Politecnico di Torino

1859

VR @POLITO

GRAINS

GRAphics and INtelligent Systems group

Laboratorio 08

OpenGL

Part III

Outline

- Transformations
- Coordinate Systems
- Camera

Transformations

Until now static objects have been used. A possible solution to make them move could be changing their vertices and re-configuring their buffers in each frame. However, this solution is cumbersome and costs quite some processing power. There are much better ways to transform an object based on multiple matrix objects.

Transformations

OpenGL does not have any form of matrix or vector knowledge built in. Luckily, there is an easy-to-use and tailored-for-OpenGL mathematics library called GLM.

GLM stands for OpenGL Mathematics and is a header-only library, which means that there is only the need to include the proper header files; no linking and compiling is necessary.

GLM can be downloaded from https://glm.g-truc.net/0.9.8/index.html

To use the library, the root directory of the header files has to be copied on your includes folder.

Transformations

Most of GLM's functionality needed for this course can be found in 3 headers files that can be included as follows:

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

Transformations

The following code can be used to translate a vector (1,0,0) by (1,1,0)

```
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;
```

A vector named `vec` is first defined by using GLM's built-in vector class. Then a matrix$_{4x4}$ (`mat4 trans`) is defined and explicitly initialized to the identity matrix by initializing the matrix's diagonals to 1.0; if the matrix is not initialized to the identity matrix, it would be a null matrix (all elements 0) and all subsequent matrix operations would end up a null matrix as well.

Transformations

The following code can be used to translate a vector (1,0,0) by (1,1,0)

```
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;
```

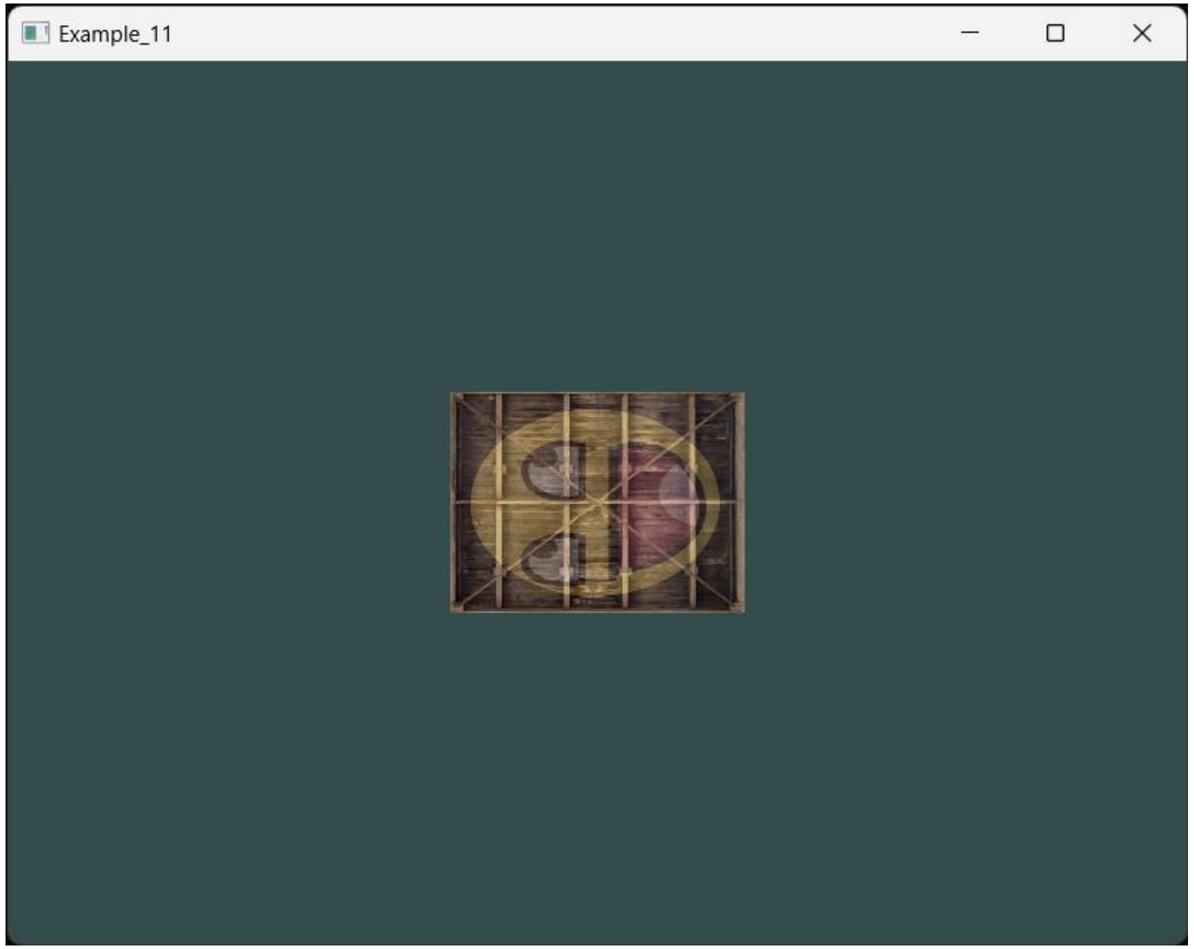
A transformation matrix is than created by passing the identity matrix to the `glm::translate` function, together with a translation vector (the given matrix is then multiplied with a translation matrix and the resulting matrix is returned).

Finally, the vector is multiplied by the transformation matrix and the result is printed in the console.

Apply transformations

Example11

This example shows how to apply transformations to the container object from Example 10



Apply transformations

Define the transformations

The following code can be used to scale and rotate the object

```
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 the object is scaled by 0.5 on each axis and then rotated by 90 degrees around the Z-axis. GLM expects to receive angles expressed in radians. For this reason, the conversion from degrees to radians is performed by using `glm::radians`.

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

Apply transformations

Updating the vertex shader

To get the transformation matrix to the shaders, the vertex shader should be adapted to accept a `mat4` uniform variable and to 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);
}
```

Apply transformations

Pass the transformation matrix to the vertex shader

To pass the transformation matrix to the shader:

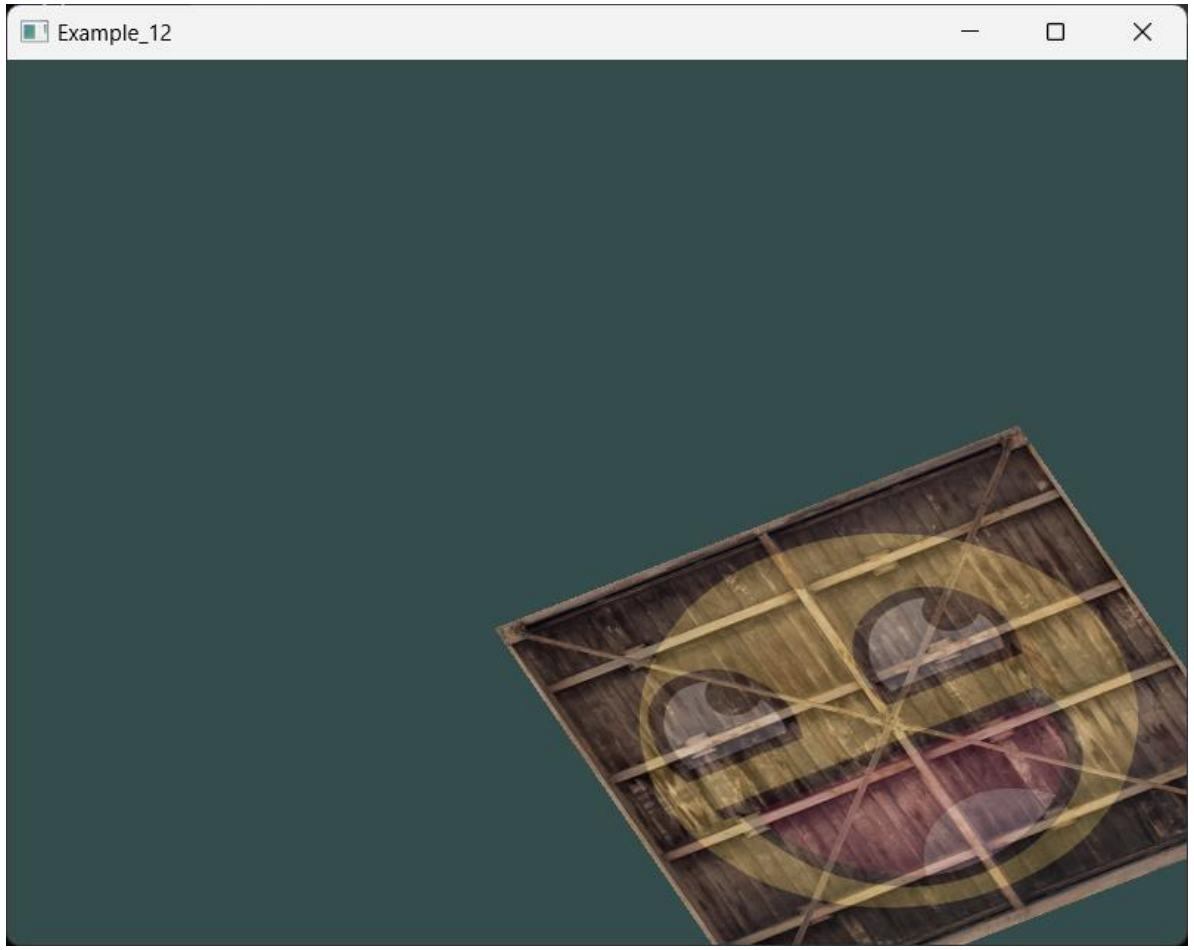
```
unsigned int transformLoc = glGetUniformLocation(ourShader.ID,
"transform");
glUniformMatrix4fv(transformLoc, 1, GL_FALSE, glm::value_ptr(trans));
```

First the location of the uniform variable is queried, then the matrix data are sent to the shaders using `glUniform` with `Matrix4fv` as its postfix.

Create an animation

Example12

This example shows how to rotate the object over time thus creating a simple animation.



Create an animation

Create transformations

To rotate the container over time, it is necessary to update the transformation matrix in the render loop, in this way, it is updated at each frame. The GLFW's time function can be used to get an angle over time.

```
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));
```

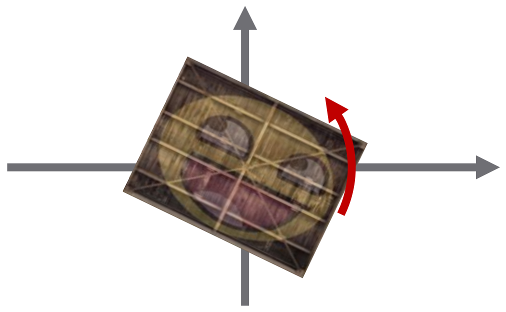
In this example, the object is first rotated around the origin (0,0,0) and once it's rotated, the rotated version of the object is translated to the bottom-right corner of the screen. The actual transformation order should be read in reverse with respect to the code.

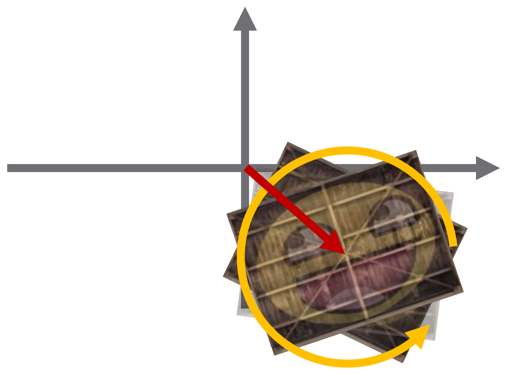
Create an animation

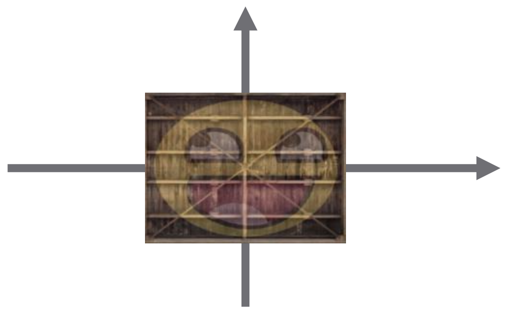
Create transformations

The order in which the transformations are applied leads to different results.

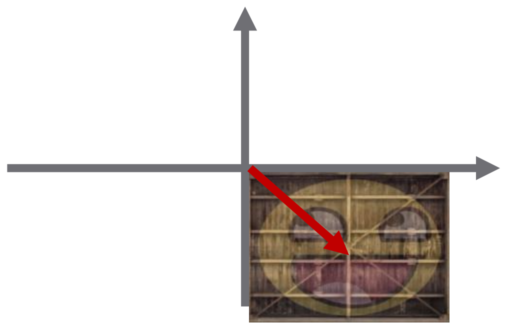
```
glm::translate(…);
glm::rotate(…);
```

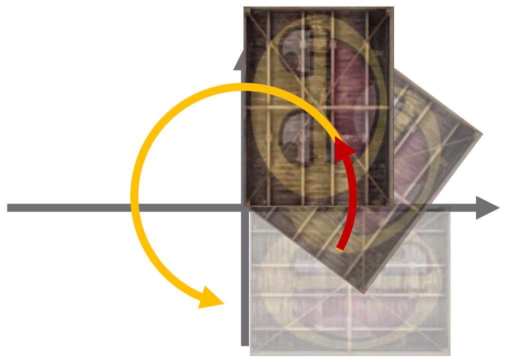






```
glm::rotate(…);
glm::translate(…);
```





Exercise #5

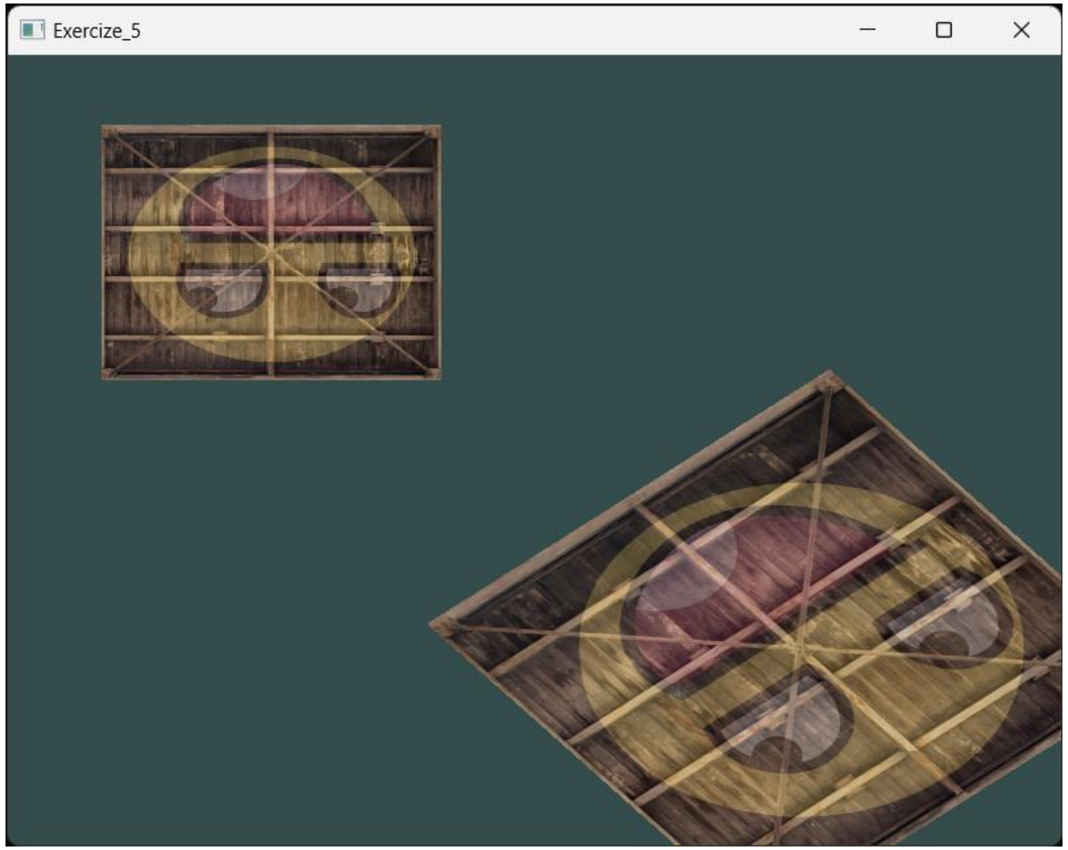
Draw a second object with another call to `glDrawElements`. Place the second object at the top-left of the window using transformations only. Make sure the second object is scaled over time (rather than rotated) using a sin function.

- Before defining the second transformations, remember to reset the matrix to an identity matrix

```
transform = glm::mat4(1.0f);
```

- Note that using sin function will cause the object to invert as soon as a negative scale is applied. The scaling factor can be computed as follow:

```
float scaleAmount = static_cast<float>(sin(glfwGetTime()));
```



Exercise #5

Instructions:

- Before defining the second transformations, remember to reset the matrix to an identity matrix

```
transform = glm::mat4(1.0f);
```

- Note that using sin function will cause the object to invert as soon as a negative scale is applied. The scaling factor can be computed as follow:

```
float scaleAmount = static_cast<float>(sin(glfwGetTime()));
```

Coordinate Systems

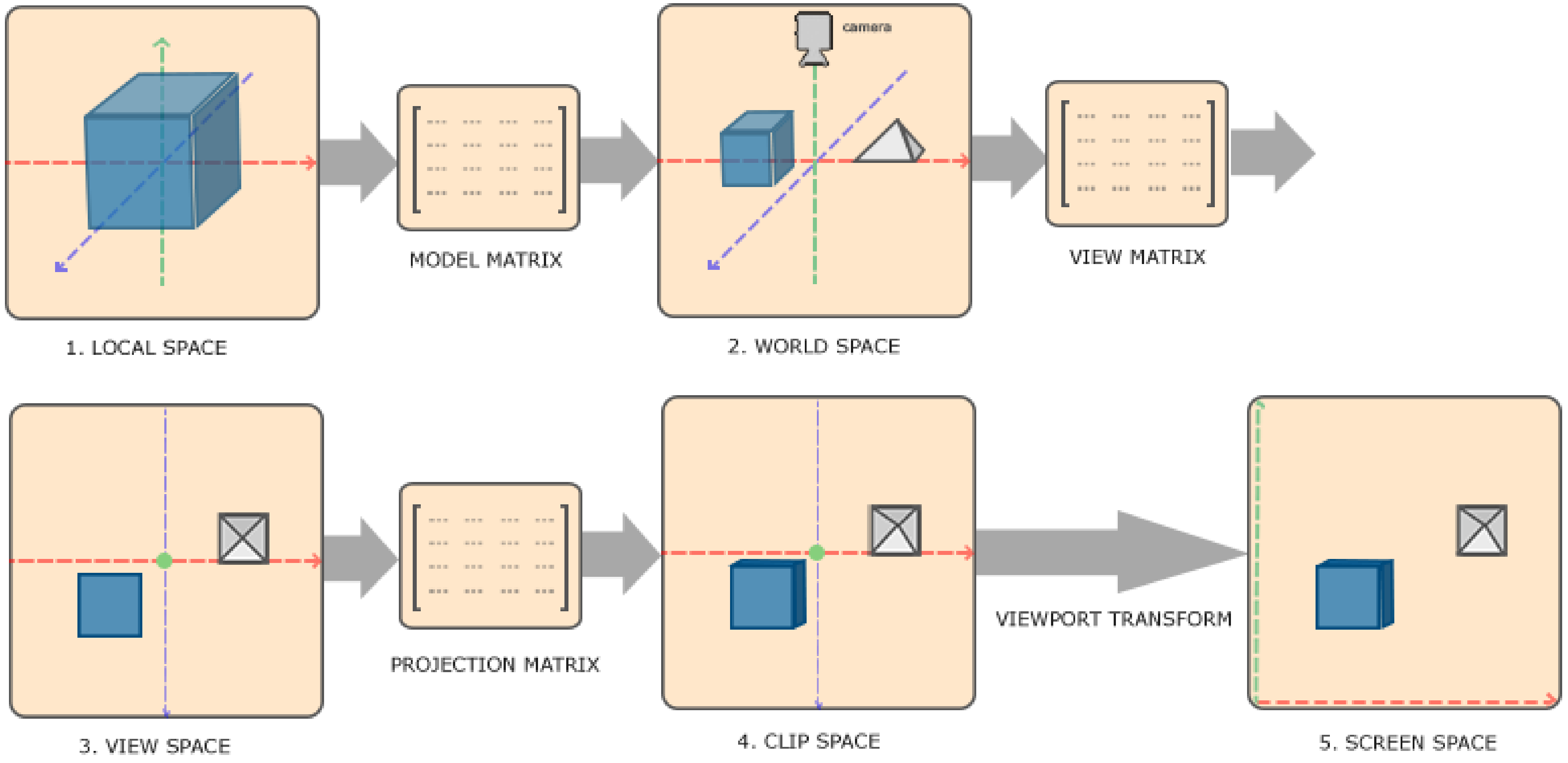
OpenGL expects all the vertices to be in normalized device coordinates after each vertex shader run, to make them visible in the screen. That is, the x, y and z coordinates of each vertex should be between -1.0 and 1.0; coordinates outside this range will not be visible. Till now vertices have been specified in a specific range (or space) determined by ourselves, then the vertex shader transforms these coordinates to normalized device coordinates (NDC). These NDC are then given to the rasterizer to transform them to 2D coordinates/pixels on your screen.

Coordinate Systems

Transforming coordinates to NDC is usually accomplished in a step-by-step fashion where object's vertices are transformed to several coordinate systems before finally transforming them to NDC. The advantage of transforming them to several intermediate coordinate systems is that some operations/calculations are easier in certain coordinate systems. There are a total of 5 different coordinate systems to be considered:

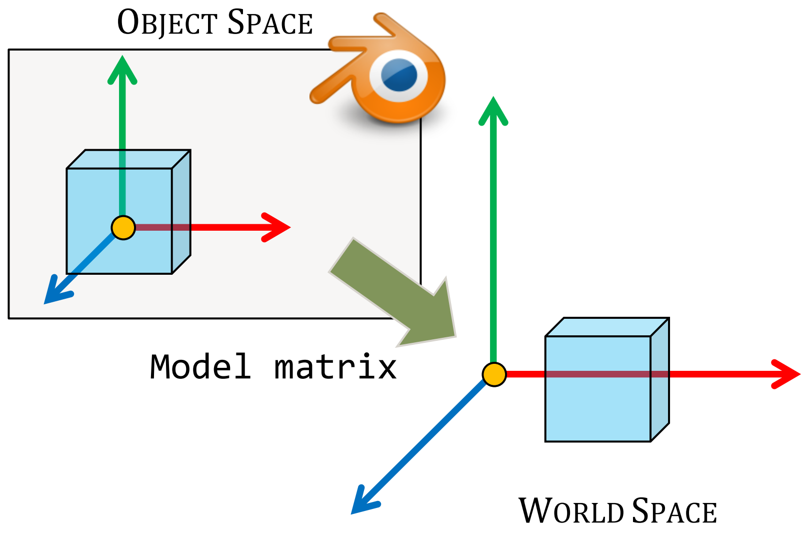
- Local space (or Object space)
- World space
- View space (or Eye space)
- Clip space
- Screen space

Coordinate Systems



World Transform

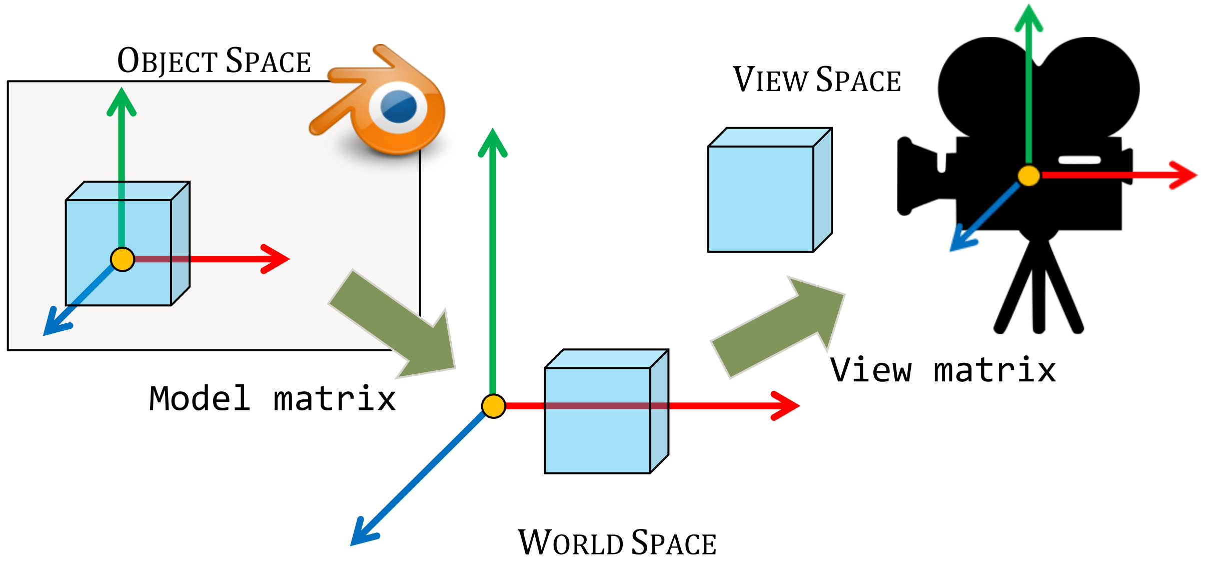
World Transform (also referred to as object transform) is defined by the model matrix, It allows you to convert the vertices (and normals) of the models from the local space (e.g., the coordinate space used in the tool in which the object was modeled) to the world space (world coordinate system).



View Trasform

View Transform converts the vertices of an object from world-space to view-space. It is defined by the view matrix, calculated as the inverse matrix of the transformation matrix used to position and orient the camera in the world.

Commonly, the first column represents the Right (X) vector, the second is associated to the Up (Y) vector, the third to the Forward (Z) vector, while the fourth is the translation vector of the space represented by the transformation matrix.



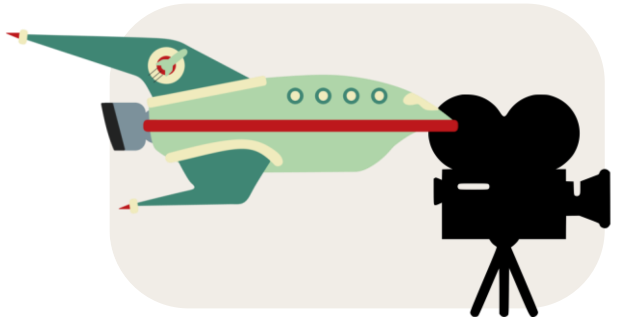
$$\begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View Trasform

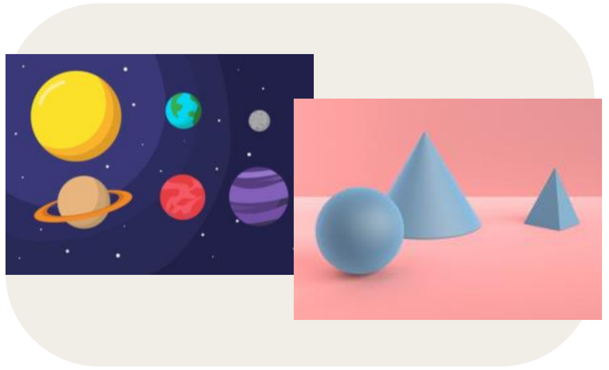
How the engine of the Futurama Planet Express shuttle works?

"The engines don't move the ship. The ship remains stationary where it is: the engines move the universe around it. " (Prof. Farnsworth)

https://www.youtube.com/watch?v=1RtMMupdOC4

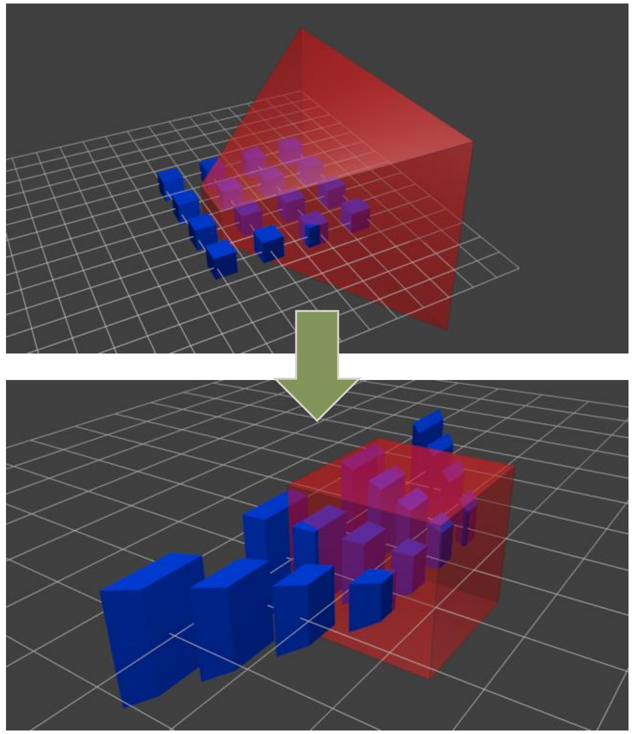


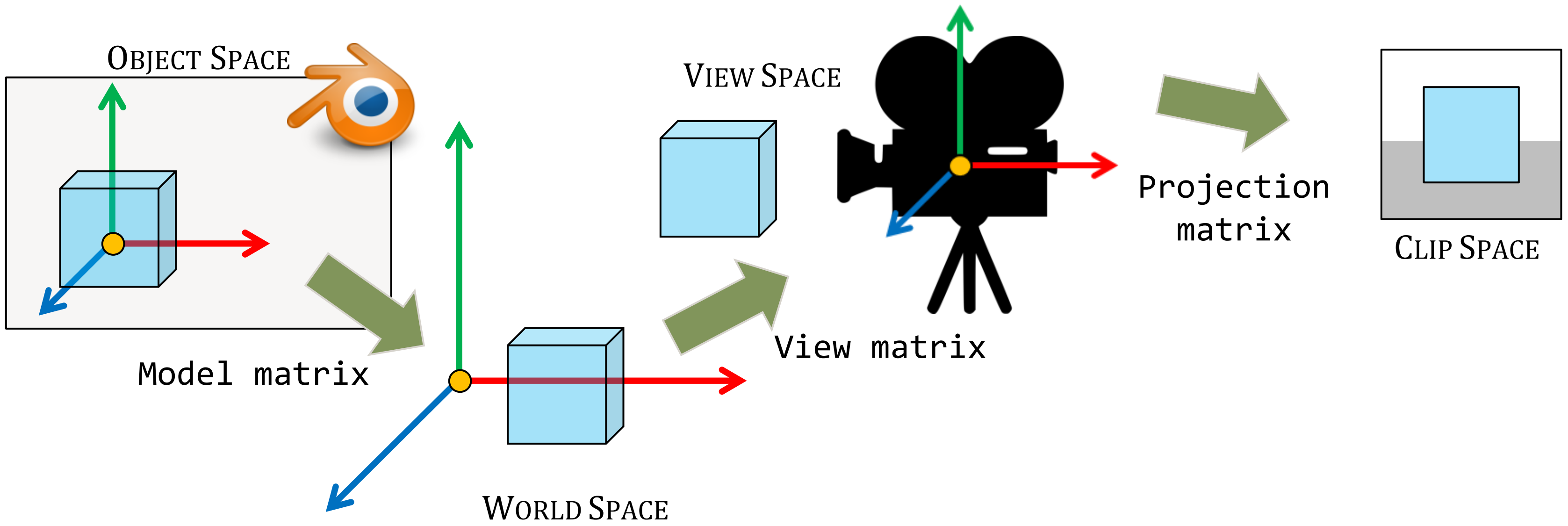
The same principle may be applied to computer graphics. If you want to observe the scene from another point of view, practically, the camera is not moved and oriented but a transformation (described by the view matrix) is applied to all the objects in the scene so that they appear in front of the camera in the same how they would look if the camera had actually been moved.



Projection Transform

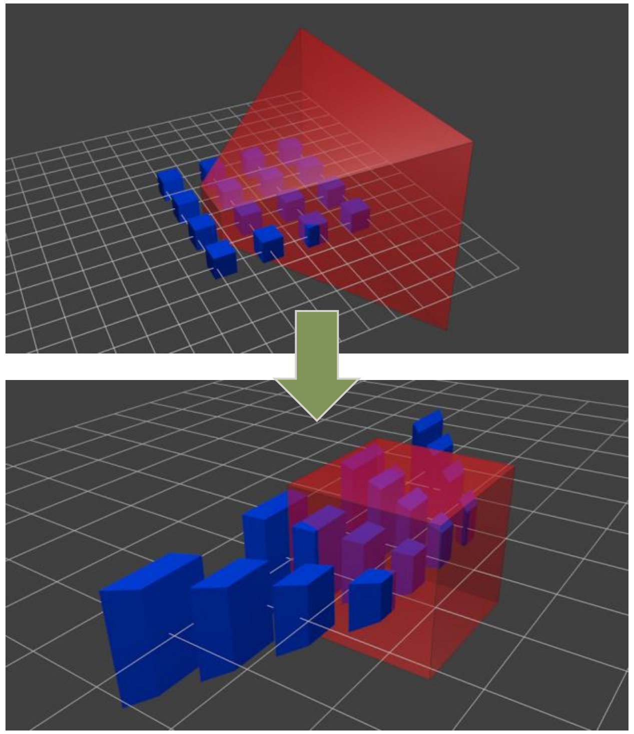
Projection Transform converts from view-space coordinates to clip-space (window coordinate system), using the projection matrix. Multiplying by this projection matrix, the frustum of the camera becomes a cube and the objects are "deformed" according to the characteristics of the camera.

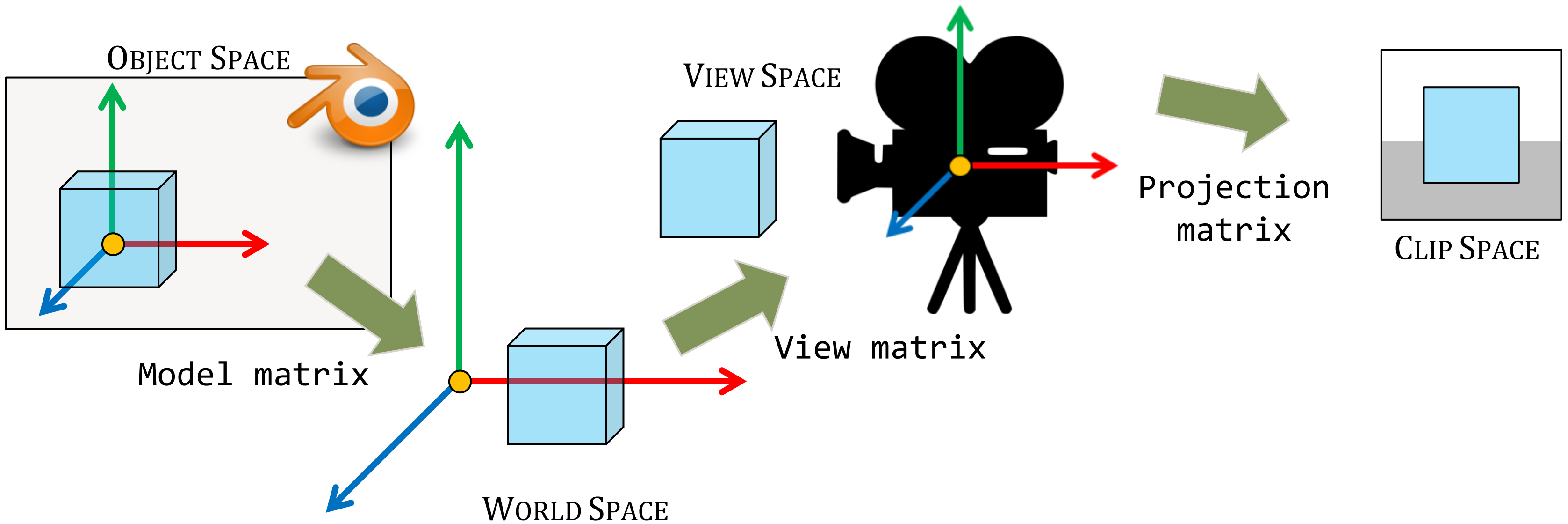




Projection Transform

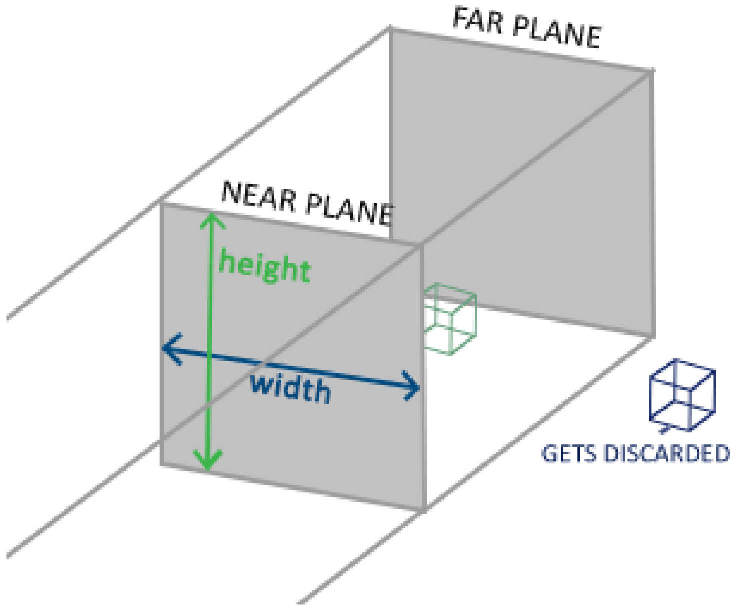
Once all the vertices are transformed to clip space a final operation called perspective division is performed in which the x, y and z components of the position vectors are divided by the vector's homogeneous w component. Perspective division is what transforms the 4D clip space coordinates to 3D normalized device coordinates.



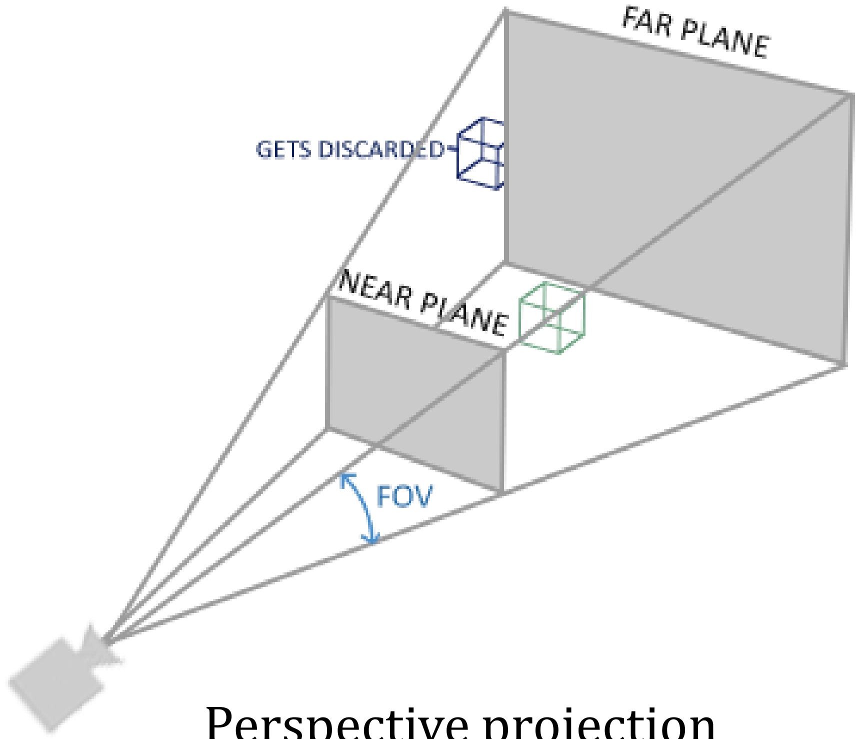


Projection Transform

The projection matrix to transform view coordinates to clip coordinates usually takes two different forms, where each form defines its own unique frustum: orthographic projection matrix or perspective projection matrix.

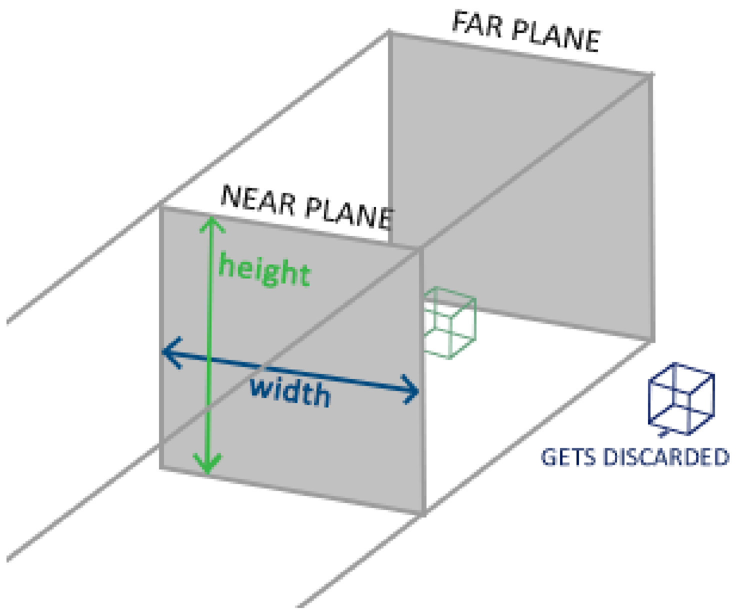


Orthographic projection



Perspective projection

Orthographic projection



An orthographic projection matrix defines a cube-like frustum box. The frustum defines the visible coordinates and is specified by a width, a height and a near and far plane.

To create an orthographic projection matrix it is possible to use the GLM's built-in function `glm::ortho`:

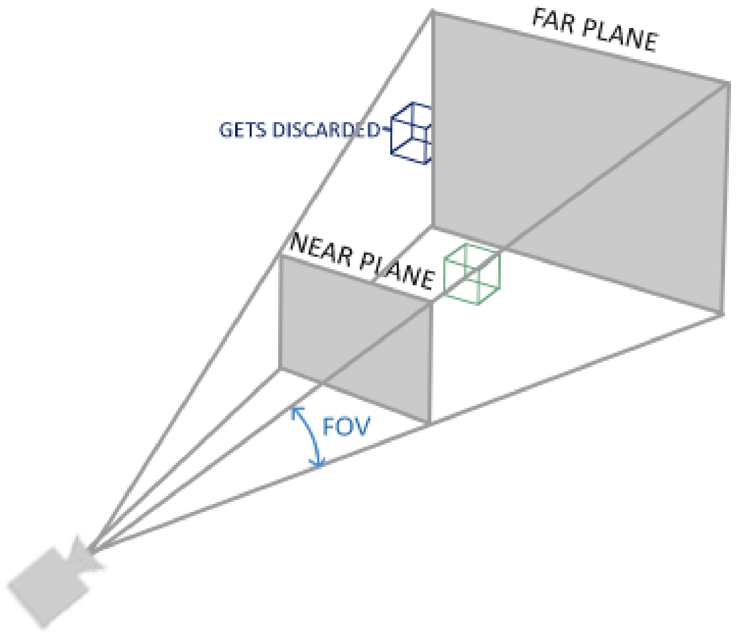
```
float aspect = (float)SCR_WIDTH / SCR_HEIGHT;
projection = glm::ortho(-aspect, aspect, -1.0f, 1.0f, 0.1f, 100.0f);
```

The first two parameters specify the left and right coordinate of the frustum (i.e., width). The third and fourth parameter specify the bottom and top part of the frustum (height). The 5th and 6th parameter then define the distances between the near and far plane.

Perspective projection

The projection matrix maps a given frustum range to clip space, but also manipulates the w value of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes. Once the coordinates are in clip space, perspective division is applied to the clip space coordinates by dividing each component of the vertex coordinate by its w component giving smaller vertex coordinates the further away a vertex is from the viewer.:

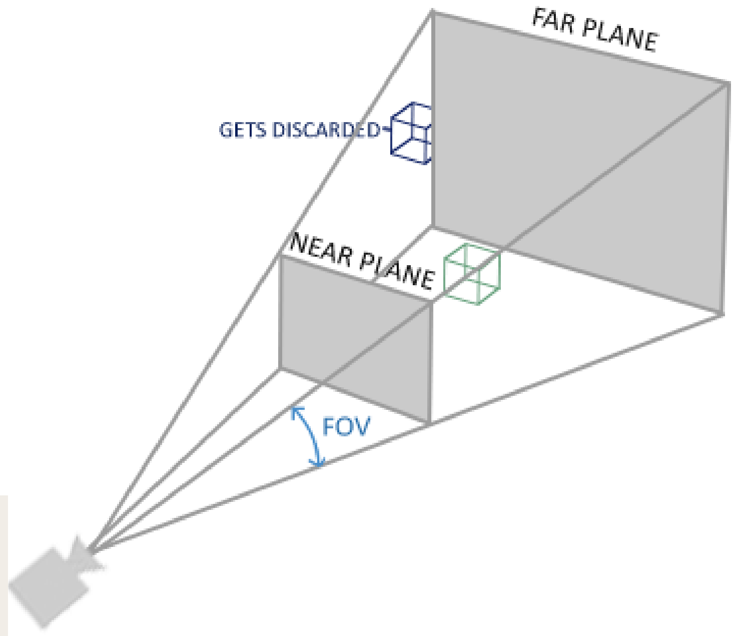
$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$



Perspective projection

A perspective projection matrix can be created in GLM using a GLM's built-in function:

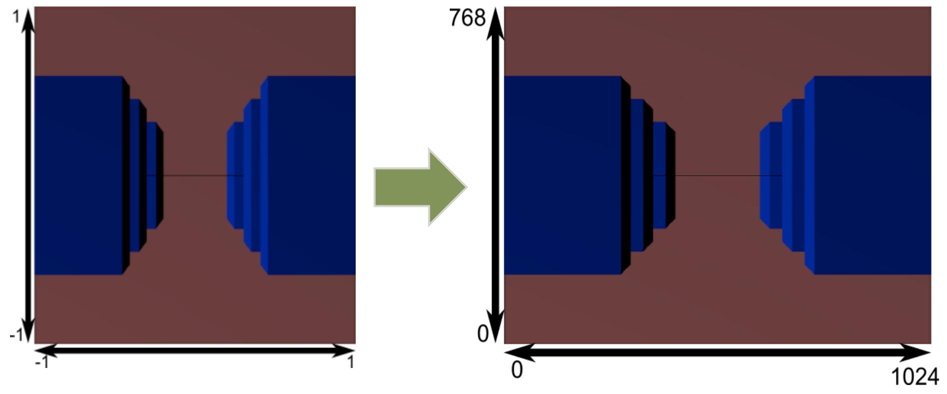
```
glm::mat4 proj =
glm::perspective(glm::radians(45.0f),
(float)width / (float)height, 0.1f, 100.0f);
```

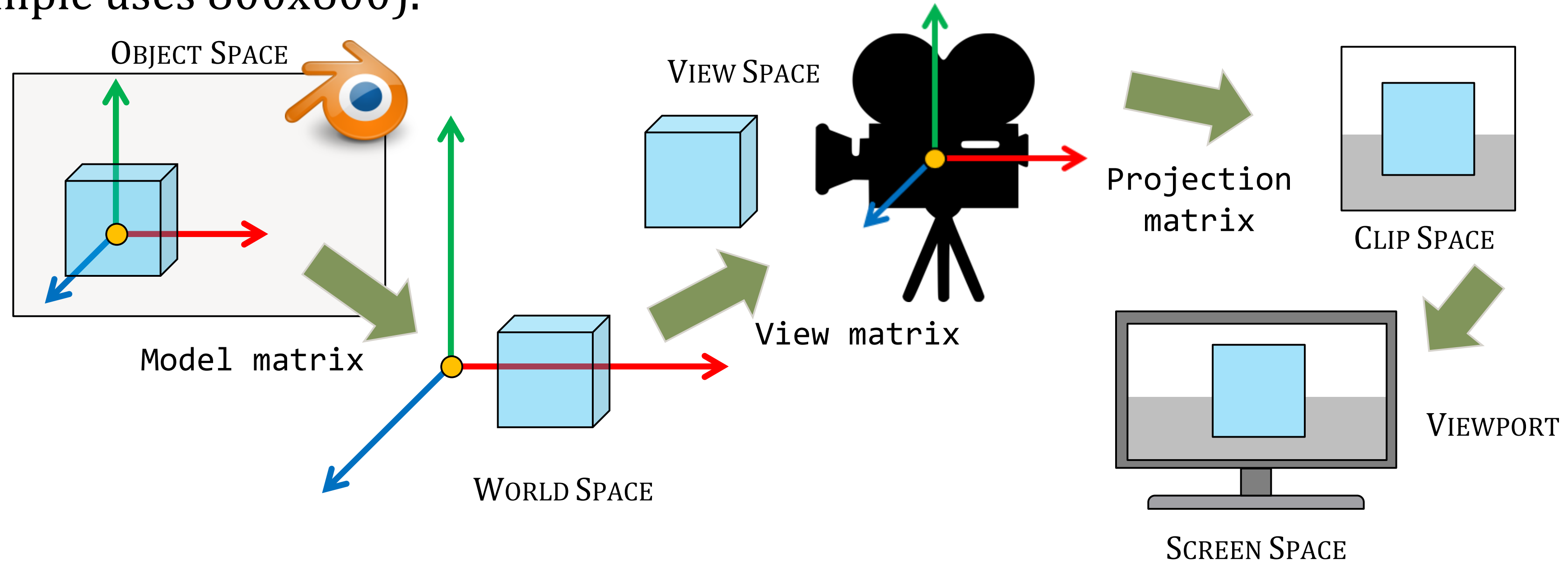


The first parameter defines the fov value, that stands for field of view and sets how large the viewspace is. The second parameter sets the aspect ratio which is calculated by dividing the viewport's width by its height. The third and fourth parameter set the near and far plane of the frustum.

Viewport Transform

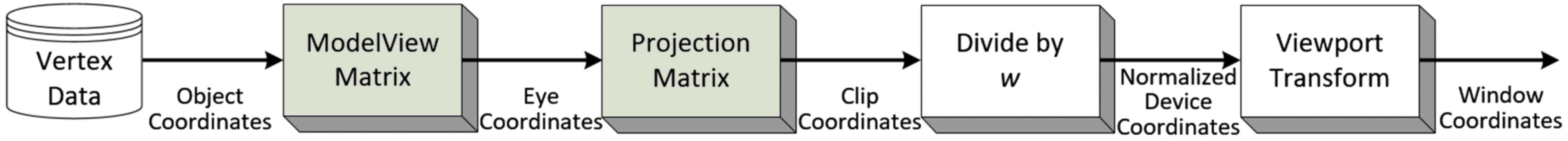
Viewport Transform is applied to adapt the image to the screen. More specifically, OpenGL uses the parameters from `glViewPort` to map the normalized-device coordinates to screen coordinates where each coordinate corresponds to a point on the screen (e.g. the previous example uses 800x600).





Overview

A transformation matrix is created for each of the aforementioned steps: model, view and projection matrix.



A vertex coordinate is then transformed to clip coordinates as follows:

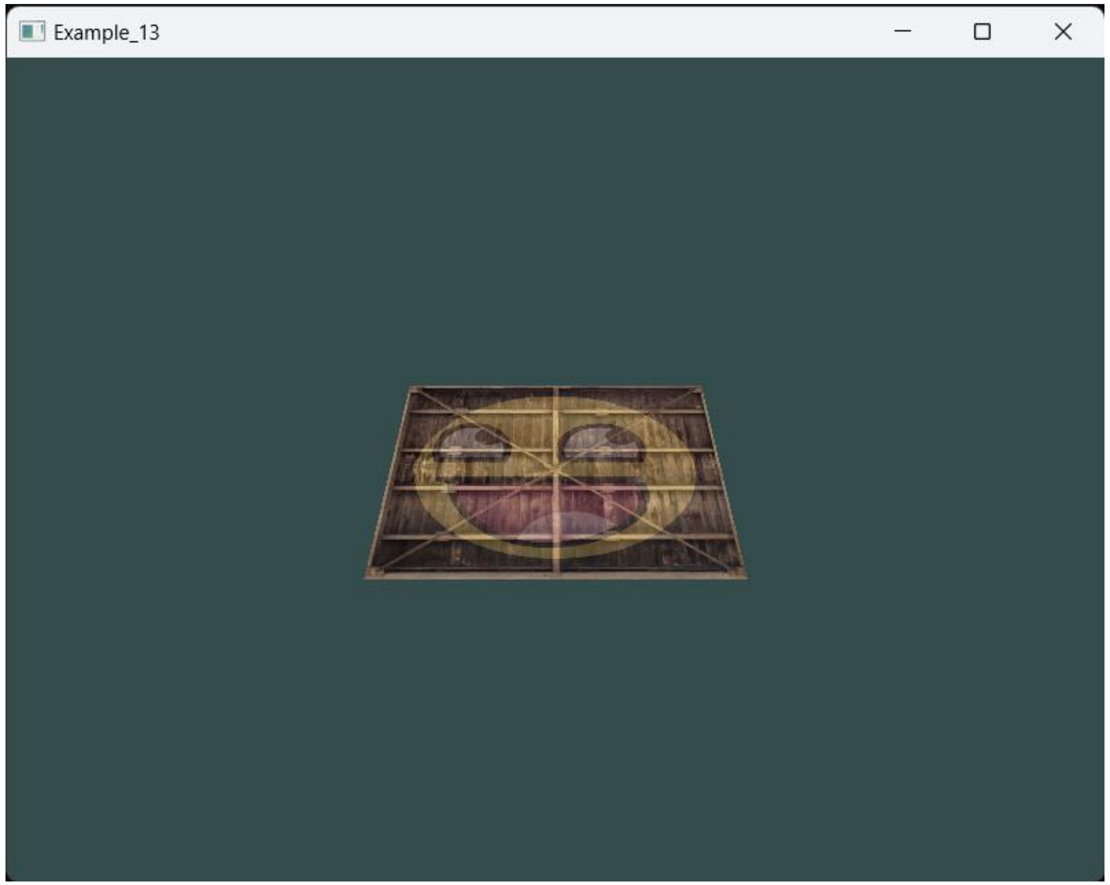
$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

The resulting vertex should then be assigned to `gl_Position` in the vertex shader and OpenGL will then automatically perform perspective division and clipping.

Coordinate Systems

Example13

To start drawing in 3D, a model, view and projection matrices have to be created and passed to the vertex shader.



Coordinate Systems

Model matrix

The model matrix consists of translations, scaling and/or rotations to be applied to transform all object's vertices to the global world space.

To transform the plane used in the previous examples by rotating it on the x-axis so it looks like it's laying on the floor, the model matrix looks like this:

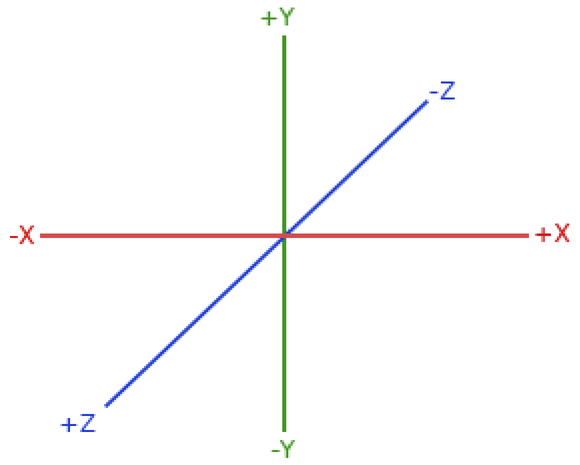
```
glm::mat4 model = glm::mat4(1.0f);
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

By multiplying the vertex coordinates with this model matrix, the vertex coordinates are transformed to world coordinates.

Coordinate Systems

View matrix

Then a view matrix is created. To move a camera backwards (so the object become visible as by default the camera is located at the world origin (0,0,0)), is the same as moving the entire scene forward. That is exactly what a view matrix does, the entire scene is moved around inversed to where the camera has to be moved.



Because the camera has to be moved backwards and since OpenGL is a right-handed system, the camera is moved in the positive z-axis. To this aim the whole scene is translated towards the negative z-axis (this gives the impression that we are moving backwards). Therefore, the view matrix looks like this:

```
glm::mat4 view = glm::mat4(1.0f);
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

Coordinate Systems

Projection matrix

The last thing to define is the projection matrix. To use the perspective projection, it is possible to declare a projection matrix like this:

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
```

Coordinate Systems

Using matrices in the vertex shader

After creating the transformation matrices, they have to be passed to the shaders.

First let's declare the transformation matrices as uniforms in the vertex shader and multiply them with the vertex coordinates:

```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ...
}
```

Note that the multiplication is read from right to left

Coordinate Systems

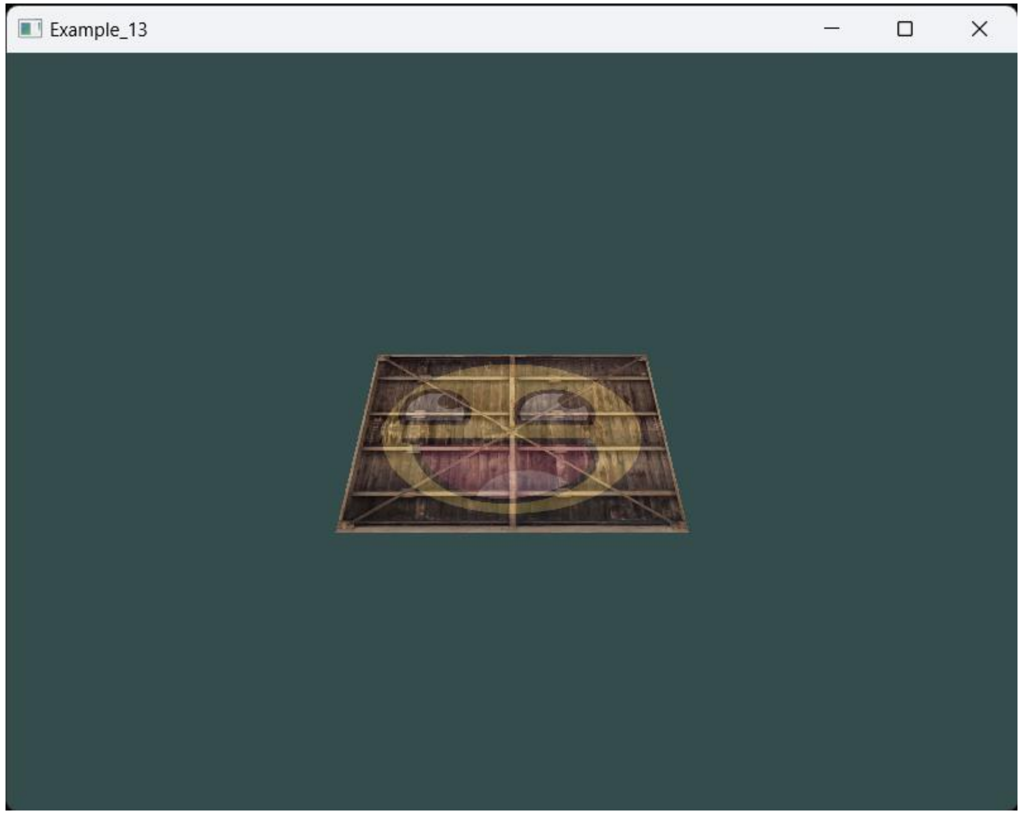
Sending matrices to the vertex shader

Sending the matrices to the shader is usually done each frame (in the render loop) since transformation matrices tend to change.

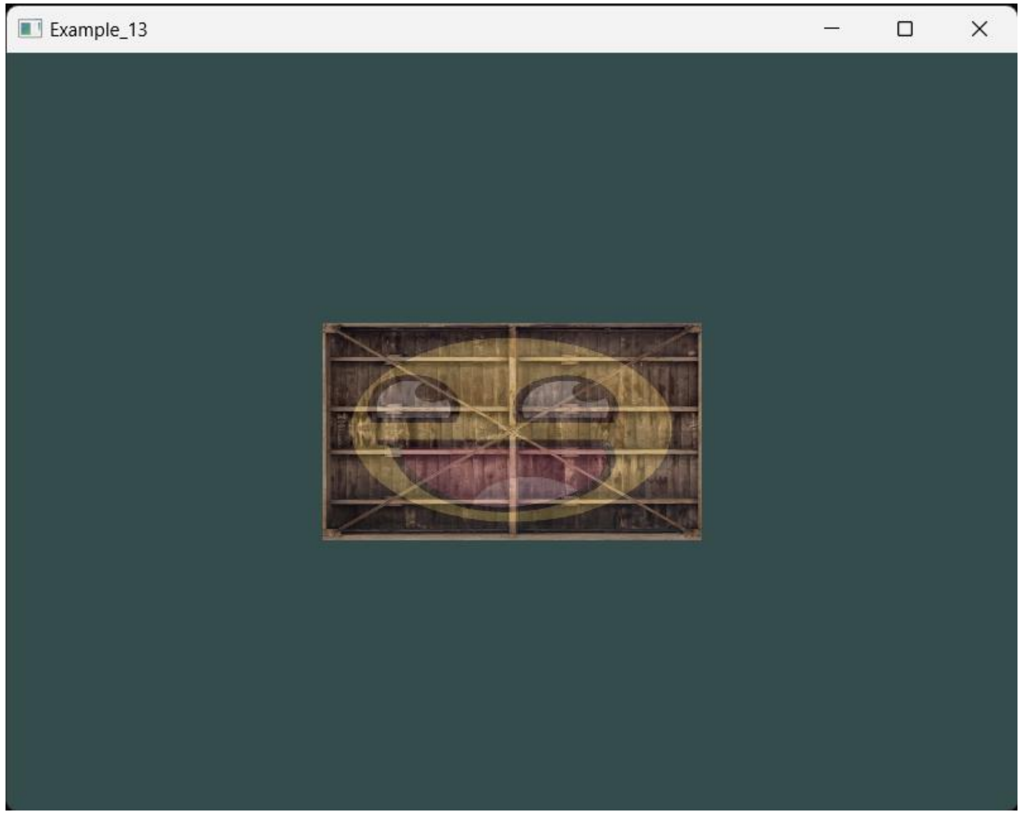
```
// retrieve the matrix uniform locations
unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");
unsigned int viewLoc = glGetUniformLocation(ourShader.ID, "view");

// pass them to the shaders (3 different ways)
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
ourShader.setMat4("projection", projection);
```

Coordinate Systems

Sending matrices to the vertex shader



`projection = glm::perspective()`

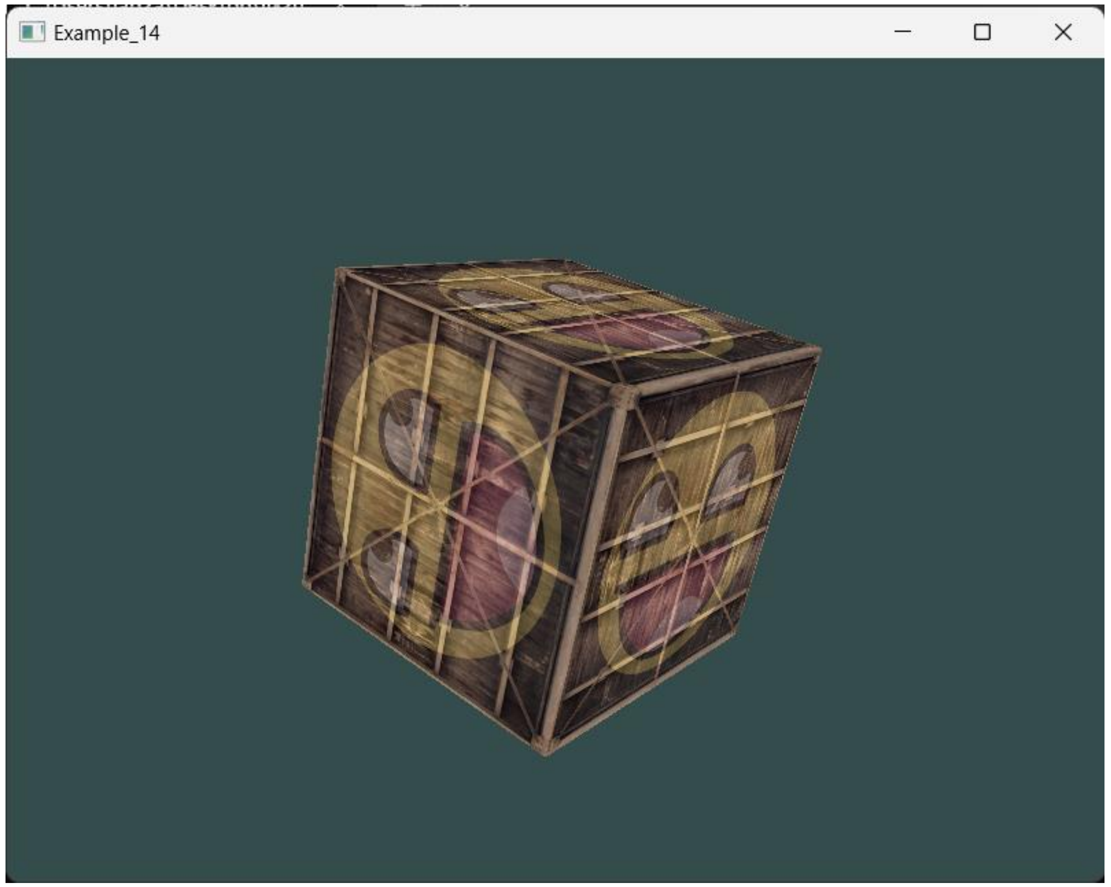


`projection = glm::ortho()`

Drawing a 3D cube

Example14

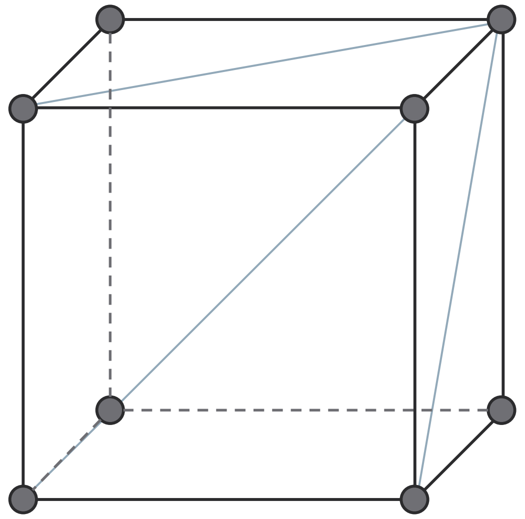
So far, the examples have show how to draw in 3D but using a simple 2D plane. Now, let's try to draw a 3D cube.



Drawing a 3D cube

Defining vertices

To render a cube a total of 36 vertices (6 faces * 2 triangles * 3 vertices each) are needed.



```
float vertices[] = {
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 0.0f,

    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,

    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
     0.5f, -0.5f, -0.5f,  1.0f, 1.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
     0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,

    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
     0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
     0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
    -0.5f,  0.5f,  0.5f,  0.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,  0.0f, 1.0f
};
```

Drawing a 3D cube

Defining matrices and draw function

It is possible to animate the cube, by applying a rotation through the model matrix,

```
model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
```

For what it concern the other matrices, the view and projection matrices defined in the previous example can be used.

In this example, as no indices have been defined, it is needed to use the `glDrawArrays` function with a count of 36 vertices.

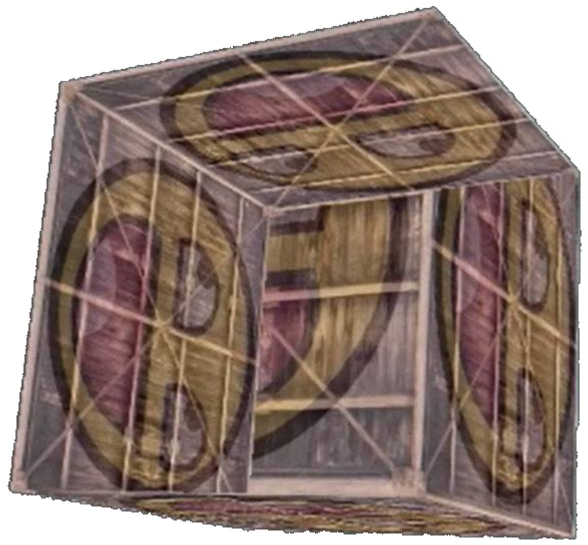
```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

Drawing a 3D cube

Z-buffer

By default, OpenGL draws the cube triangle-by-triangle, fragment by fragment, thus overwrite any pixel color that may have already been drawn there before. This results in some sides of the cubes that are being drawn over other sides of the cube. Since OpenGL gives no guarantee on the order of triangles rendered (within the same draw call), some triangles are drawn on top of each other even though one should clearly be in front of the other.

To fix this issue, it is possible to look at depth information stored in a buffer called **z-buffer** that allows OpenGL to decide when to draw over a pixel and when not to.



Drawing a 3D cube

Z-buffer

The z-buffer, also known as a **depth buffer** is automatically created by GLFW (just like the color-buffer that stores the colors of the output image). The depth is stored within each fragment (as the fragment's z value) and whenever the fragment wants to output its color, OpenGL compares its depth values with the z-buffer. If the current fragment is behind the other fragment it is discarded, otherwise overwritten. This process is called **depth testing** and is done automatically by OpenGL.

To make sure OpenGL actually performs the depth testing, it has to be enabled (it is disabled by default).

Drawing a 3D cube

Z-buffer

The depth testing can be enabled using the `glEnable` function. Generally, the `glEnable` and `glDisable` functions are used to enable/disable certain functionality in OpenGL (not only the depth testing). That functionality is then enabled/disabled until another call is made to disable/enable it. The depth testing is enabled using the `GL_DEPTH_TEST` parameter.

```
glEnable(GL_DEPTH_TEST);
```

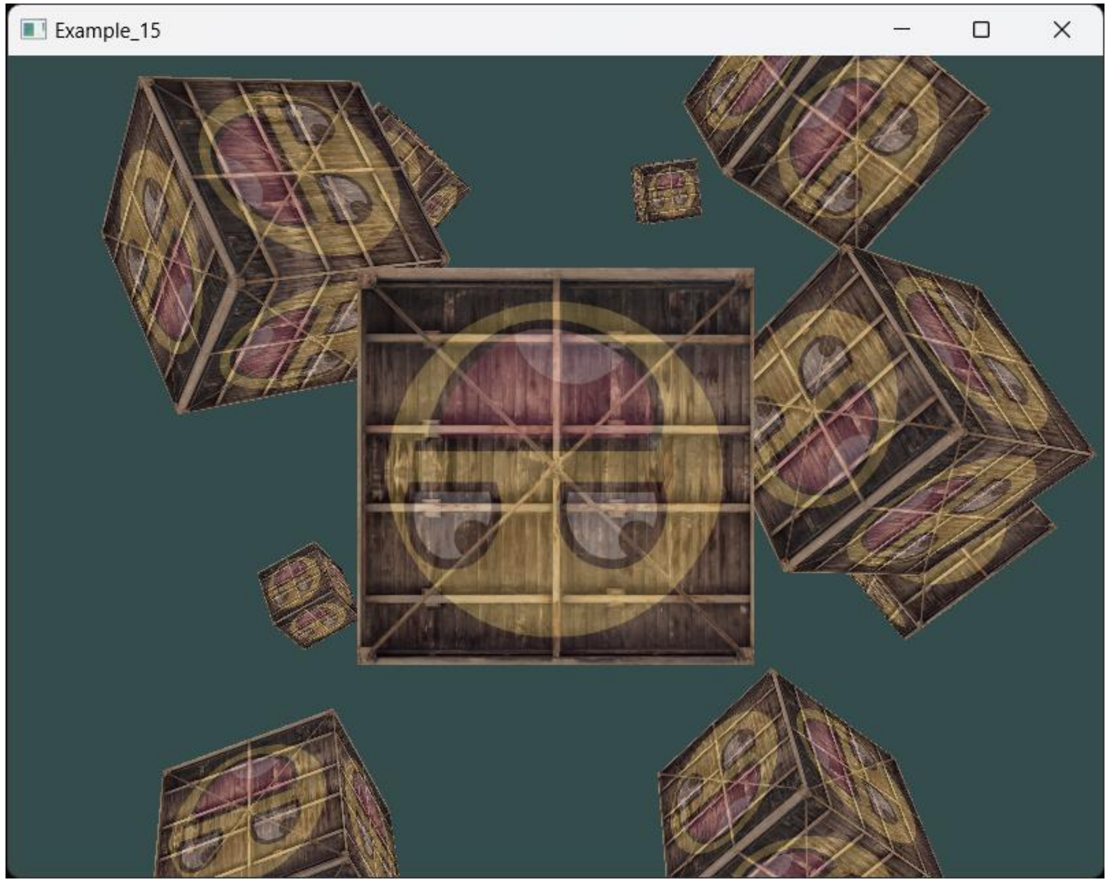
Since this example uses a depth buffer, it is needed to clear the depth buffer before each render iteration (otherwise the depth information of the previous frame stays in the buffer). Like clearing the color buffer, the depth buffer can be cleared by specifying the `DEPTH_BUFFER_BIT` bit in the `glClear` function:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Drawing more cubes

Example15

The aim of this example is to display 10 cubes on the screen, that will look the same but will only differ in where the cube is located in the world and rotated.



Drawing more cubes

Update model matrix

The graphical layout of the cube is already defined so it is not needed to change the buffers or attribute arrays when rendering more objects. The only thing to do is to change for each object its model matrix, to transform the cubes into the world.

To this aim, a translation vector for each cube is defined. Each element of the array specifies the position of a given cube in world space. The 10 cube positions are defined using a `glm::vec3` array:

```
glm::vec3 cubePositions[] = {
    glm::vec3(0.0f,  0.0f,  0.0f),
    glm::vec3(2.0f,  5.0f, -15.0f),
    glm::vec3(-1.5f, -2.2f, -2.5f),
    glm::vec3(-3.8f, -2.0f, -12.3f),
    glm::vec3(2.4f, -0.4f, -3.5f),
    glm::vec3(-1.7f,  3.0f, -7.5f),
    glm::vec3(1.3f, -2.0f, -2.5f),
    glm::vec3(1.5f,  2.0f, -2.5f),
    glm::vec3(1.5f,  0.2f, -1.5f),
    glm::vec3(-1.3f,  1.0f, -1.5f)
};
```

Drawing more cubes

Update model matrix

Now, within the render loop, the `glDrawArrays` is called 10 times by sending, for each cube, a different model matrix in the vertex shader. A small loop is created, within the render loop, that renders our object 10 times with a different model matrix each time. A small unique rotation is also added to each cube.

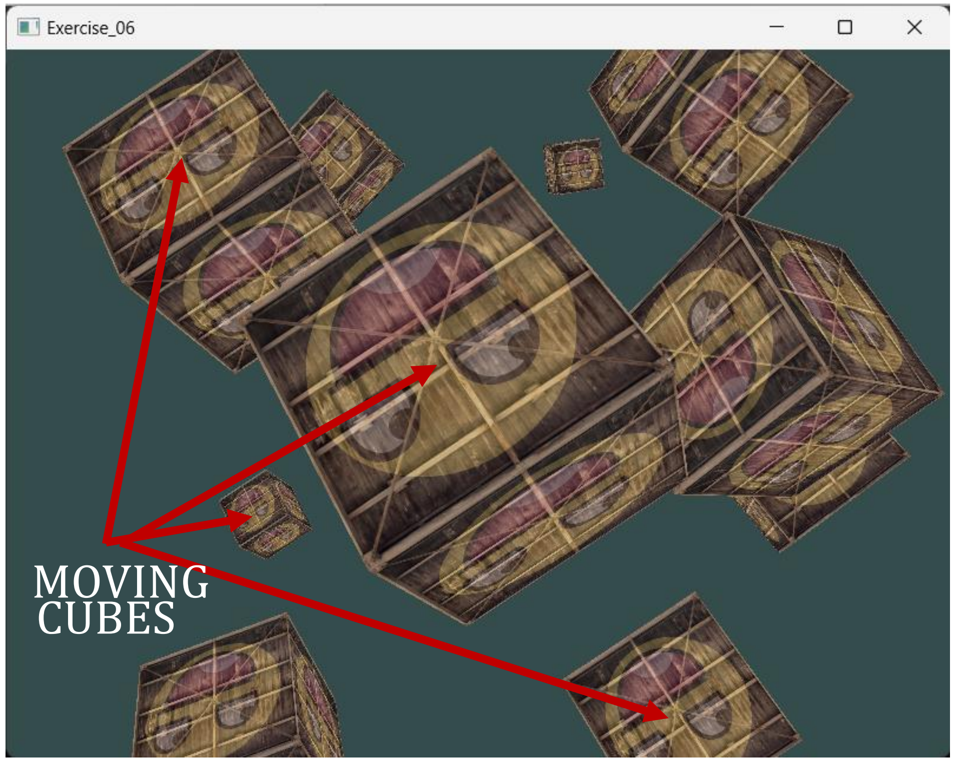
```
glBindVertexArray(VAO);
for (unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Exercise #6

Make every 3rd object (including the 1st) rotate over time, while leaving the other object static using only the model matrix.

- Starting from Example 15, introduce addition check to apply rotation only to given cubes.
- To apply rotation over time, you can use the following code:

```
angle = glfwGetTime() * 25.0f;
```

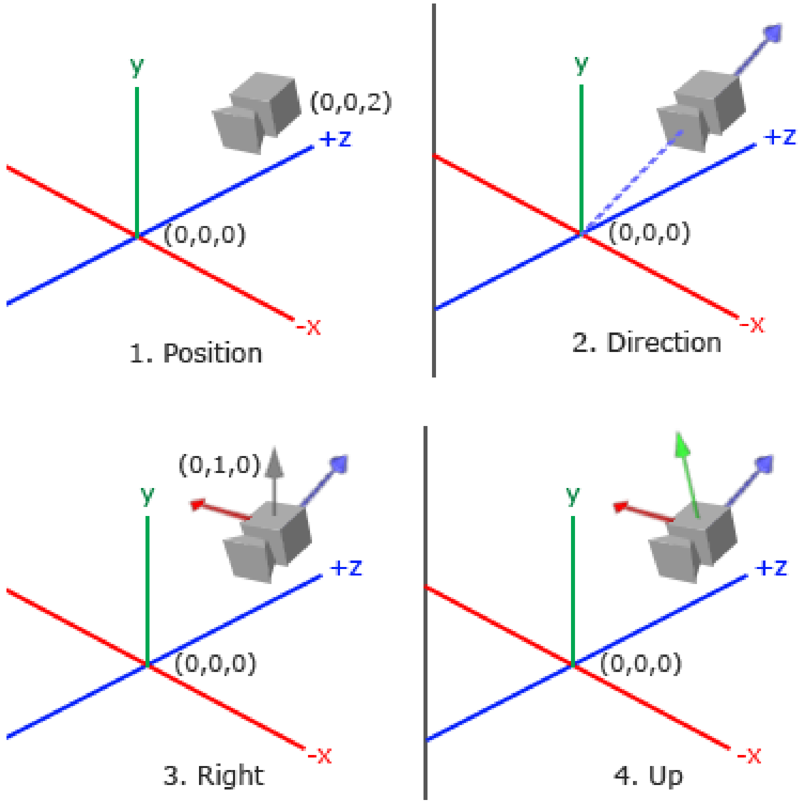


Camera

The camera/view space can be regarded as the space in which vertex coordinates are transformed as they are seen from the camera's perspective. In this respect, the camera represents the origin of the scene. The view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction.

To define a camera the following elements are needed: its **position** in world space, the **direction** it's looking at, a vector pointing to the **right** and a vector pointing **upwards** from the camera.

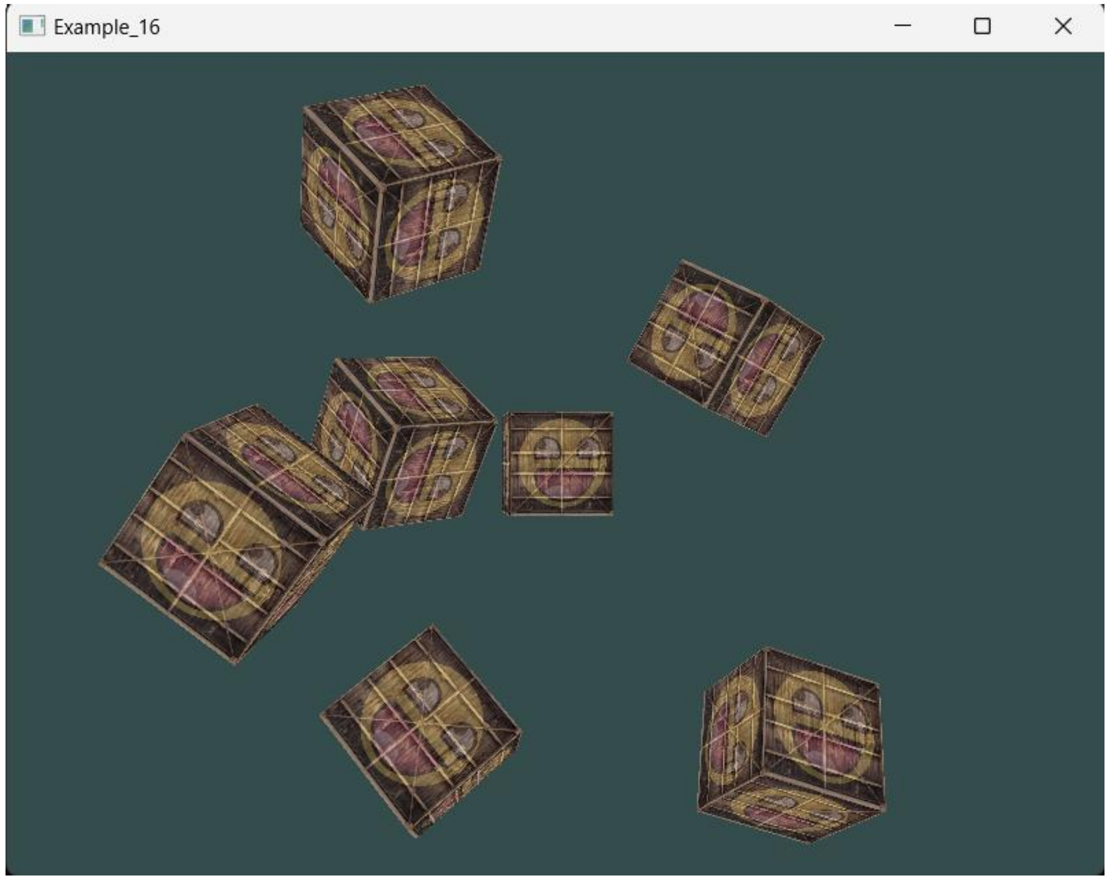
In other words, a new coordinate system is created with 3 perpendicular unit axes with the camera's position as the origin.



Configuring a camera

Example16

The example shows how to configure the camera used to frame the OpenGL 3D scene, by leveraging the `LookAt` function.



Configuring a camera
LookAt matrix

If a coordinate space is defined by using 3 perpendicular (or non-linear) axes , it is possible to create a matrix with those 3 axes plus a translation vector. Any vector can be transformed to that coordinate space by multiplying it with the computed matrix. This is exactly what the LookAt matrix does.

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R is the right vector, U is the up vector, D is the direction vector and P is the camera's position vector.

Configuring a camera

LookAt matrix

It is worth noticing that the rotation (left matrix) and translation (right matrix) parts are inverted (transposed and negated respectively) since the entire world has to be rotated and translated in the opposite direction of where the camera should be.

Using this `LookAt` matrix as a view matrix effectively transforms all the world coordinates to the view space just defined.

In other words, the `LookAt` matrix creates a view matrix that looks at a given target.

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Configuring a camera

LookAt matrix

Using GLM it is possible to create a LookAt matrix (to be used as view matrix) by specifying a camera position, a target position and a vector that represents the up vector in world space.

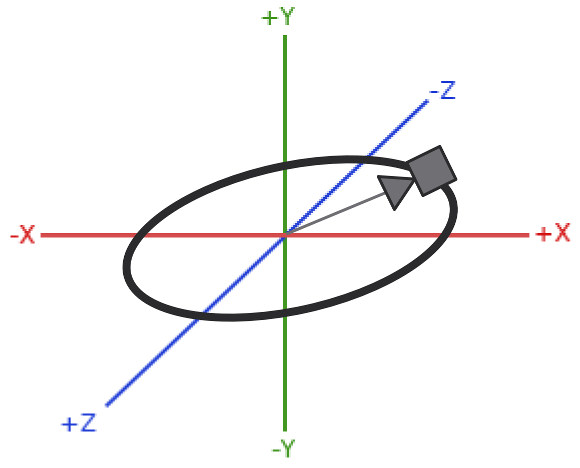
```
glm::mat4 view;
view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```

Configuring a camera

Dynamic LookAt matrix

To make the camera rotate around the scene, the target is kept to (0,0,0). A little bit of trigonometry is used to create an x and z coordinate for each frame that represents a point on a circle used to move the camera.

By re-calculating the coordinate over time, the camera traverses all the points in a circle, thus resulting in a rotating camera. A predefined radius is used to enlarge this circle and the `glfwGetTime` function is called at each frame to create a new matrix.

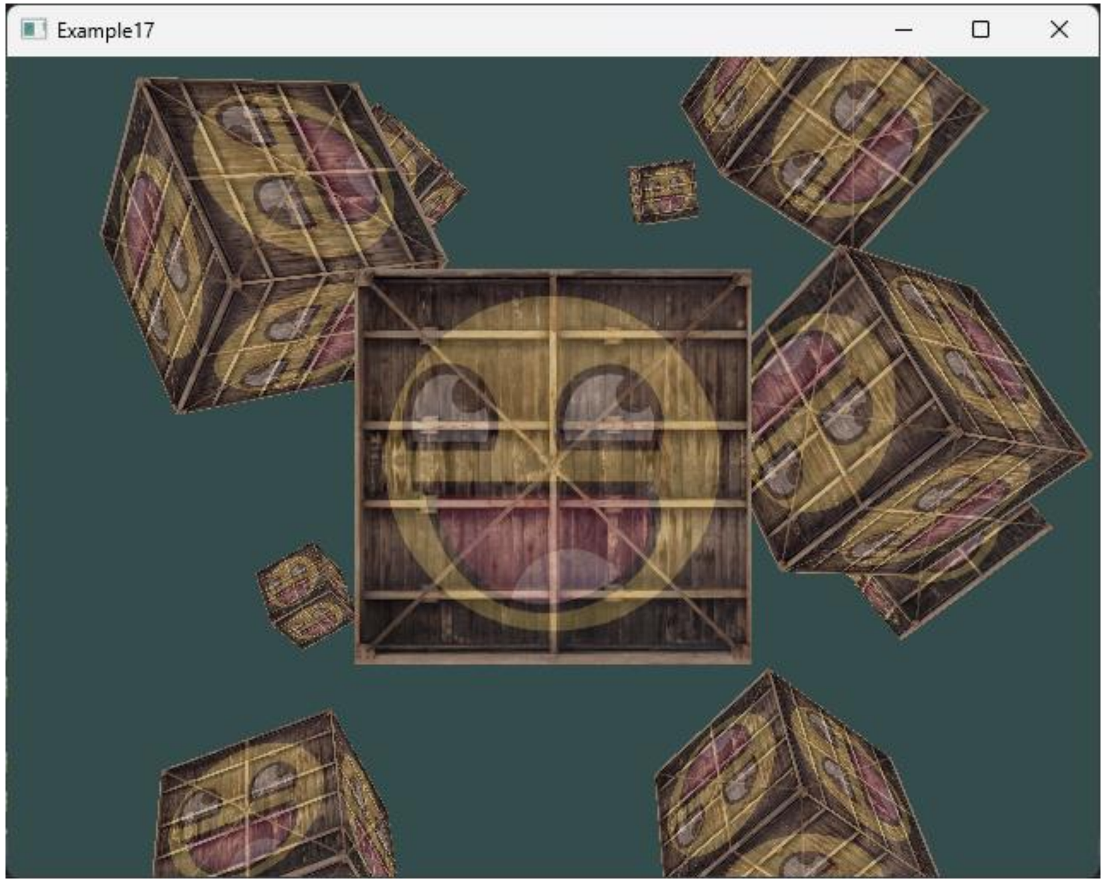


```
float radius = 10.0f;
float camX = static_cast<float>(sin(glfwGetTime()) * radius);
float camZ = static_cast<float>(cos(glfwGetTime()) * radius);
view = glm::lookAt(glm::vec3(camX, 0.0f, camZ),
                   glm::vec3(0.0f, 0.0f, 0.0f),
                   glm::vec3(0.0f, 1.0f, 0.0f));
```

Walk around

Example17

The example shows how to configure the camera to implement a walk around of the 3D scene.



Walk around

Set up the camera system

To set up the moving camera, a number of camera variable have to be defined at the top of the program.

```
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
```

The LookAt function now becomes:

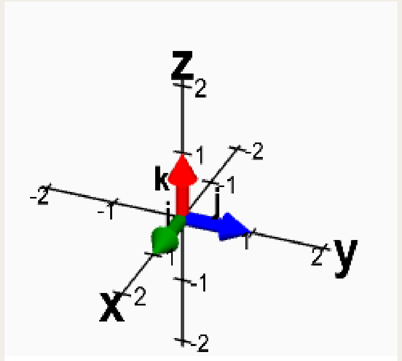
```
glm::mat4 view = glm::lookAt(cameraPos,
                              cameraPos + cameraFront,
                              cameraUp);
```

Walk around

Connect camera movements to keyboard event

The `cameraPos` vector can be updated when some keys are pressed. To this aim, the `processInput` is modified to manage additional GLFW's keyboard input.

```
void processInput(GLFWwindow* window)
{
    ...
    const float cameraSpeed = 0.05f;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        cameraPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        cameraPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
}
```



cross product

Note that the resulting right vector is normalized. If not normalized, the resulting cross product returns differently sized vectors based on the `cameraFront` variable.

Walk around

Movement speed

To avoid different speed movements that result from different processing powers of the machines, i.e., different time to render frame and calls of the `processInput` function, it is possible to use a `deltatime` variable that stores the time it took to render the last frame. All velocities can be multiplied with `deltatime` in this way, the velocity of the camera will be balanced out accordingly.

To compute `deltatime` two: global variable are needed:

```
float deltaTime = 0.0f;
float lastFrame = 0.0f;
```

Walk around

Movement speed

The `deltatime` value is calculated each frame as:

```
float currentFrame = static_cast<float>(glfwGetTime());
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

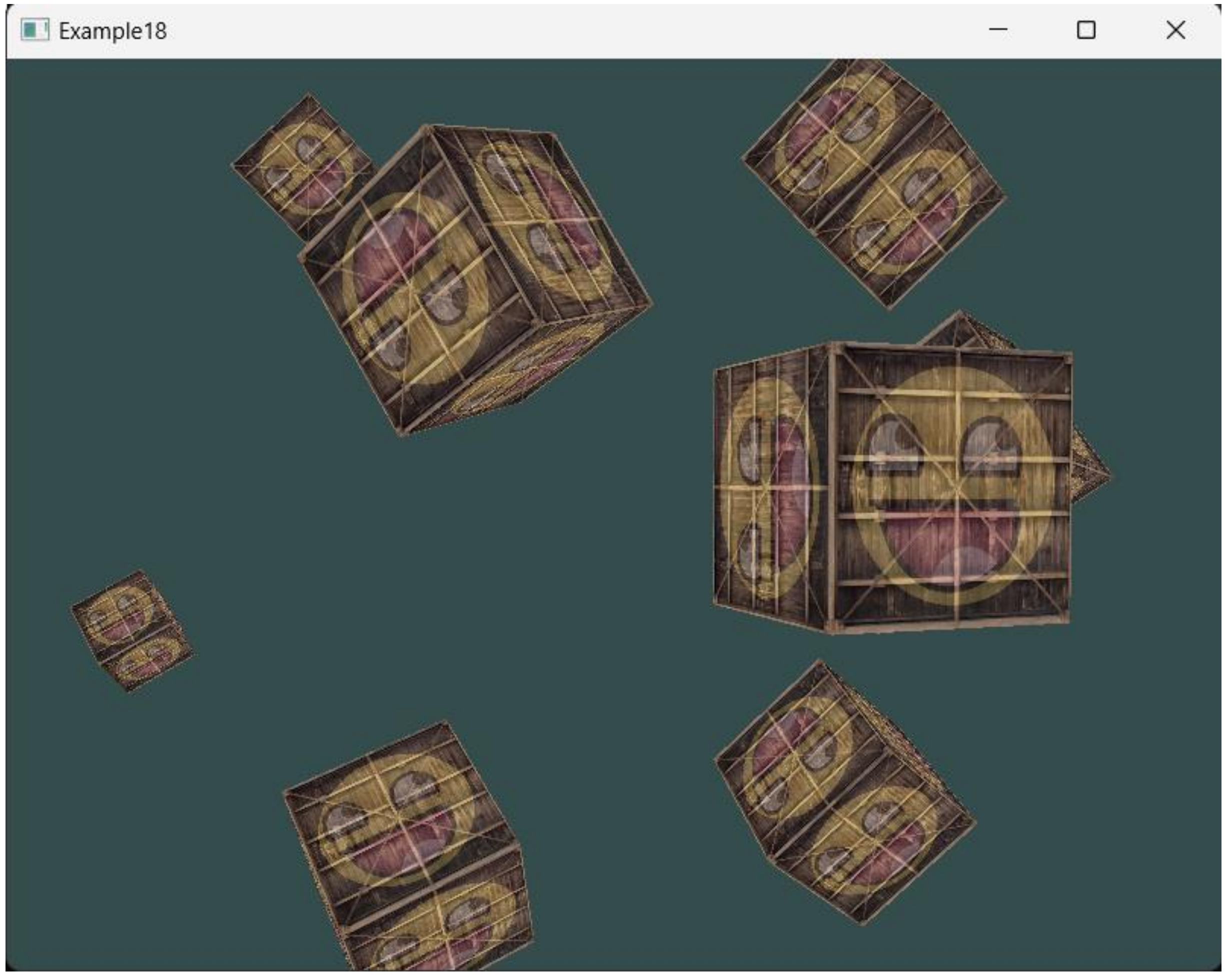
Once the `deltatime` is computed, it is possible to take it into account for calculating the velocities:

```
float cameraSpeed = static_cast<float>(2.5 * deltaTime);
```

Look around

Example18

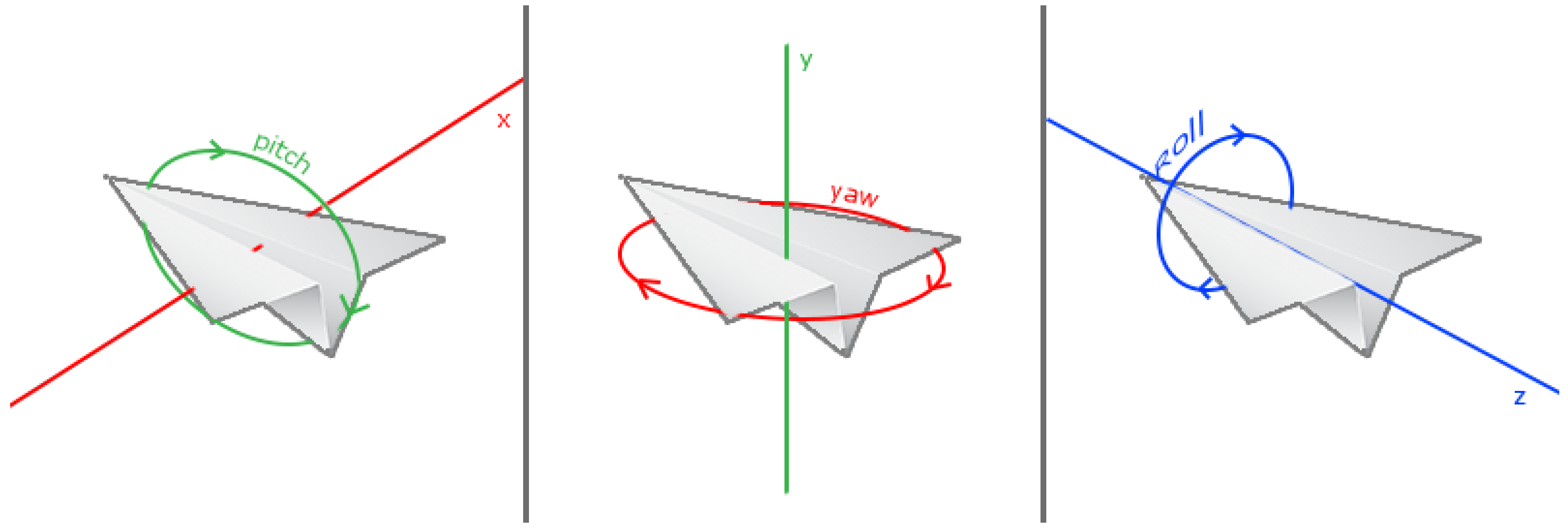
The example shows how to use the input of the mouse to change the direction of the `cameraFront` vector to enable the look around of the scene.



Look around

Euler angles

Euler angles are 3 values that can represent any rotation in 3D. There are 3 Euler angles: pitch, yaw and roll. Each of the Euler angles are represented by a single value and with the combination of all 3 of them it is possible to calculate any rotation vector in 3D.

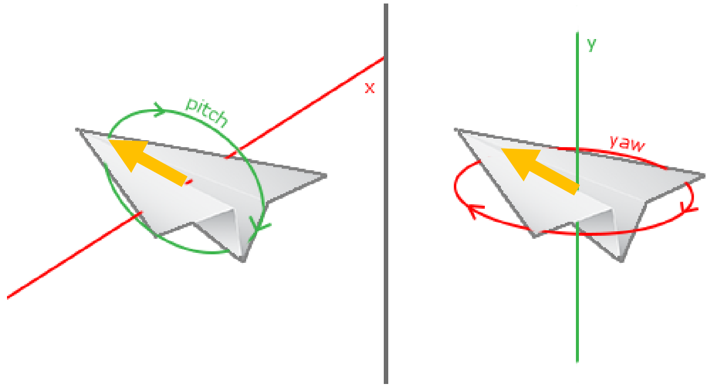


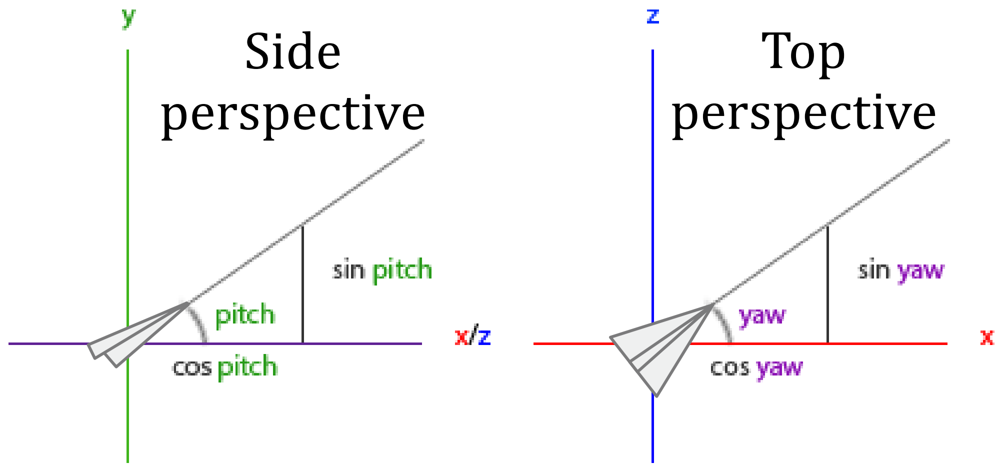
Look around

Euler angles

For the camera system, only the yaw and pitch values are needed. Given a pitch and a yaw value it is possible to convert them into a 3D vector that represents a new direction vector. The process of converting yaw and pitch values to a direction vector requires basic concepts of trigonometry:

```
glm::vec3 front;
front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
front.y = sin(glm::radians(pitch));
front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```





Look around

Euler angles

To make sure the camera points towards the negative z-axis by default it is necessary to give the yaw a default value of a 90-degree clockwise rotation. In fact, a yaw of 0.0 results in a direction vector pointing to the right. Positive degrees rotate counter-clockwise so the default yaw value is set to:

```
float yaw = -90.0f;
float pitch = 0.0f;
```

Look around

Mouse input

To control the yaw and pitch value, it is possible to use the mouse (or controller/joystick) movements. More specifically, the horizontal mouse-movement could affect the yaw, whereas vertical mouse-movement could affect the pitch. The idea is to store the last frame's mouse positions and calculate in the current frame how much the mouse values changed. The higher the horizontal or vertical difference, the more the pitch or yaw value are updated and thus the more the camera should move.

Look around

Mouse input

First, GLFW is used to hide the cursor and capture it. Capturing a cursor means that, once the application has focus, the mouse cursor stays within the center of the window (unless the application loses focus or quits). This is done with the following simple configuration:

```
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

To calculate the pitch and yaw values, it is necessary to tell GLFW to listen to mouse-movement events. This is sone by creating a callback function with the following prototype:

```
void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
```

The `xposIn`, `yposIn` represent the current mouse positions.

Look around

Mouse input

After registering the callback using the following function, each time the mouse moves, the `mouse_callback` function is called:

```
glfwSetCursorPosCallback(window, mouse_callback);
```

Look around

Mouse input

When handling mouse input for a fly style camera there are several steps to be considered before calculate the camera's direction vector:

- Calculate the mouse's offset since the last frame.
- Add the offset values to the camera's yaw and pitch values.
- Add some constraints to the minimum/maximum pitch values.
- Calculate the direction vector.

Look around

Mouse input

The first step is to calculate the offset of the mouse since last frame. First the last mouse positions has to be stored in the application. The position is initialized to be in the center of the screen (screen size is 800 by 600) initially:

```
float lastX = 800.0f / 2.0;
float lastY = 600.0 / 2.0;
```

Look around

Mouse input

Then in the mouse's callback function, the offset movement between the last and current frame is calculated and added to the globally declared pitch and yaw values:

```
float xoffset = xpos - lastX;
float yoffset = lastY - ypos; // reversed since y-coordinates go from 
bottom to top
lastX = xpos;
lastY = ypos;

float sensitivity = 0.1f;
xoffset *= sensitivity;
yoffset *= sensitivity;

yaw += xoffset;
pitch += yoffset;
```

Look around

Mouse input

In the third step some constraints are added to the camera so users won't be able to make weird camera movements (also causes a LookAt flip once direction vector is parallel to the world up direction). The pitch needs to be constrained in such a way that users won't be able to look higher than 89 degrees (at 90 degrees we get the LookAt flip) and also not below -89 degrees. This ensures the user will be able to look up to the sky or below to his feet but not further. The constraints work by replacing the Euler value with its constraint value whenever it breaches the constraint:

```
if (pitch > 89.0f)
    pitch = 89.0f;
if (pitch < -89.0f)
    pitch = -89.0f;
```

Look around

Mouse input

The fourth and last step is to calculate the actual direction vector using the previous formula:

```
glm::vec3 front;
front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
front.y = sin(glm::radians(pitch));
front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
cameraFront = glm::normalize(front);
```

Look around

Mouse input

To avoid a large sudden jump whenever the window first receives focus of the mouse cursor due to the original position of the mouse that is often significantly far away from the center of the screen (thus resulting in large offsets to be applied when the application is launched) it is possible to define a global bool variable to check if this is the first time that a mouse input is received. If yes, the initial mouse positions is updated to the new `xpos` and `ypos` values. The resulting mouse movements will then use the newly entered mouse's position coordinates to calculate the offsets:

```
if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }
```

Look around

Zoom

To implement a zooming interface, it is possible to manipulate the field of view. To zoom in, the mouse's scroll wheel can be used by registering a new callback function:

```
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    fov -= (float)yoffset;
    if (fov < 1.0f)
        fov = 1.0f;
    if (fov > 45.0f)
        fov = 45.0f;
}
```

When scrolling, the `yoffset` value contains the amount of vertical scroll. When the `scroll_callback` function is called, we change the value of a globally declared fov variable can be changed accordingly. Since 45.0 is the default fov value, constrains are added to limit the the zoom level between 1.0 and 45.0.

Look around

Zoom

The perspective projection matrix has to be updated each frame, by using the fov variable as its field of view:

```
glm::mat4 projection = glm::perspective(glm::radians(fov),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
ourShader.setMat4("projection", projection);
```

The scroll callback function is registered:

```
glfwSetScrollCallback(window, scroll_callback);
```

Camera class

Example19

The example shows how to implement the previous example by using a new class (`camera.h`) for managing the camera.