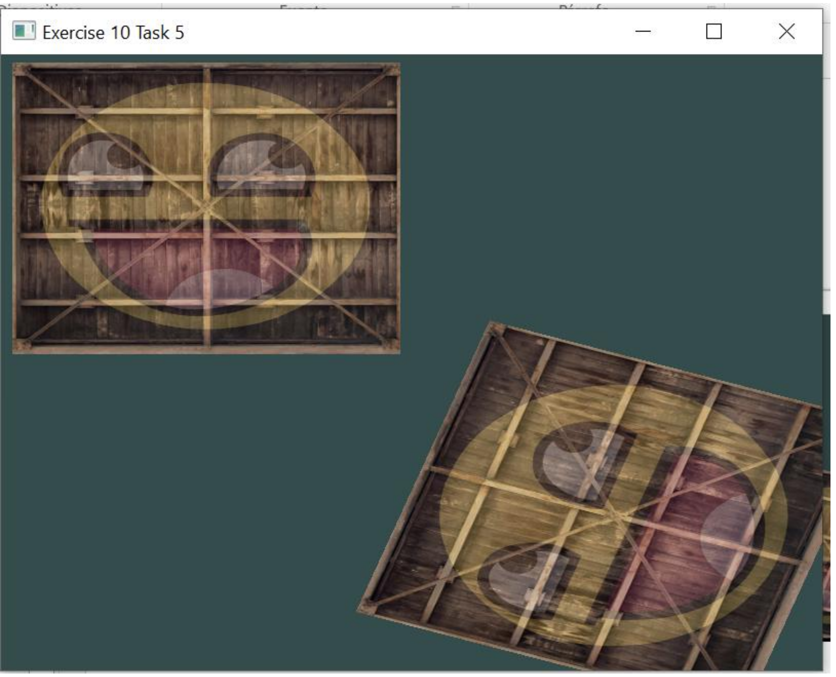
Exercise 10 Task 5:

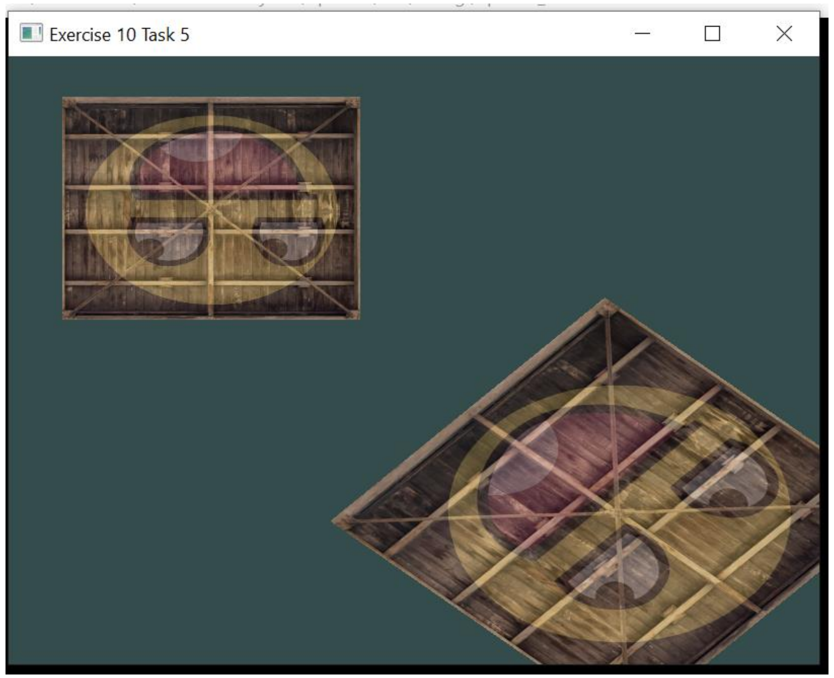
- Try drawing a second container with another call to **glDrawElements** but place it at a different position using transformations only.
- Make sure this second container is placed at the **top-left of the window** and instead of rotating, **scale it over time** (using the sin function is useful here; note that using **sin** will cause the object to invert as soon as a negative scale is applied):

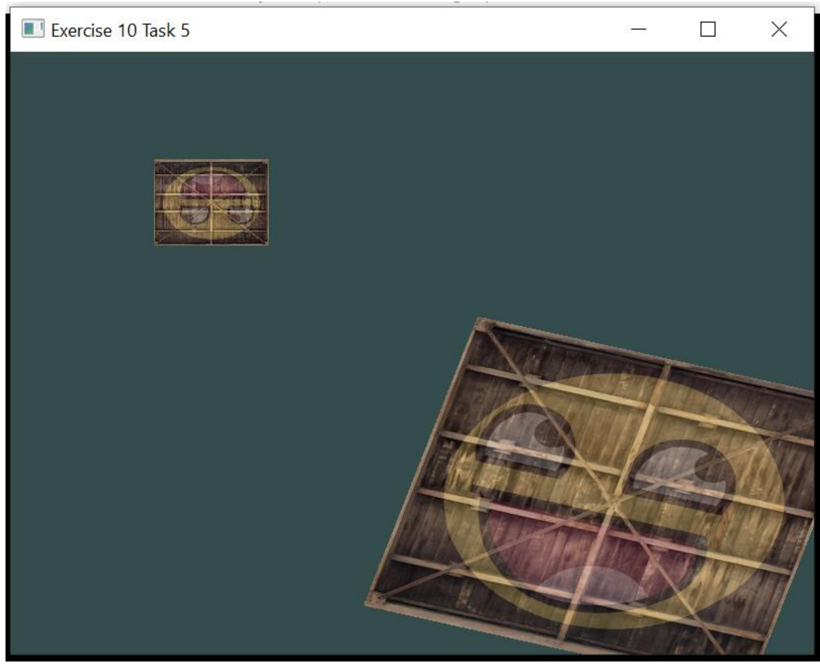
T2D/3D – In Practice

Exercise 10 Task 5:

- Try drawing a second container with another call to **glDrawElements** but place it at a different position using transformations only.
- Make sure this second container is placed at the **top-left of the window** and instead of rotating, **scale it over time** (using the sin function is useful here; note that using **sin** will cause the object to invert as soon as a negative scale is applied):







T2D/3D – In Practice

Exercise 10 Task 5: In render loop

```
glm::mat4 transform = glm::mat4(1.0f);
transform = glm::translate(transform, glm::vec3(0.5f, -0.5f, 0.0f));
transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));

// get matrix's uniform location and set matrix
ourShader.use();
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform));

 // render container
 glBindVertexArray(VAO);
 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
...
```

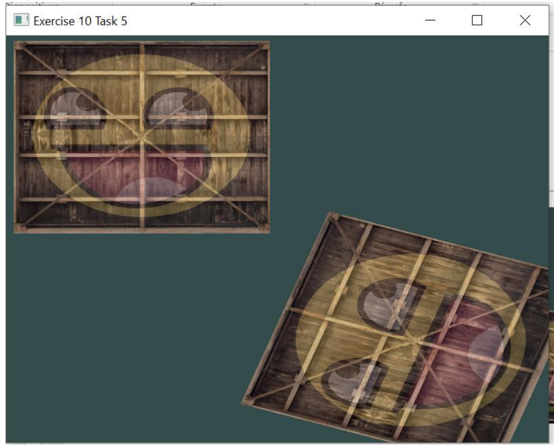
```
transform = glm::mat4(1.0f); //RESET THE TRANSFORMATION MATRIX
transform = glm::translate(transform, glm::vec3(-0.5f, 0.5f, 0.0f));
float scale = sin(glfwGetTime());
transform = glm::scale(transform, glm::vec3(scale, scale, scale));

// get matrix's uniform location and set matrix
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform));
//another option
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, &transform[0][0]);

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

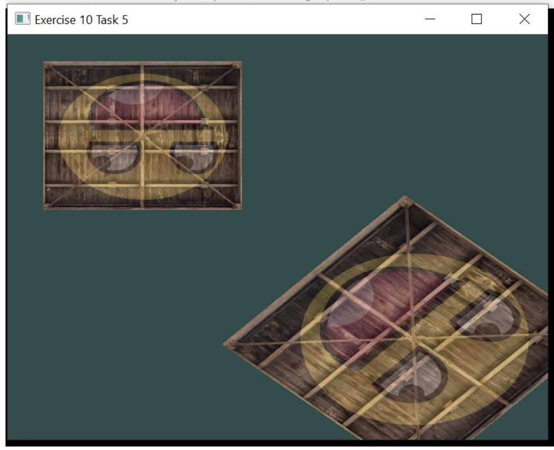
T2D/3D – In Practice

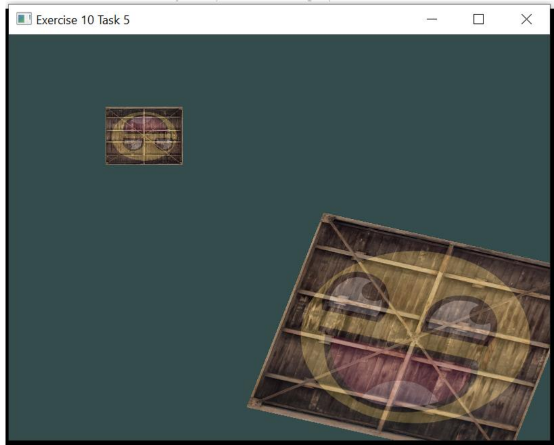
Exercise 10 Task 5:

```
glm::mat4 transform = glm::mat4(1.0f);
transform = glm::translate(transform, glm::vec3(0.5f, -0.5f, 0.0f));
transform = glm::rotate(transform, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));

// get matrix's uniform location and set matrix
ourShader.use();
unsigned int transformLoc = glGetUniformLocation(ourShader.ID, "transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform));

 // render container
 glBindVertexArray(VAO);
 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
...
```

```
transform = glm::mat4(1.0f); //RESET THE TRANSFORMATION  MATRIX
transform = glm::translate(transform, glm::vec3(-0.5f, 0.5f, 0.0f));
float scale = sin(glfwGetTime());
transform = glm::scale(transform, glm::vec3(scale, scale, scale));

// get matrix's uniform location and set matrix
//glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(transform));
//another option
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, &transform[0][0]);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
...
```


Exercise 10 Task 5


Exercise 10 Task 5


Exercise 10 Task 5