# Laboratory work 4

## 1 Objectives

This laboratory presents the key notions on 3D transformations. It will cover the basic model (translation, scale, rotation), camera and projection transformations.

## 2 Defining 2D and 3D points

A 2D point is defined in a homogenous coordinate system by (x*w, y*w, w). For simplicity, in bi-dimensional systems the w parameter is set to 1. Therefore, the point definition is (x, y, 1).

Another representation for the point is the following: $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

Similarly, in 3D we define points as (x*w, y*w, z*w, w) and we represent the point as a column vector:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

## 3 Transformations

After being processed by the Vertex Shader, the vertices that are to be rendered are expected to have their coordinates fit within the [-1, +1] range – also called Normalized Device Coordinates (NDC). In order to get the vertices to this stage, thy must undergo a series of transformations, bringing them from the Local Object Space to the World Coordinates Space, on to the View Coordinates Space and finally to the Clipping Space.

### 3.1 Modeling transformations

When you define an object, vertex-by-vertex, you do so with respect to a local coordinates system. Most often, for convenience, you define your objects centered around this system's origin – this is standard practice for most object modeling tools. When placing the object within a given scene (world), you have to position it in the right place and maybe re-dimension it to fit inside the scene.

The series of transformations (translation, rotation, scaling) needed to position the model within a scene are called modeling transformations. They transform the vertices of the object from Local Space to World Space.

### 3.1.1 Translation

The translation transformation is used to move an object (point) by a given amount.

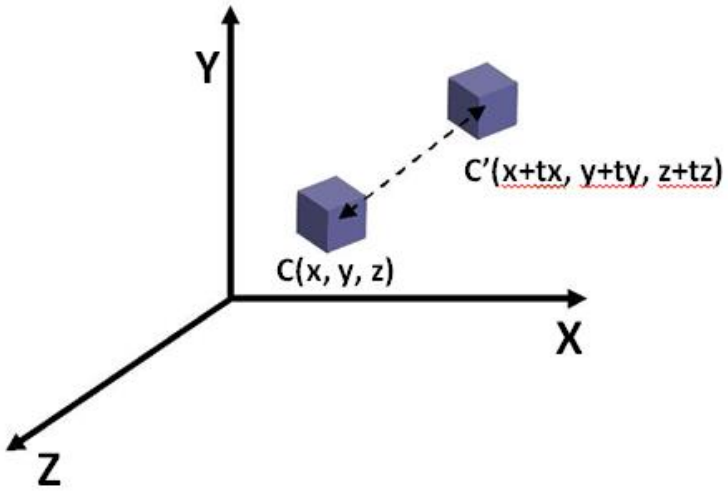
The translation matrix is:

$$T = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix for the inverse translation is:

$$T' = \begin{bmatrix} 1 & 0 & 0 & -Tx \\ 0 & 1 & 0 & -Ty \\ 0 & 0 & 1 & -Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tx, Ty an Tz represent the translation factors on x, y and z axes.




You can generate a translation matrix using:

```
glm::translate(glm::mat4 init_matrix, glm::vec3(float Tx, float Ty, float Tz))
```

### 3.1.2 Scale

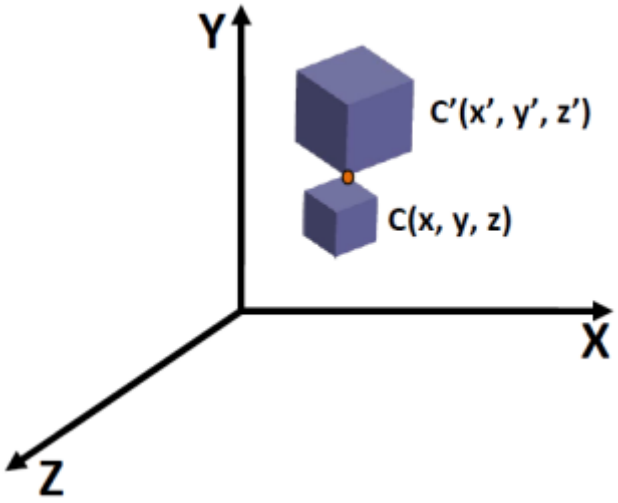
The scale transformation enlarges or reduces an object. The transformation is *relative to the origin*.

The matrix for the scale operation is the following:

$$S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix for the inverse transformation is the following:

$$S' = \begin{bmatrix} 1/Sx & 0 & 0 & 0 \\ 0 & 1/Sy & 0 & 0 \\ 0 & 0 & 1/Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$




Sx, Sy and Sz represent the scale factors on x, y and z axes. If the Sx, Sy and Sz factors are equal, then the scaling transformation is uniform. If the Sx, Sy and Sz factors are not equal, then the scaling transformation is non-uniform.

If you set the scaling factors to +/- 1, then you can reflect the original shape.

You can generate a scaling matrix using:

```
glm::scale(glm::mat4 init_matrix, glm::vec3(float Sx, float Sy, float Sz))
```

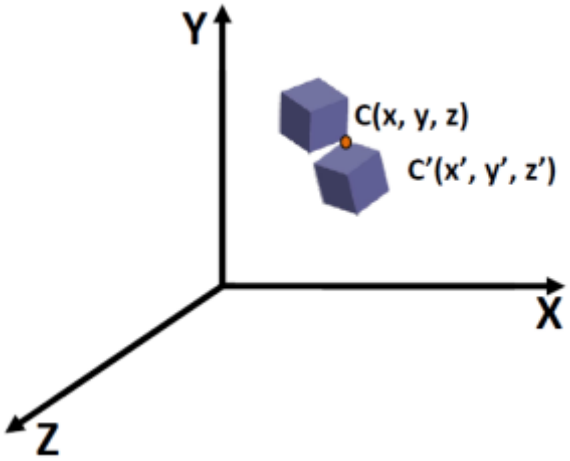
### 3.1.3 Rotation

This transformation rotates an object with a given angle. This transformation is also relative to the origin.

For a 2D point, the rotation and inverse rotation matrices are the following:

$$R = \begin{bmatrix} \cos\propto & -\sin\propto & 0 \\ \sin\propto & \cos\propto & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

In 3D space, we specify rotation independently on the x, y, and z axes. Rotation around the z axis is similar to the rotation in 2D (the z coordinate remains unchanged).




$$R_z = \begin{bmatrix} \cos\propto & -\sin\propto & 0 & 0 \\ \sin\propto & \cos\propto & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\propto & -\sin\propto & 0 \\ 0 & \sin\propto & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\propto & 0 & \sin\propto & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\propto & 0 & \cos\propto & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To change the rotation center – i.e. rotate around a different point instead of the origin, one must do a series of three transformations:

- Translate the new center point to the origin
- Perform the rotation
- Do the inverse translation transformation

Mathematically, this would mean: $\mathbf{P_{new} = T_{inverse} * R * T * P}$

You can generate a rotation matrix using:

```
glm::rotate(glm::mat4 init_matrix, float angle, glm::vec3(float x, float y, float z))
```

- x, y and z represent the 3D vector (axis) around which the rotation is performed
- the rotation angle is specified in degrees, not radians.

## 3.2 Camera transformations

After the scene has been set up, you need to define a viewer's position (camera) and viewing direction. This is done in order to render the scene from that viewer's perspective. Whereas until now the vertex coordinates were relative to the scene's origin, now they will be expressed relative to the camera's

position. We need to perform a series of rotations and translations to bring the scene's origin to the camera's position. This is called transforming from World Space to View Space.

The series of transformations needed in order to convert the scene to View Space are contained within one matrix, called camera or view matrix. It is built by taking into account the camera position, its orientation and a user specified "UP" vector.

The view matrix can be built using the following GLM function:

```
glm::lookAt(glm::vec3 camPosition, glm::vec3 viewPoint, glm::vec3 UP)
```

where:

- camPosition – a vec3 specifying the 3D coordinates of the camera position;
- viewPoint – a vec3 specifying the 3D coordinates of the point at which the camera is pointed. Note that the vector defined as viewPoint - camPosition specifies the viewing direction;
- UP – the "UP" vector – specifies the direction of the Y axis and is usually equal to (0,1,0);

## 3.3 Projection transformations

These transformations are required in order to create a 2D mapping from a portion of the 3D scene of objects defined within your project. The "portion" of the scene contributing to the obtained 2D representation is delimited by a viewing volume. This, in turn, is dependent upon parameters such as the viewer's position, field of view and the direction of the viewing vector. In the simplest of terms, you may think of a projection as the shadow that an object casts upon a 2D plane or surface.

While there are a number of different projection types, the most common are the perspective and orthographic projections.

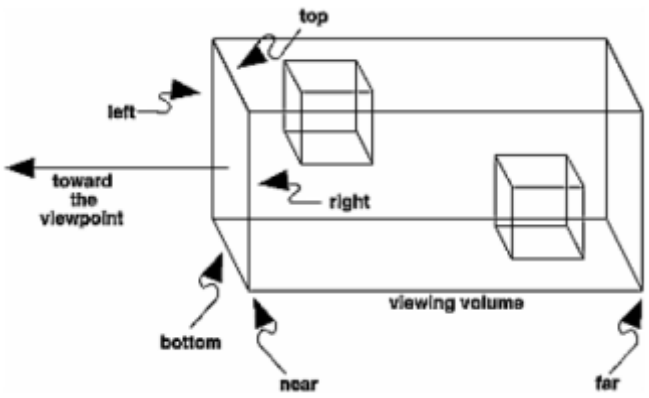
### 3.3.1 Orthographic projection

An orthographic projection transformation defines the viewing volume as a cube-like shape, using four parameters: width, height, a near and a far plane. Each 3D vertex will get mapped to a position determined by their X and Y coordinates.

You can create an orthographic projection matrix by using the GLM function `glm::ortho`, seen below:

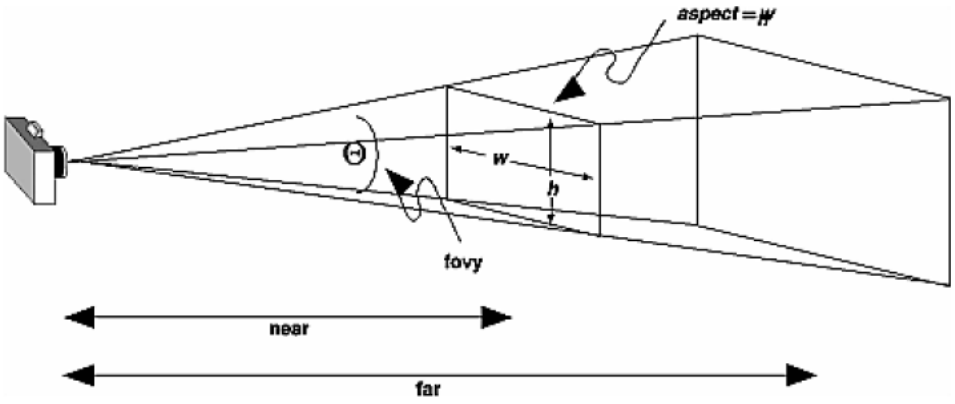
```
glm::ortho(GLint left, GLint right, GLint bottom, GLint top, GLfloat near, GLfloat far)
```

Each parameter of this function is a simple variable defining the X, Y or Z limits of the viewing volume.

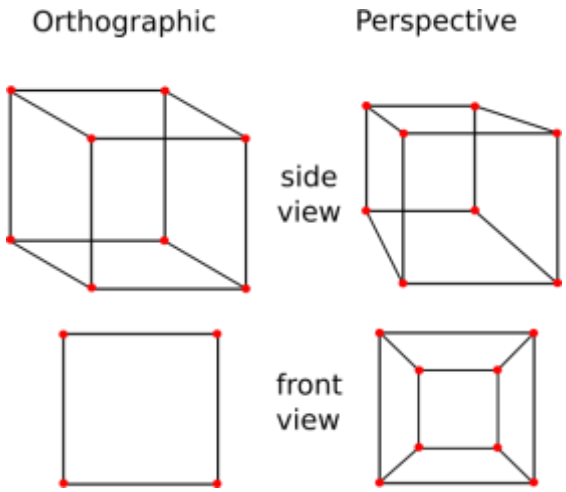



### 3.3.2 Perspective projection

The perspective transformation defines the viewing volume as frustum, starting from the camera position and expanding outwards, along the Z axis.




In this case, all three of the vertex's coordinates affect the 2D coordinates of its projection. The net result is that the further away an object is from the viewer, the smaller it will appear. This will give the viewer a sense of distance with regards to the objects in the scene.



You can create a perspective transformation matrix using the following GLM function:

```
glm::perspective(GLfloat fov, GLfloat aspect, GLfloat near, GLfloat far);
```

Where:

- fov – the angle that defines how large the viewing frustum is (usually 45 degrees)
- aspect – defines the aspect ratio, and is equal to the viewport's width divided by its height
- near & far – the Z coordinates of the near and far planes, which are the ends of the view frustum

# 4 Tutorial

We need to create the transformation matrices for each of the three steps:

- Model transformation
- View/camera transformation
- Projection transformation

After defining the matrices, we will use them to transform each and every vertex of the scene. To do this, we will apply the three transformations inside the Vertex Shader. After applying the transformations, the resulting vertex will look like this:

$$P_{final} = M_{projection} * M_{view} * M_{model} * P_{initial}$$

**Keep in mind that the order in which the transformations are applied to the vertex is the opposite of the order in which they appear in the formula above.** To get the correct order in which they are applied to the vertex, you should read the transformations for right to left.

## 4.1 Modeling transformations

Start by downloading the application from the laboratory website. Build and run the application.

You should see a purple square. This is in fact a multi-colored cube, seen from the front. In order to perceive it as a cube we will first add a rotation transformation to it.

As mentioned earlier, modeling transformations should be applied on the vertices inside the **Vertex Shader**. To this end, we will send it the modeling transformation matrix as a uniform variable:

```
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;

out vec3 colour;

uniform mat4 model;

void main() {
```

```
    colour = vertexColour;
    gl_Position = model * vec4(vertexPosition, 1.0);
}
```

We need to send the model matrix from the main program to the vertex shader. Declare the model matrix and its location as global variables:

```
GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;

gps::Shader myCustomShader;
glm::mat4 model;
GLint  modelLoc;
```

Create and send the model matrix:

```
int main(int argc, const char * argv[]) {

                . . . . . .

        model = glm::mat4(1.0f);
        modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        while (!glfwWindowShouldClose(glWindow)) {
                renderScene();

                glfwPollEvents();
                glfwSwapBuffers(glWindow);
        }
        //close GL context and any other GLFW resources
        glfwTerminate();

        return 0;
}
```

Let us add a rotation transform inside “renderScene()”. This will activate on pressing the “R” key:

```
float angle=0;
void renderScene()
{
        . . . . . .

        model = glm::mat4(1.0f);

        if (glfwGetKey(glWindow, GLFW_KEY_R)) {
            angle += 0.0002f;
        }
```

```
        // create rotation matrix
        model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

Try out the rotation by pressing the "R" key.

## 4.2 Camera and projection transformations

The next step is to apply the view and projection transformations on our scene of objects. This will allow for a more realistic and controllable representation of the scene.

Start by adding two more matrices inside the vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;

out vec3 colour;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
   colour = vertexColour;
   gl_Position = projection * view * model * vec4(vertexPosition, 1.0);
}
```

We now have to define, initialize and send the two matrices to the vertex shader. Add the following:

```
int main(int argc, const char * argv[]) {

        . . . .

        model = glm::mat4(1.0f);
        modelLoc = glGetUniformLocation(myCustomShader.shaderProgram, "model");
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        // initialize the view matrix
        glm::mat4 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 5.0f), glm::vec3(0.0f, 0.0f, -10.0f), glm::vec3(0.0f, 1.0f,
0.0f));
        // send matrix data to shader
```

```
        GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

    // initialize the projection matrix
        glm::mat4 projection = glm::perspective(70.0f, (float)glWindowWidth / (float)glWindowHeight, 0.1f, 1000.0f);
        // send matrix data to shader
        GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));

        . . . .
}
```

## 4.3 Adding transformations

Let us draw a second cube and place it 2 units to the left of the first one. By default, an object will always be drawn centered at the origin. To avoid this, we perform a translation before drawing the second cube:

```
void renderScene()
{
        . . . .

        // create rotation matrix
        model = glm::rotate(model, angle, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);

        // create a translation matrix
        model = glm::translate(model, glm::vec3(2, 0, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        // draw the second cube
        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

The cube will still be drawn at the origin, but the translation transformation has just moved that origin two units to the right.

Do a rotation by holding the “R” key. Notice how the transformation applies to the whole scene.

Now, let us apply a rotation on the second cube:

```
void renderScene()
```

```
{
        . . . .

        // create rotation matrix
        model = glm::rotate(model, 0.3f, glm::vec3(0, 1, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));

        // create a translation matrix
        model = glm::translate(model, glm::vec3(2, 0, 0));
        // send matrix data to vertex shader
        glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
        // draw the second cube
        myCustomShader.useShaderProgram();
        glBindVertexArray(objectVAO);
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
}
```

# 5 Assignment

1. Control the rotation of the second cube with the “T” key.
2. Modify the rotation of the second cube by making it spin around its center – the point (2, 0, 0).
   * Tip: Use the sequence of operations $\mathbf{T_{inverse}}$ *** R * T**
3. Implement the Camera class (start from the Camera.hpp present on the web site). Control the camera using keyboard and mouse inputs. **Hint:** Register some callback functions to process keyboard and mouse inputs:

```
void keyboardCallback(GLFWwindow* window, int key, int scancode, int action, int mode);
void mouseCallback(GLFWwindow* window, double xpos, double ypos);

...

glfwSetKeyCallback(glWindow, keyboardCallback);
glfwSetCursorPosCallback(glWindow, mouseCallback);
glfwSetInputMode(glWindow, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```