Lesson 4
3D drawing & coordinate systems

Minjing Yu
minjingyu@tju.edu.cn

Transformations

Transformations

- Transform an object using (multiple) matrix objects
- Transform an object in vertex shader
- Usually work with 4x4 transformation matrices because most of the vectors are of size 4
- Scaling, Translation, Rotation

Transformations

- Identity Matrix
  - an NxN matrix with only 0s except on its diagonal
  - leave a vector unchanged
  - usually a starting point for generating other transformation matrices

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Transformations

- Scaling Matrix
  - Scaling variables: (S_1, S_2, S_3)
  - 4th scaling vector stays 1

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Transformations

- Translation Matrix
  - Translation vector : (T_x, T_y, T_z)

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

W component

Transformations

- Homogeneous Coordinates
  - To get the homogeneous vector from a 3D vector
    - Add w component as 1
  - To get the 3D vector from a homogeneous vector
    - Divide the x, y and z coordinate by its w coordinate
  - Allows us to do translations on 3D vectors

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

W component

Transformations

- Rotation Matrix
  - Most rotation functions require an angle in radians, but luckily degrees are easily converted to radians:

    angle in degrees = angle in radians * (180.0f / PI)
    angle in radians = angle in degrees * (PI / 180.0f)
    Where PI equals (sort of) 3.14159265359.

  - Rotations in 3D are specified with an angle and a rotation axis
  - The angle specified will rotate the object along the rotation axis given

Transformations

- Rotation Matrix
  - A rotation matrix is defined for each unit axis in 3D space where the angle is represented as θ

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the X-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the Y-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Rotation around the Z-axis

Transformations

- Rotation Matrix
  - Rotate around an arbitrary axis (R_x, R_y, R_z)
  - the angle is θ

$$
\begin{bmatrix}
\cos\theta + {R_x}^2(1-\cos\theta) & R_xR_y(1-\cos\theta) - R_z\sin\theta & R_xR_z(1-\cos\theta) + R_y\sin\theta & 0 \\
R_yR_x(1-\cos\theta) + R_z\sin\theta & \cos\theta + {R_y}^2(1-\cos\theta) & R_yR_z(1-\cos\theta) - R_x\sin\theta & 0 \\
R_zR_x(1-\cos\theta) - R_y\sin\theta & R_zR_y(1-\cos\theta) + R_x\sin\theta & \cos\theta + {R_z}^2(1-\cos\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Transformations

- Combining matrices
  - Combine multiple transformations in a single matrix
  - Example: a vector (x,y,z), scale it by 2 and translate it by (1,2,3)

$$Trans.Scale = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Matrix multiplication is not commutative, order is important
  - Firstly scaling operations, then rotations and lastly translations

GLM library

- OpenGL does not have any form of matrix or vector knowledge built in, 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
  - only need include the proper header files
  - no linking and compiling necessary
- Copy the root directory of the header files into your includes folder

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

GLM library

- Data type:
  - glm::vec4, glm::mat4
- Function:

```
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));
trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
```

GLM expects its angles in radians so we convert the degrees to radians using glm::radians

- Example

```
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;
```

Coordinate Systems

Coordinate Systems

- OpenGL expects all the vertices, that we want to become visible, to be in Normalized Device Coordinates after each vertex shader
- 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
- We specify the coordinates in a range we configure ourselves and in the vertex shader transform these coordinates to NDC
- These NDC are then given to the rasterizer to transform them to 2D coordinates/pixels on your screen

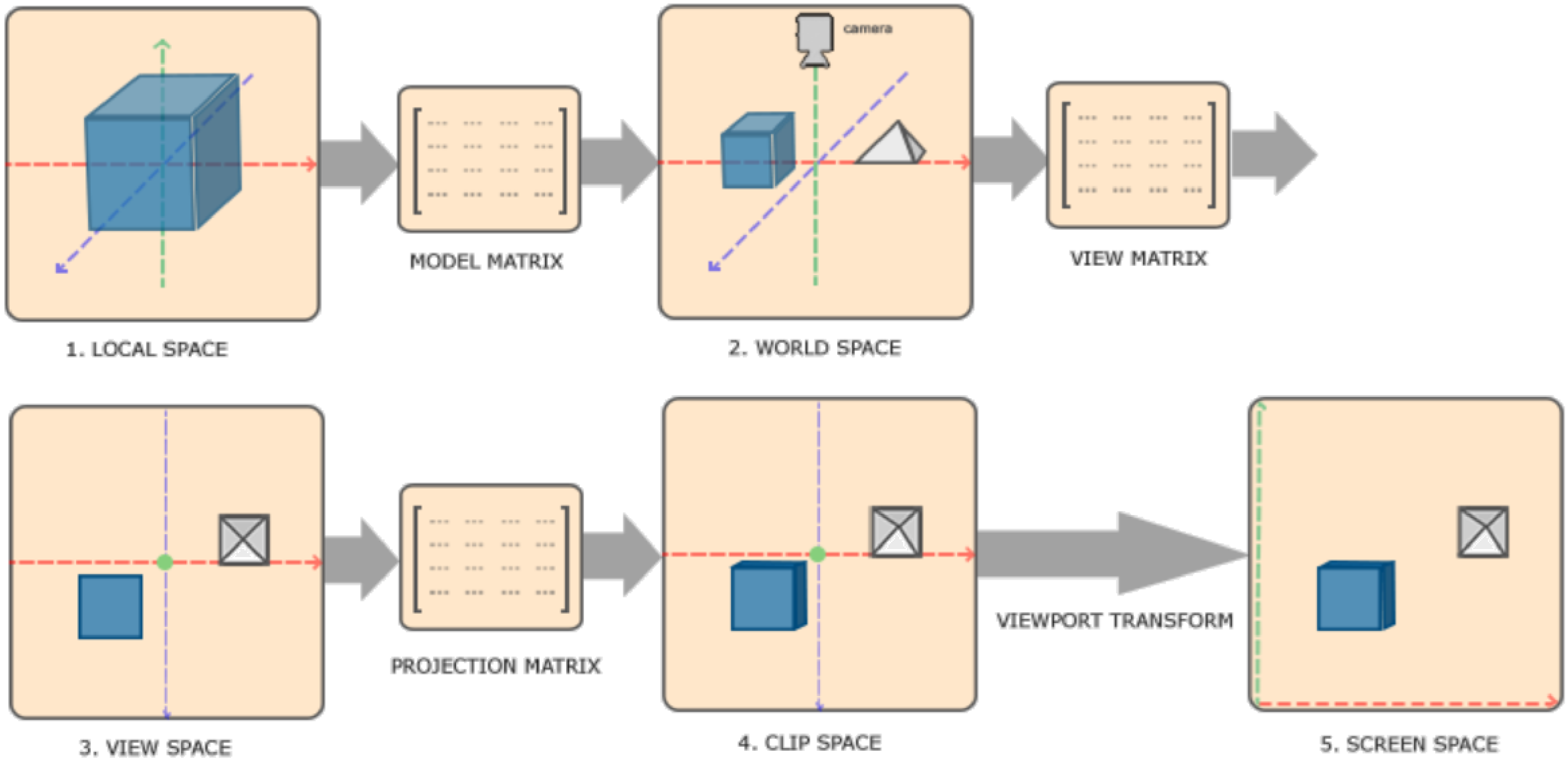
Coordinate Systems

- Transforming coordinates to NDC and then to screen coordinates is step-by-step, several coordinate systems are involved:

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

Coordinate Systems

- To transform the coordinates in one space to the next coordinate space, we will use Model, View and Projection matrix



Coordinate Systems

- Local coordinates are the coordinates of your object relative to its local origin
- World-space coordinates are relative to a global origin of the world
- View-space coordinates is as seen from the camera or viewer's point of view.
- Clip coordinates are processed to the -1.0 and 1.0 range and determine which vertices will end up on the screen.
- Screen coordinates are transformed by glViewport function to the coordinate range defined by glViewport, to be sent to the rasterizer to turn them into fragments.

Coordinate Systems

- Local Space
  - The coordinate space that is local to object
  - The origin of your object is probably at (0,0,0) even though might end up at a different location in your final application
- World space
  - The coordinates of all your vertices relative to a world
  - Accomplished with the Model Matrix
  - (Model Matrix) translates, scales and/or rotates your object to place it in the world at a location/orientation they belong to

```
glm::mat4 model(1);
model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

Coordinate Systems

- View space
  - Also known as the camera space or eye space
  - The result of transforming your world-space coordinates to coordinates that are in front of the user's view
  - The space as seen from the camera's point of view
  - Accomplished with the View Matrix
  - (View Matrix)Store the combination of translations and rotations to translate/rotate the scene

```
glm::mat4 view(1);      camera position              target position              up vector
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));
```

```
glm::mat4 view(1);
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

Coordinate Systems

- Clip space
  - OpenGL expects the coordinates to be within a specific range and any coordinate that falls outside this range is Clipped
  - We specify our own coordinate and convert those back to NDC
  - Accomplished with the Projection Matrix
  - (Projection Matrix) transforms coordinates within specified range to normalized device coordinates (-1.0, 1.0)

Coordinate Systems

- Frustum
  - This viewing box a projection matrix creates
  - Each coordinate inside frustum will end up on the user's screen
- Projection
  - The total process to convert coordinates within a specified range to NDC that can easily be mapped to 2D view-space coordinates
- The projection matrix has two forms
  - Orthographic projection matrix
  - Perspective projection matrix

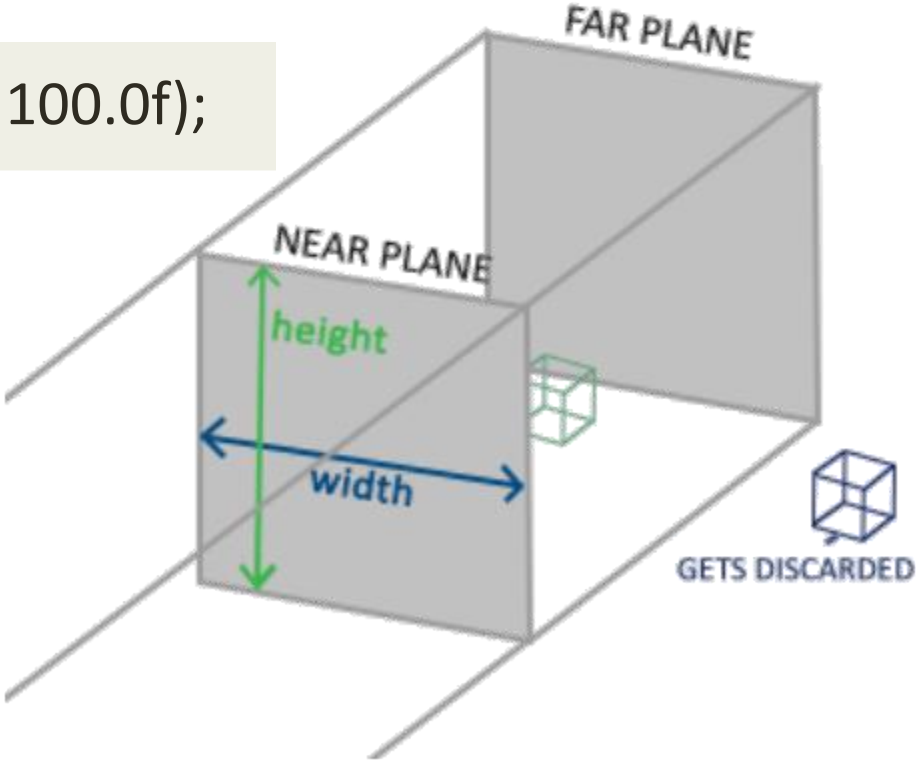
Orthographic Projection

- A cube-like frustum box
  - Vertex outside this box is clipped
  - Specified by a width, a height and a near and far plane
- Use function glm::ortho()

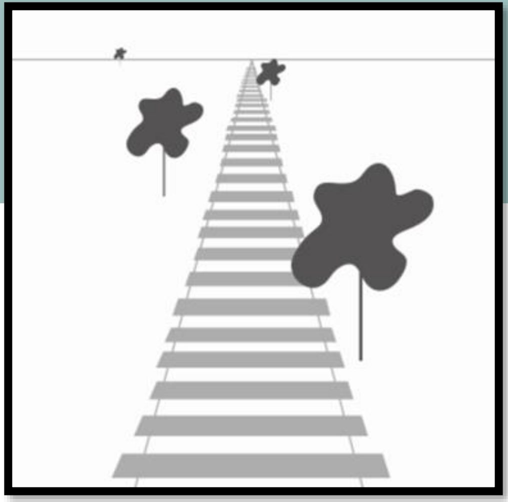
```
glm::mat4 proj = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);
```

The 1st and 2nd: left and right coordinate
The 3rd and 4th: bottom and top coordinate
The 5th and 6th: near and far plane coordinate

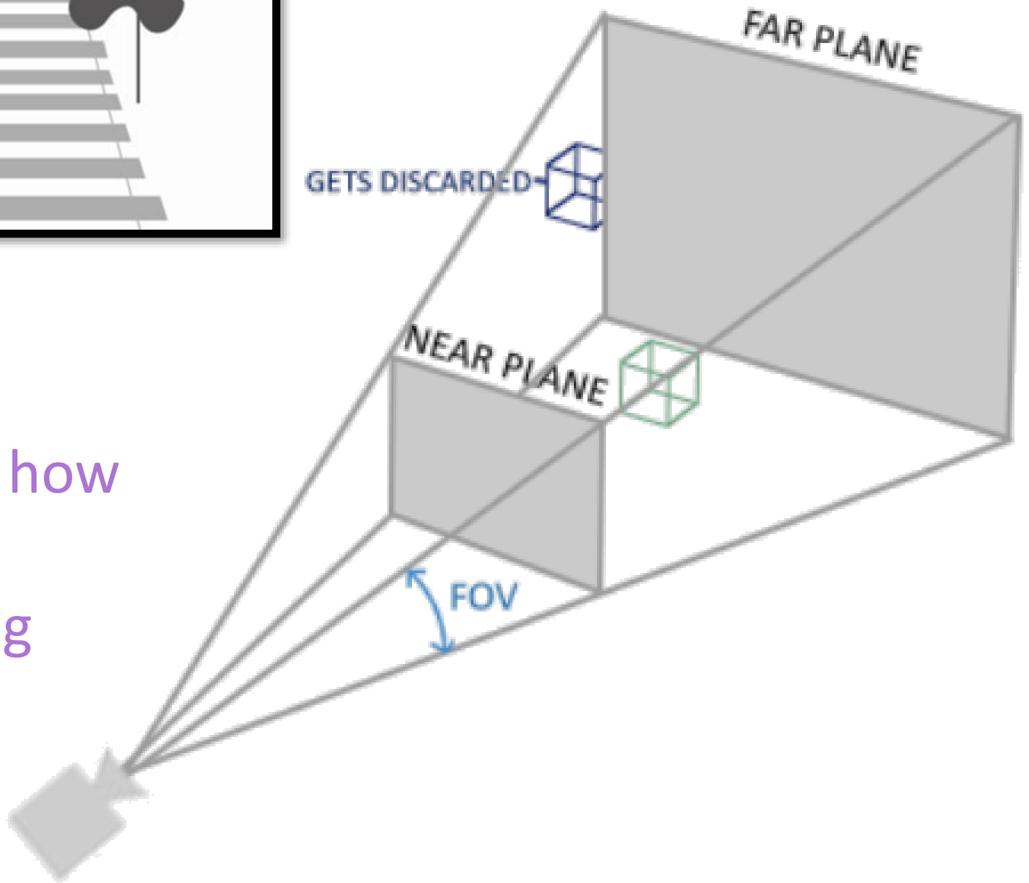
- Unrealistic since the projection doesn't take perspective into account



Perspective Projection



- Perspective
  - the farther, the smaller
- A non-uniformly shaped box
- Use function glm::perspective()



The 1st: the fov value, stands for field of view and sets how large the viewspace is

The 2nd: the aspect ratio which is calculated by dividing the viewport's width by its height

The 3rd and 4th: near and far plane coordinate

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

3D drawing

3D drawing

- A vertex coordinate is transformed to clip coordinates as follows

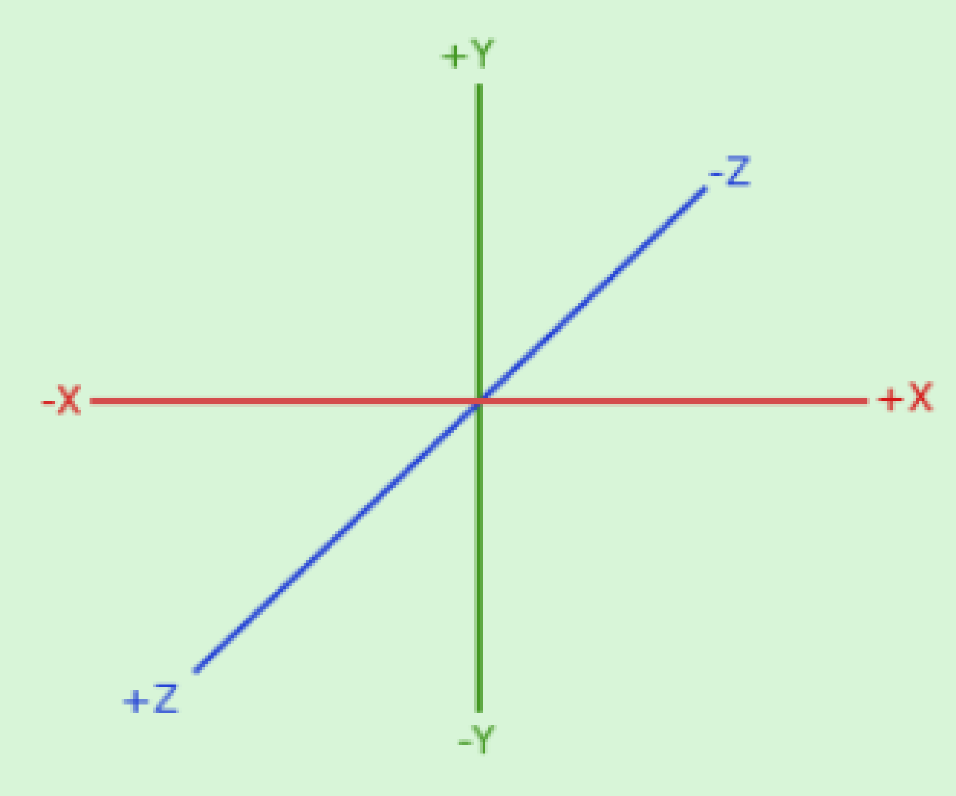
$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

- The order of matrix multiplication is reversed (remember that we need to read matrix multiplication from right to left)
- The resulting vertex should then be assigned to gl_Position in the vertex shader and OpenGL will then automatically perform perspective division and clipping.

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

3D drawing

- OpenGL is a right-handed system
- The positive x-axis is to your right, the positive y-axis is up and the positive z-axis is backwards



3D drawing

- Define a model matrix

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

- Define a view matrix

```
glm::mat4 view = glm::mat4(1.0f);
// note that we're translating the scene in the reverse direction of where we want to move
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

- Define a projection matrix

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

3D drawing

- Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
void main(){
    // note that we read the multiplication from right to left
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    ...
}
```

- Send the matrices to the shader

```
int modelLoc = glGetUniformLocation(ourShader.ID, "model");
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
... // same for View Matrix and Projection Matrix
```

3D drawing

- Depth Testing
  - OpenGL stores all its depth information in a z-buffer(Depth buffer)
  - Whenever the fragment wants to output its color, OpenGL compares its depth values with the z-buffer and if the current fragment is behind the other fragment it is discarded, otherwise overwritten
  - Enable depth testing:

This functionality is enabled/disabled until another call is made to disable/enable it

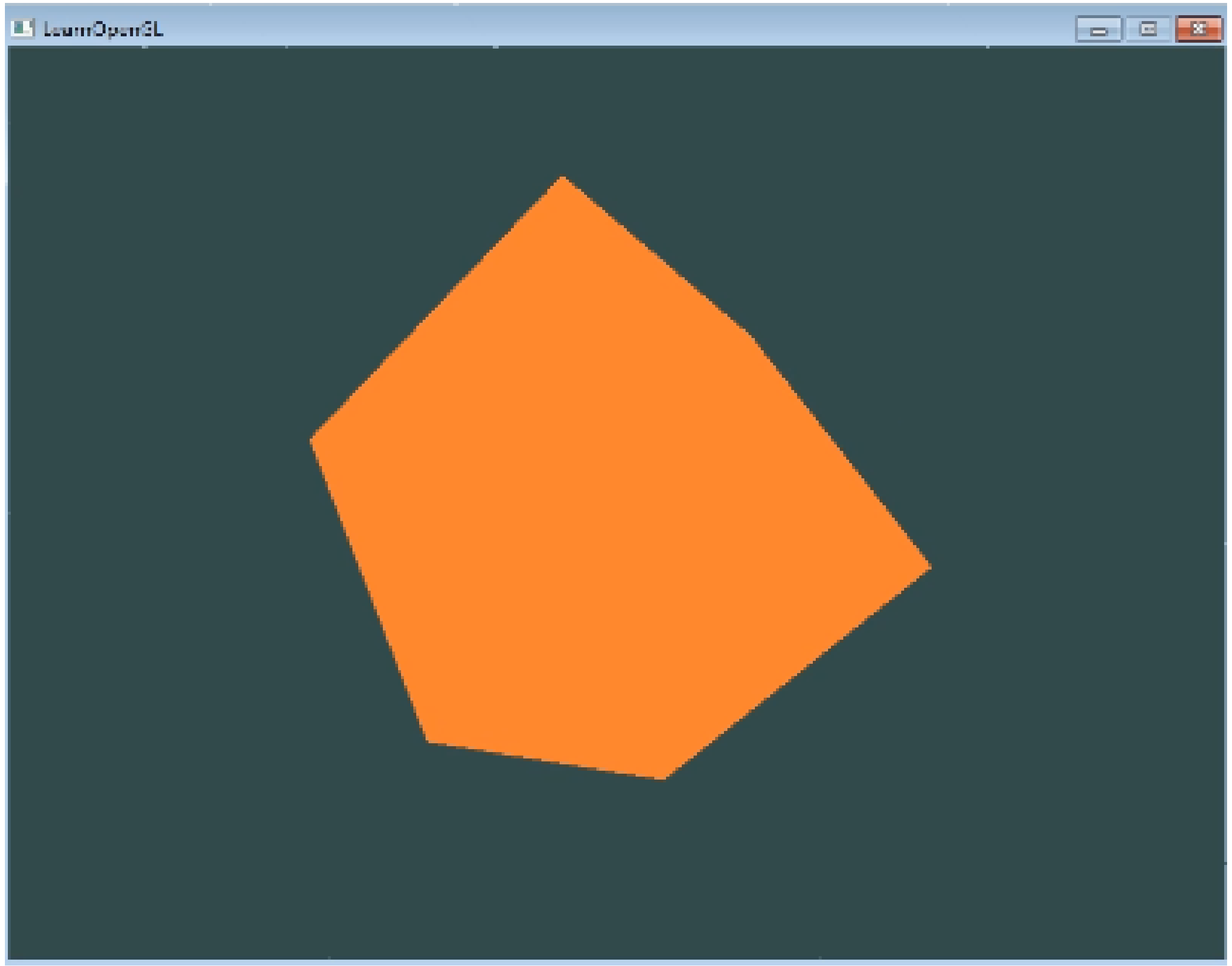
```
glEnable(GL_DEPTH_TEST);
```

  - Clearing the color buffer:

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

3D drawing

(SEE attached Tutorial4 folder)



Thanks!