Computer Graphics
– Coordinate Systems

J.-Prof. Dr. habil. Kai Lawonn

Introduction

- Last lecture: learned how to use matrices to transform vertices with
- Reminder: OpenGL expects visible vertices to be in normalized device coordinates (NDC) after each vertex shader run ($x, y, z \in [-1,1]$)
- Usually the coordinates are in a given range and in the vertex shader these coordinates are transformed to NDC
- NDC coordinates are then given to the rasterizer to transform them to 2D coordinates/pixels on the screen

Introduction

- Transforming coordinates to NDC and then to screen coordinates is usually accomplished in a step-by-step fashion where object's vertices transformed to several coordinate systems
- The advantage: some operations/calculations are easier in certain coordinate systems

Introduction

- There are a total of 5 different coordinate systems that are of importance:
  - Local space (or Object space)
  - World space
  - View space (or Eye space)
  - Clip space
  - Screen space

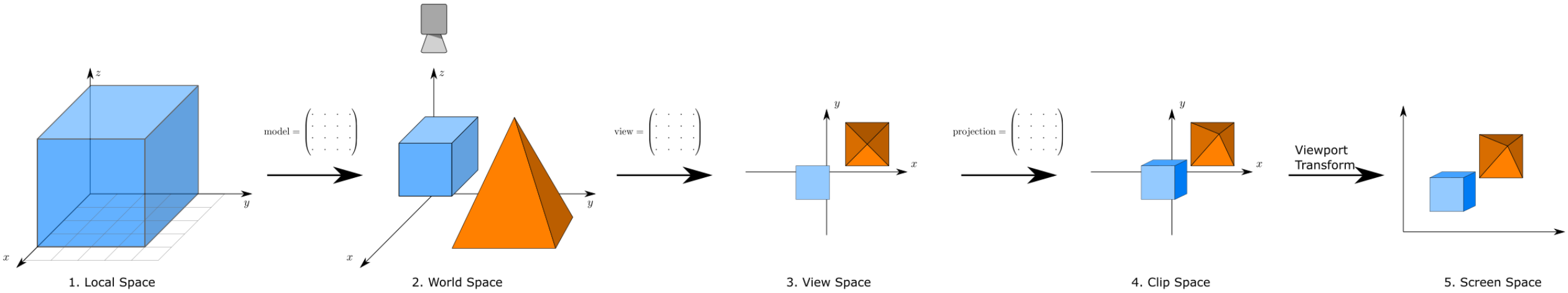
Spaces

Global Picture

- Transformation of the coordinates from one space to the next coordinate space involves several transformation matrices
- Most important matrices are:
  - model
  - view
  - projection
- Vertex coordinates (order of coordinates):
  - Local space (local coordinates),
  - World coordinates
  - View coordinates,
  - clip coordinates
  - (eventually) end up as screen coordinates

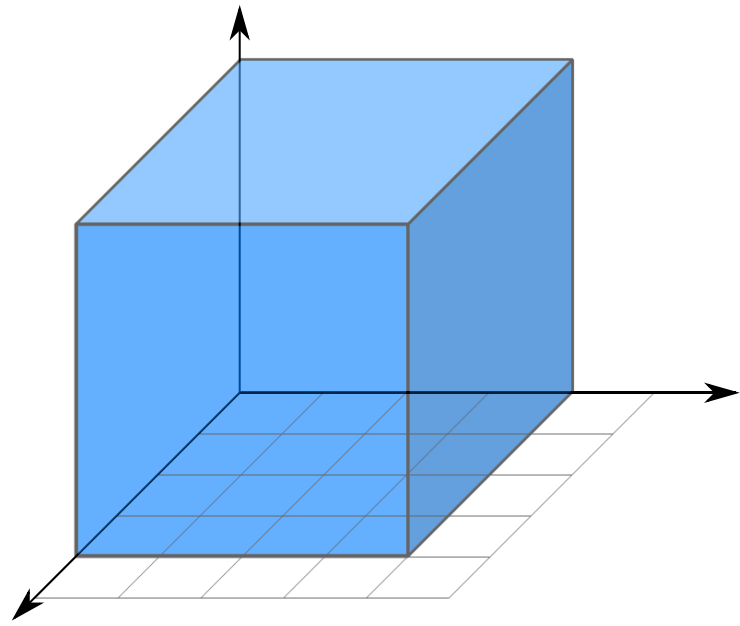
Global Picture

- Overview



Global Picture

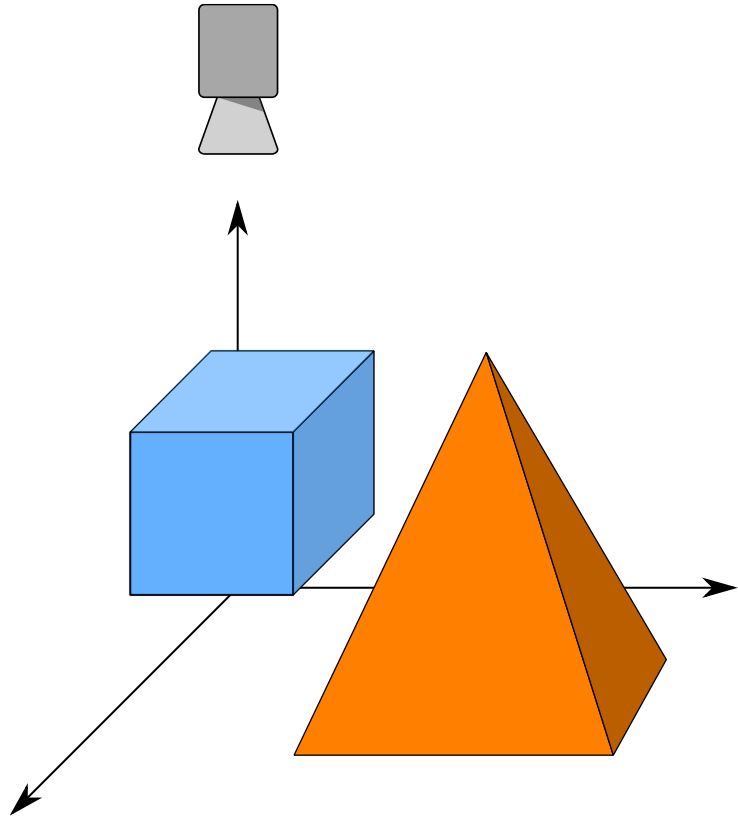
- Local coordinates are the coordinates of the object relative to its local origin
- E.g., the positions of vertices of your 3D scan



1. Local Space

Global Picture

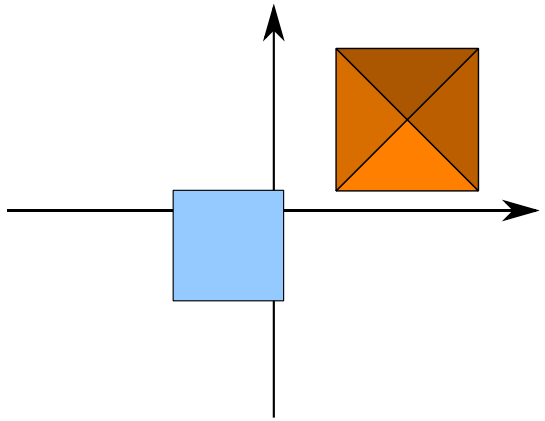
- Next step is to transform the local coordinates to world-space coordinates (model matrix)
- If we have several objects, they life initially in their own coordinate system
- The model transformation puts them in a global space (world space)



2. World Space

Global Picture

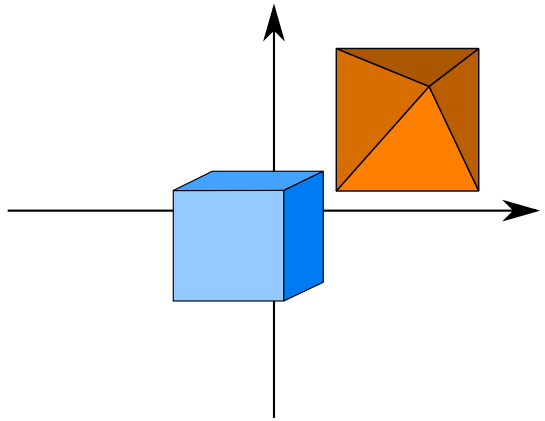
- Next, transform the world coordinates to view-space coordinates (view matrix) in such a way that each coordinate is as seen from the camera or viewer's point of view (in direction of the negative z axis)



3. View Space

Global Picture

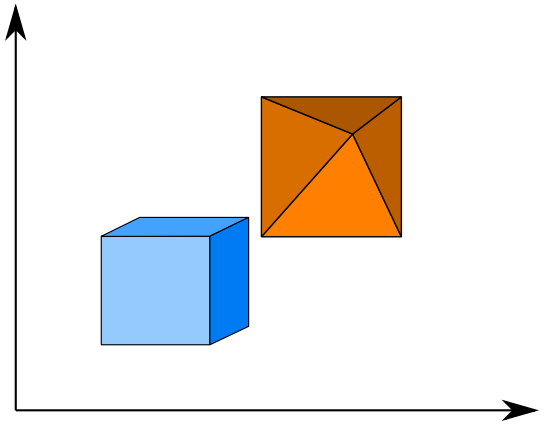
- From view space, we project them to clip coordinates (projection matrix)
- Clip coordinates are processed to the -1.0 and 1.0 range and determine which vertices will end up on the screen



4. Clip Space

Global Picture

- Lastly, transformation of the clip coordinates to screen coordinates (viewport transform) that transforms the coordinates from -1.0 and 1.0 to the coordinate range defined by glViewport
- The resulting coordinates are then sent to the rasterizer to turn them into fragments

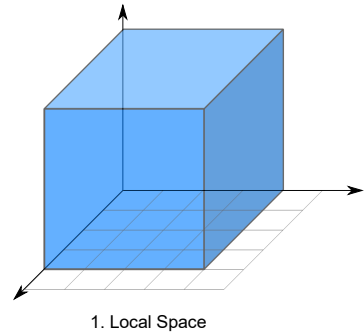


5. Screen Space

Global Picture

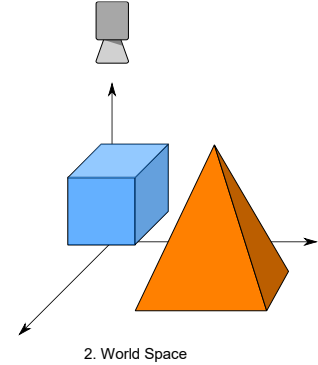
- Reason for transforming the vertices into all these different spaces is that some operations make more sense or are easier to use in certain coordinate systems
- E.g., modifying an object makes most sense to do this in local space
- E.g., calculating operations on the object with respect to the position of other objects makes most sense in world coordinates
- Could define one transformation matrix that goes from local space to clip space all in one go, but that leaves us with less flexibility

Local Space



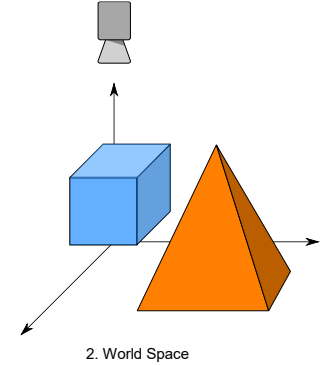
- Local space is the coordinate space that is local to the object (where object begin)
- E.g., a modelled cube from a software package (like Blender)
- Origin of cube is probably at (0,0,0) even though the cube might end up at a different location in the final application
- Probably all the models you've created all have (0,0,0) as their initial position
- All the vertices of your model are therefore in local space → they are all local to the object
- The vertices of the wall we defined in the last lectures were specified as coordinates between -0.5 and 0.5 with 0.0 as its origin → these are local coordinates

World Space



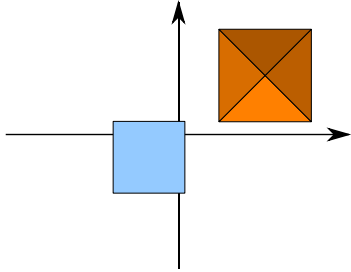
- Import all objects directly would probably result all being stacked → want to define individual positions for each object inside a larger world
- World space coordinates are coordinates of all vertices relative to a world
- Coordinates of the objects are transformed from local to world space with the model matrix.

World Space



- Model matrix is a transformation matrix that translates, scales and/or rotates
- E.g., transforming a house by scaling it down (it was a bit too large in local space), translating it to a suburbia town and rotating it a bit to the left on the y-axis
- Similar to the transformations of last lecture

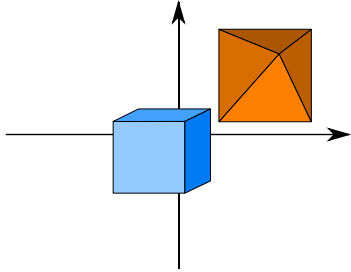
View Space



3. View Space

- View space is usually referred to as the camera of OpenGL (alias camera space/eye space)
- View space is the result of transforming world-space coordinates to coordinates that are seen from the camera’s point of view
- Accomplished with a combination of translations and rotations that certain items are in front of the camera (view matrix)

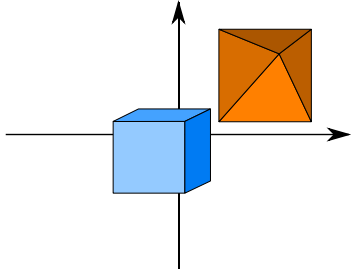
Clip Space



4. Clip Space

- At the end of each vertex shader run, OpenGL expects NDC any coordinate outside this range is clipped (clipped cordinates are discarded)
- Remaining coordinates will end up as visible fragments
- Projection matrix transforms vertex coordinates from view to clip-space, it specifies a range of coordinates e.g. -1000 and 1000 in each dimension and it transforms them to the NDC
- With this example a coordinate of (1250, 500, 750) would not be visible (x outside)

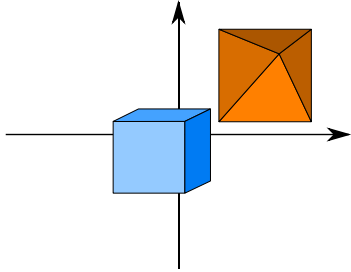
Clip Space



4. Clip Space

- Specifying a range of coordinates in each dimension results in a *viewing box*
- This is called a frustum and each coordinate that ends up inside it will end up on the screen
- Converting coordinates within a specified range to NDC is called projection (projects 3D coordinates to 2D NDC)

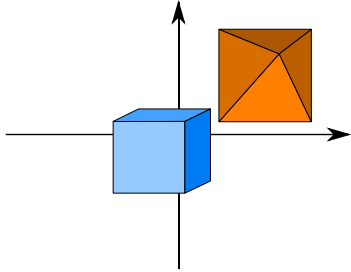
Clip Space



4. Clip Space

If only a part of a primitive, e.g., a triangle is outside the clipping volume OpenGL will reconstruct the triangle as one or more triangles to fit inside the clipping range

Clip Space



4. Clip Space

- After the vertices are transformed to clip space a final operation called perspective division is performed (division of x-, y-, z-components of the position vectors by the vector's homogeneous w component)
- It transforms the 4D clip space coordinates to 3D NDCs (automatically performed at the end of each vertex shader run)
- It is after this stage where the resulting coordinates are mapped to screen coordinates (using the settings of glViewport) and turned into fragments
- The projection matrix can take two different forms:
  - orthographic projection matrix
  - perspective projection matrix

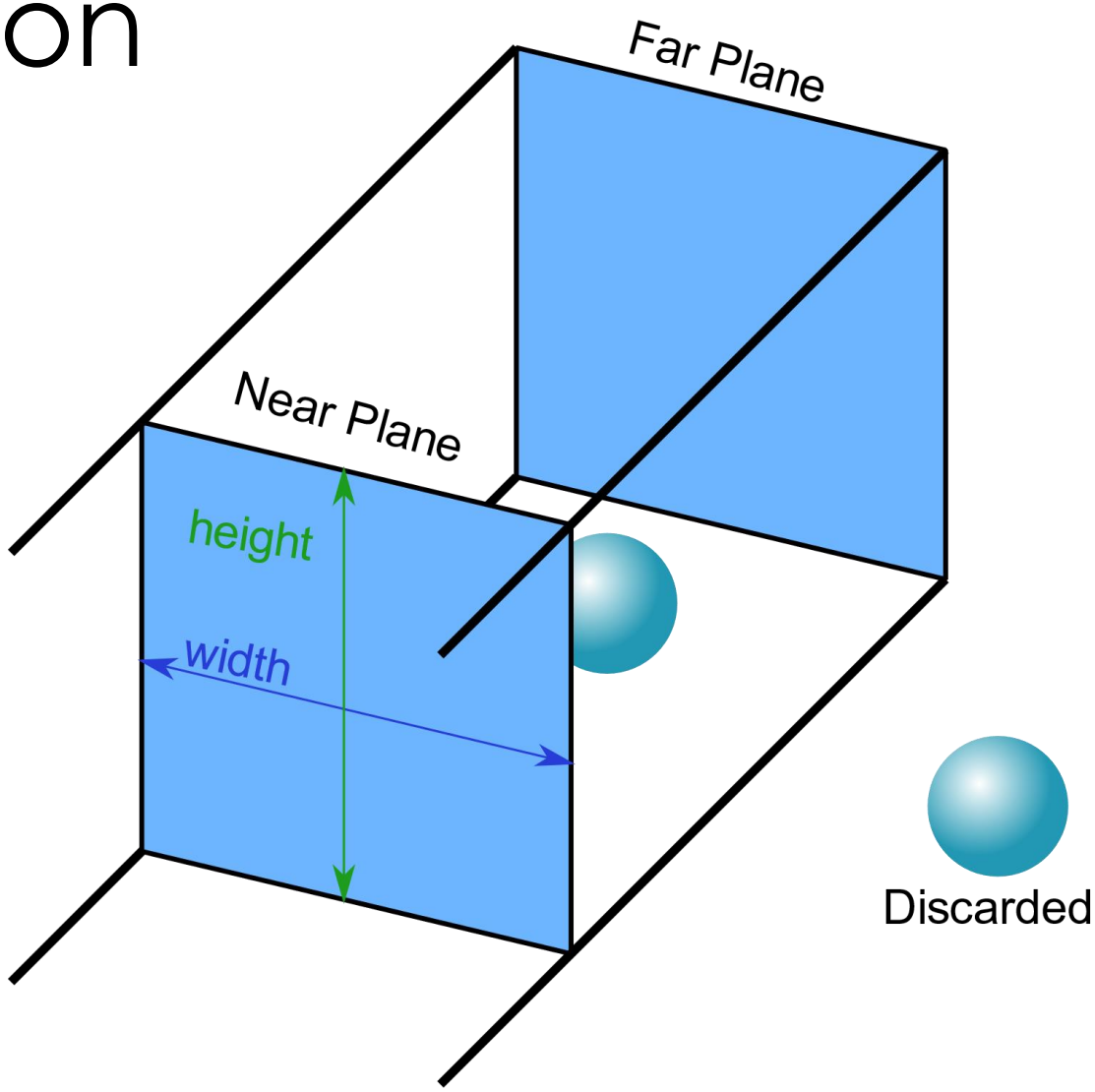
Projection Matrix

Orthographic Projection

- An orthographic projection matrix defines a cube-like frustum box that defines the clipping space where each vertex outside this box is clipped
- Orthographic projection matrix needs the width, height and length of the visible frustum
- All the coordinates that end up inside this frustum after transforming them to clip space with the orthographic projection matrix won’t be clipped
- The frustum looks a bit like a container

Orthographic Projection

- Frustum defines visible coordinates (width, height, near, far plane)
- Coordinates outside are clipped/discarded
- The orthographic frustum maps coordinates inside the frustum to NDC (w component = 1 → perspective division doesn't change the coordinates)



Orthographic Projection

- To create an orthographic projection matrix, use GLM's built-in function glm::ortho:

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

- 1./2. parameters: specify the left and right coordinate of the frustum
- 3./4. parameters: specify the bottom and top part of the frustum
- 5./6. parameters: define the distances between the near and far plane
- This projection matrix transforms all coordinates between these x, y and z range values to NDCs

Orthographic Projection

- An orthographic projection matrix directly maps coordinates to the 2D plane (screen), but in reality, a direct projection produces unrealistic results since the projection doesn’t take perspective into account
- That is something the perspective projection matrix fixes for us

Spaces

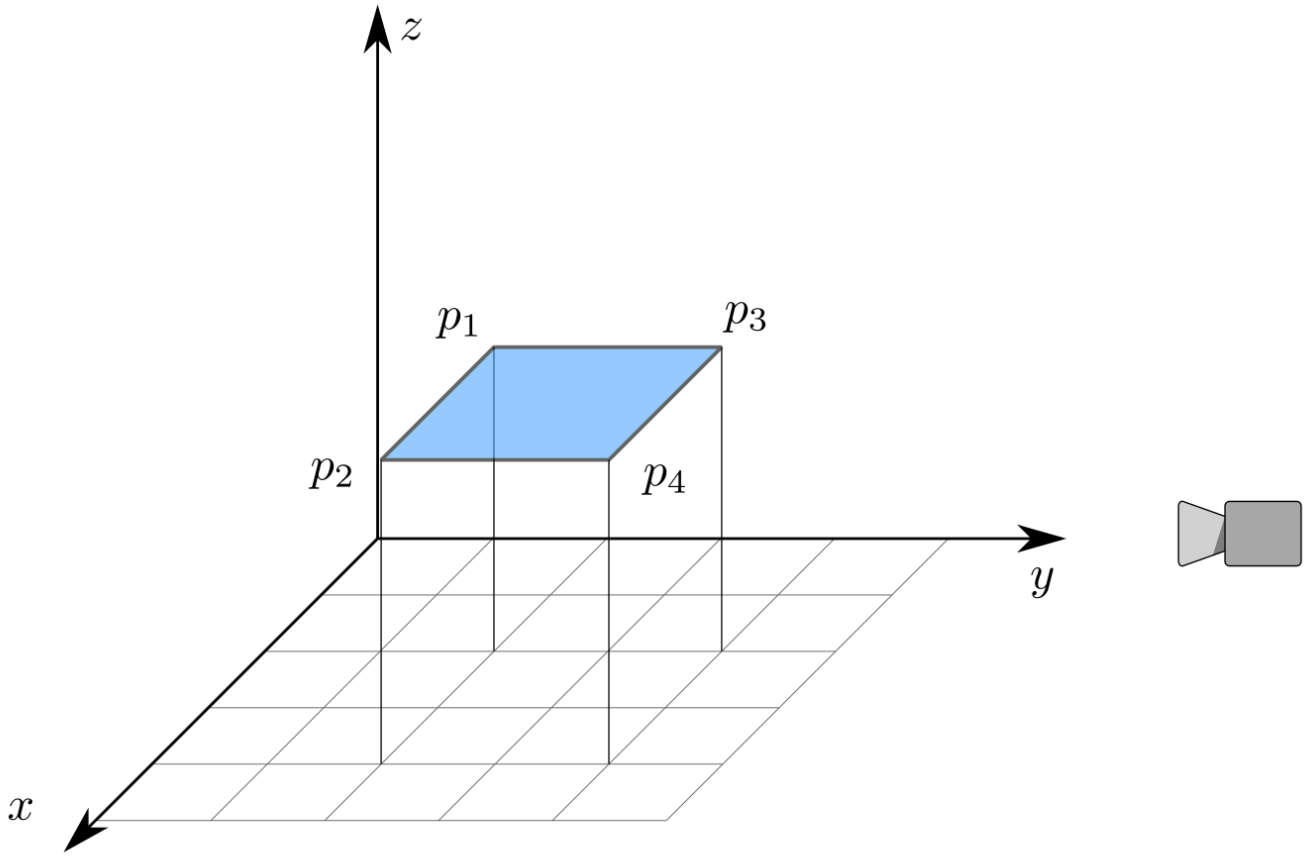
- Example: Given are four points that yield a plane

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

- The camera is positioned on the y-axis looking to the origin

Spaces

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$



Spaces

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

- We want translate, scale, and rotate the plane
- We want to translate the midpoint to the origin: translate by $-\bar{p}$
- Scale it by 5
- Rotate around the x-axis by -30°

Spaces

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

- We want translate, scale, and rotate the plane
- We want to translate the midpoint to the origin: translate by $-\bar{p}$
- Scale it by 5
- Rotate around the x-axis by -30°

$$rotate \cdot scale \cdot trans = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-30°) & -\sin(-30°) & 0 \\ 0 & \sin(-30°) & \cos(-30°) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -7.5 \\ 0 & 1 & 0 & -7.5 \\ 0 & 0 & 1 & -10 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\approx \begin{pmatrix} 5 & 0 & 0 & -37.5 \\ 0 & 4.33 & 2.5 & -57.48 \\ 0 & -2.5 & 4.33 & -24.55 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Spaces

- Note: translation*rotation*scale (this is what you read in other tutorials probably)
- That works, if the model is placed properly and you want to translate it afterwards

Spaces

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

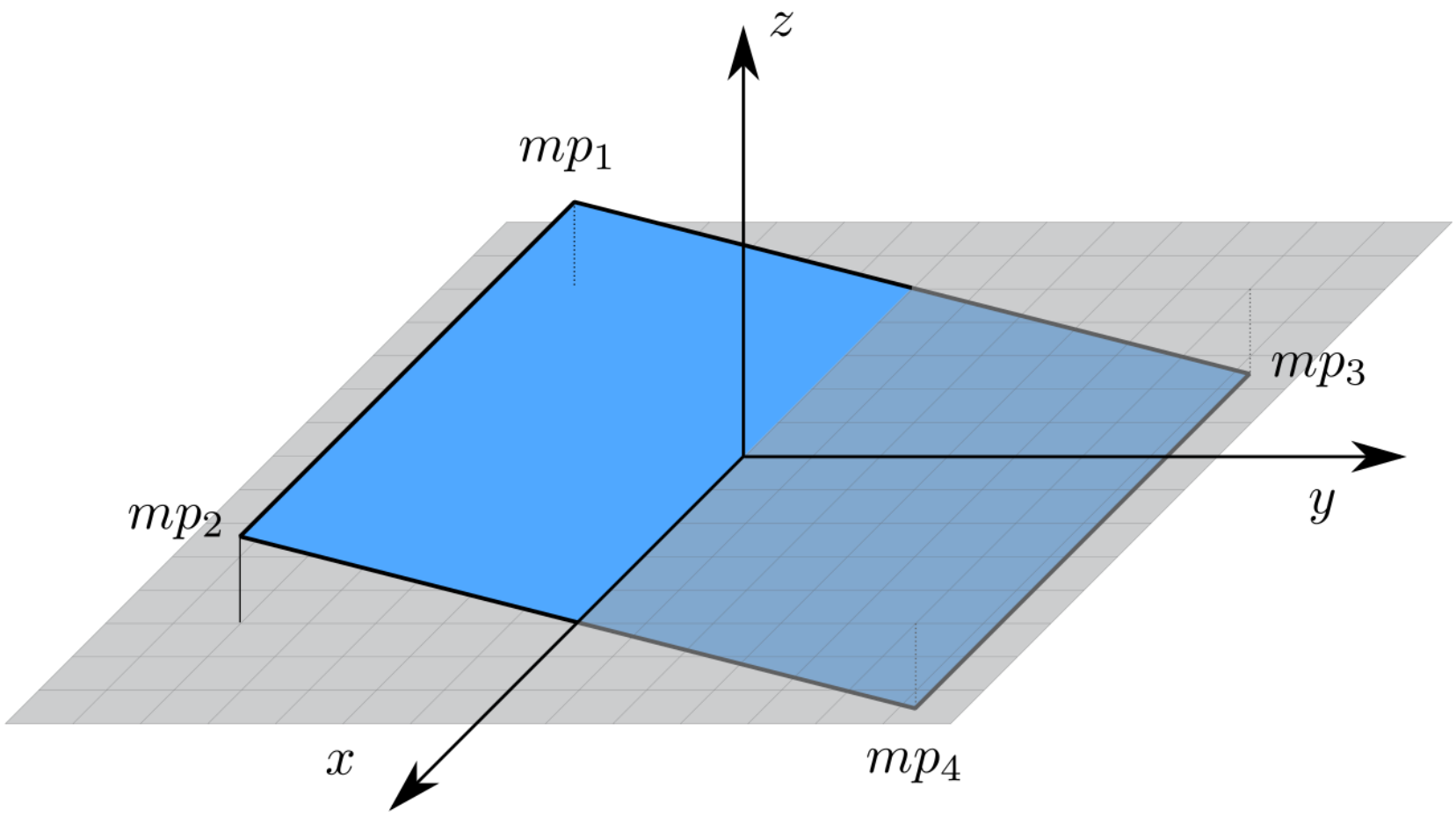
- We get: $mp_i := model \cdot p_i = rotate \cdot scale \cdot trans \cdot p_i$

$$mp_1 \approx \begin{pmatrix} -12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_2 \approx \begin{pmatrix} 12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_3 \approx \begin{pmatrix} -12.5 \\ 10.83 \\ -6.25 \end{pmatrix}, mp_4 \approx \begin{pmatrix} 12.5 \\ 10.83 \\ -6.25 \end{pmatrix}$$

- Note: for multiplication with a 4x4 matrix, we add a 1 to the last component and omit it here for the 3D vector

Spaces

$$mp_1 \approx \begin{pmatrix} -12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_2 \approx \begin{pmatrix} 12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_3 \approx \begin{pmatrix} -12.5 \\ 10.83 \\ -6.25 \end{pmatrix}, mp_4 \approx \begin{pmatrix} 12.5 \\ 10.83 \\ -6.25 \end{pmatrix}$$

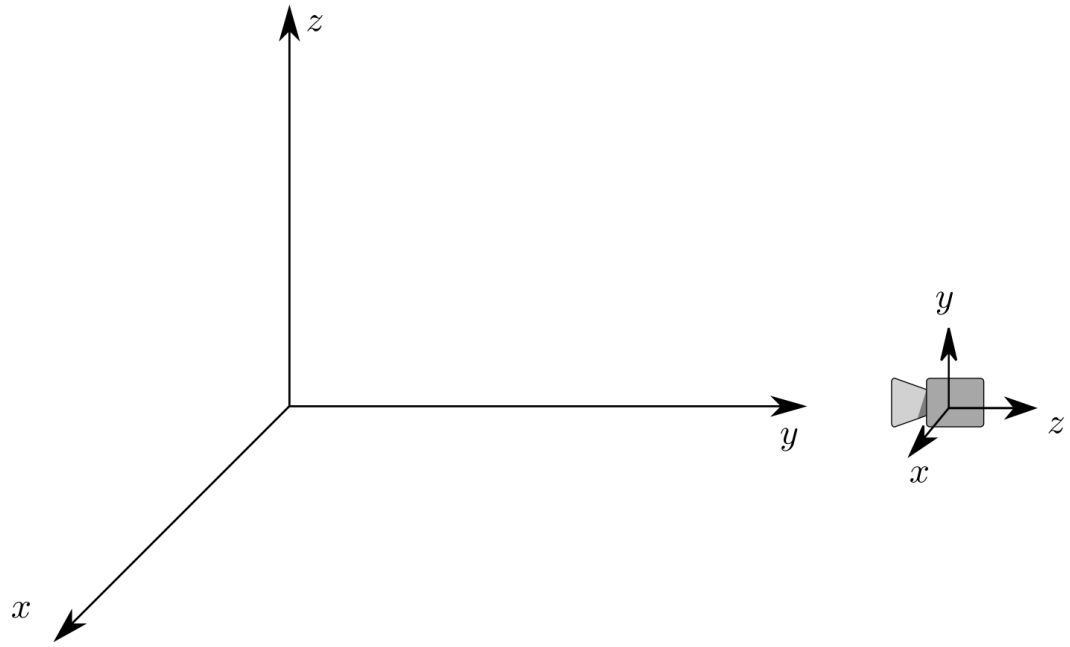


Spaces

$$mp_1 \approx \begin{pmatrix} -12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_2 \approx \begin{pmatrix} 12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_3 \approx \begin{pmatrix} -12.5 \\ 10.83 \\ -6.25 \end{pmatrix}, mp_4 \approx \begin{pmatrix} 12.5 \\ 10.83 \\ -6.25 \end{pmatrix}$$

- The camera is positioned on the y-axis

$$view = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$mp_1 \approx \begin{pmatrix} -12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_2 \approx \begin{pmatrix} 12.5 \\ -10.83 \\ 6.25 \end{pmatrix}, mp_3 \approx \begin{pmatrix} -12.5 \\ 10.83 \\ -6.25 \end{pmatrix}, mp_4 \approx \begin{pmatrix} 12.5 \\ 10.83 \\ -6.25 \end{pmatrix}$$

Spaces

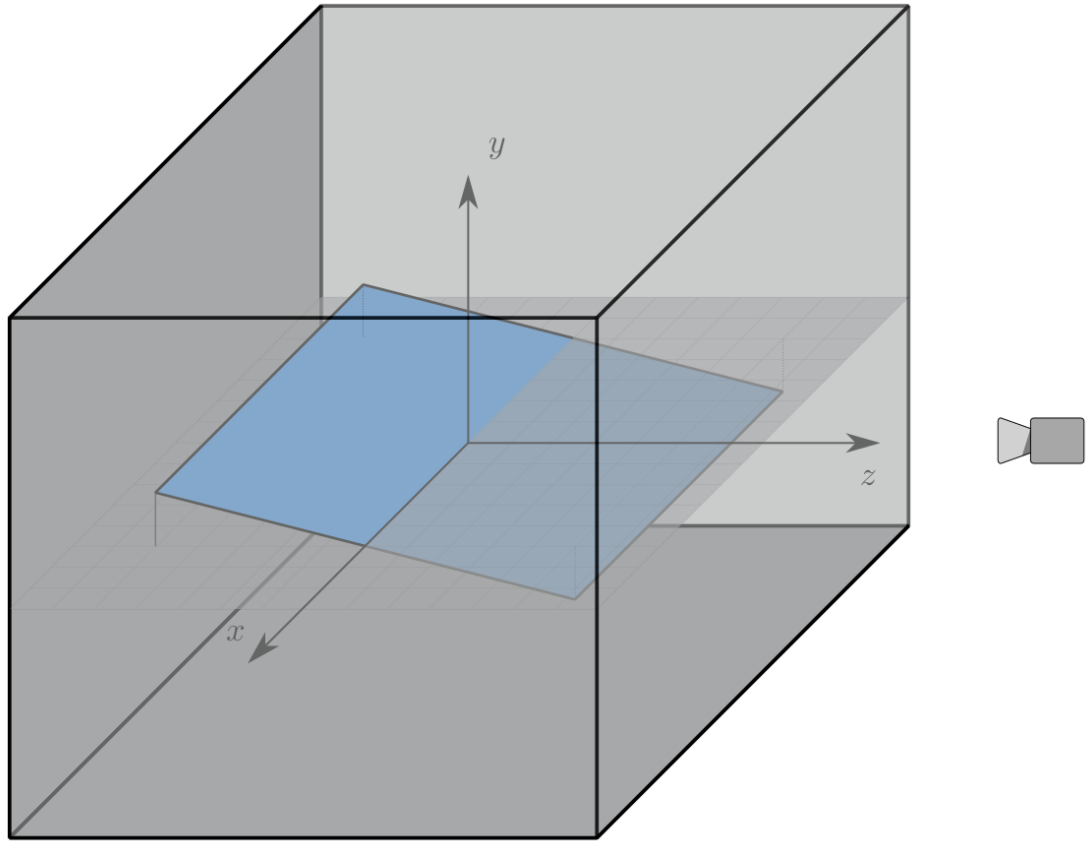
- We get: $vmp_i := view \cdot mp_i$

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ 10.83 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ 10.83 \end{pmatrix}$$

Spaces

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ 10.83 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ 10.83 \end{pmatrix}$$

- -15, 15: left and right coordinate of the frustum
- -10,10: bottom and top part of the frustum
- -12,12: distances between the near and far plane



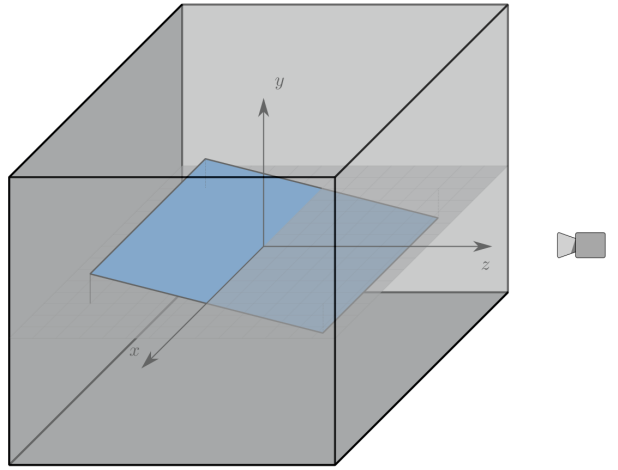
Spaces

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ 10.83 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ 10.83 \end{pmatrix}$$

- Projection matrix

$$proj = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1/15 & 0 & 0 & 0 \\ 0 & 1/10 & 0 & 0 \\ 0 & 0 & -1/12 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Spaces

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -10.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ 10.83 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ 10.83 \end{pmatrix}$$

- We get: $pvmp_i := proj \cdot vmp_i$

$$pvmp_1 \approx \begin{pmatrix} -0.83 \\ 0.63 \\ 0.90 \end{pmatrix}, pvmp_2 \approx \begin{pmatrix} 0.83 \\ 0.63 \\ 0.90 \end{pmatrix}, pvmp_3 \approx \begin{pmatrix} -0.83 \\ -0.63 \\ -0.90 \end{pmatrix}, pvmp_4 \approx \begin{pmatrix} 0.83 \\ -0.63 \\ -0.90 \end{pmatrix}$$

Spaces

- All together:

```
 float vertices[] = {
// positions           // texture coords
5.f,  5.f, 10.0f,   1.0f, 1.0f, // top right
5.f, 10.f, 10.0f,   1.0f, 0.0f, // bottom right
10.f, 10.f, 10.0f,   0.0f, 0.0f, // bottom left
10.f,  5.f, 10.0f,   0.0f, 1.0f  // top left
};
```

Spaces

- All together:

```
glm::mat4 model = glm::mat4(1.0f);
glm::mat4 view= glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);

model = glm::rotate(model, glm::radians(-30.0f), glm::vec3(1.0, 0.0, 0.0));
model = glm::scale(model, glm::vec3(5.0));
model = glm::translate(model, glm::vec3(-7.5f, -7.5f, -10.0f));

glm::vec4 r1 = glm::vec4(1,0,0,0);
glm::vec4 r2 = glm::vec4(0, 0, 1, 0);
glm::vec4 r3 = glm::vec4(0, 1, 0, 0);
glm::vec4 r4 = glm::vec4(0, 0, 0, 1);
view = glm::mat4(r1, r2, r3, r4);

projection = glm::ortho(-15.0f, 15.0f, -10.0f, 10.0f, -12.0f, 12.0f);
```

Spaces

- All together:

```
unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");
unsigned int viewLoc  = glGetUniformLocation(ourShader.ID, "view");
unsigned int projLoc  = glGetUniformLocation(ourShader.ID, "projection");

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, &model[0][0]);
glUniformMatrix4fv(viewLoc,  1, GL_FALSE, &view[0][0]);
glUniformMatrix4fv(projLoc,  1, GL_FALSE, &projection[0][0]);
```

Spaces

- All together:

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

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

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

F5…

- …finally done

Perspective Projection

- In real life objects that are farther away appear much smaller
- This weird effect is called perspective
- Perspective is especially noticeable when looking down the road

Perspective Projection

- Due to perspective the lines seem to coincide the farther they are away
- This is the effect the perspective projection matrix tries to mimic
- Maps a frustum range to clip space and manipulates the w value of each vertex coordinate

Perspective Projection

- The further away a vertex coordinate is from the viewer, the higher is w
- The coordinates are transformed to clip space are in the range -w to w (anything outside this range is clipped)
- OpenGL requires NDC as the final vertex shader output, thus in clip space, perspective division is applied to the clip space coordinates:

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

Perspective Projection

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

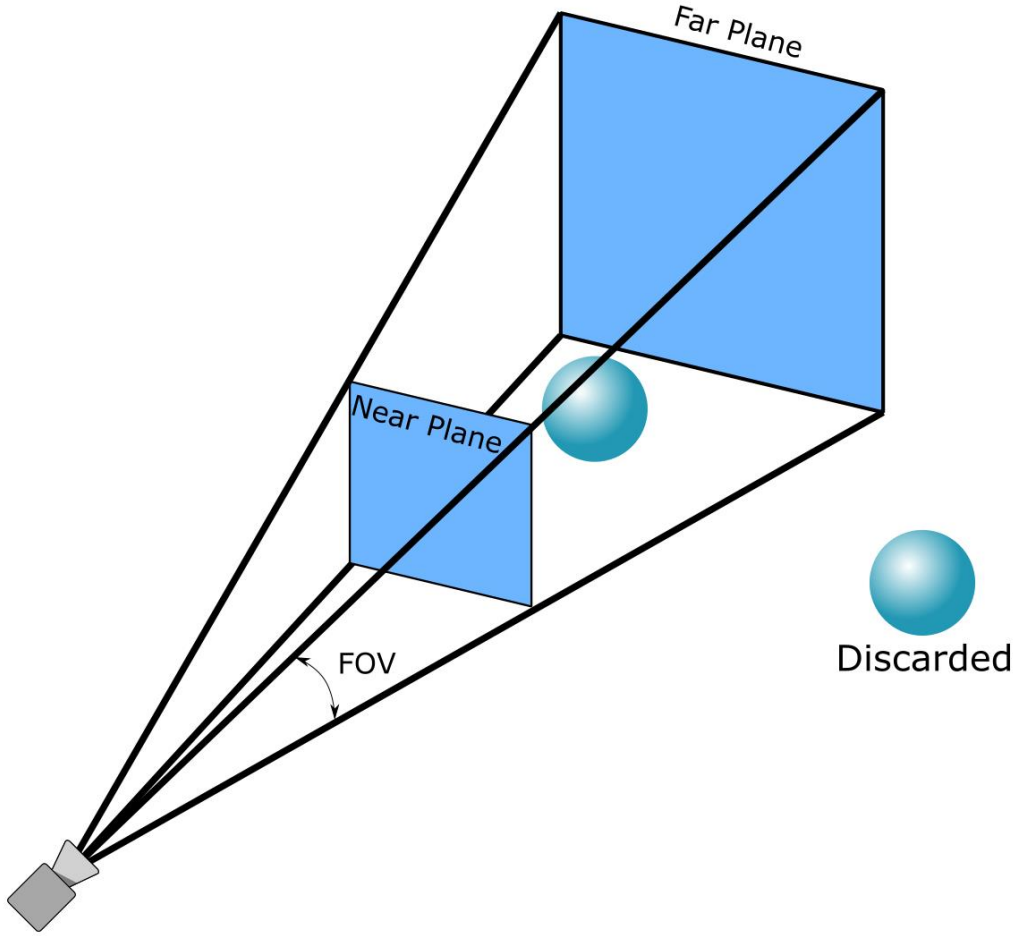
- Each component is divided by its w component
- A perspective projection matrix can be created in GLM as follows

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH
/ (float)SCR_HEIGHT, 0.01f, 100.0f);
```

- glm::perspective creates a large frustum that defines the visible space (outside clipped)

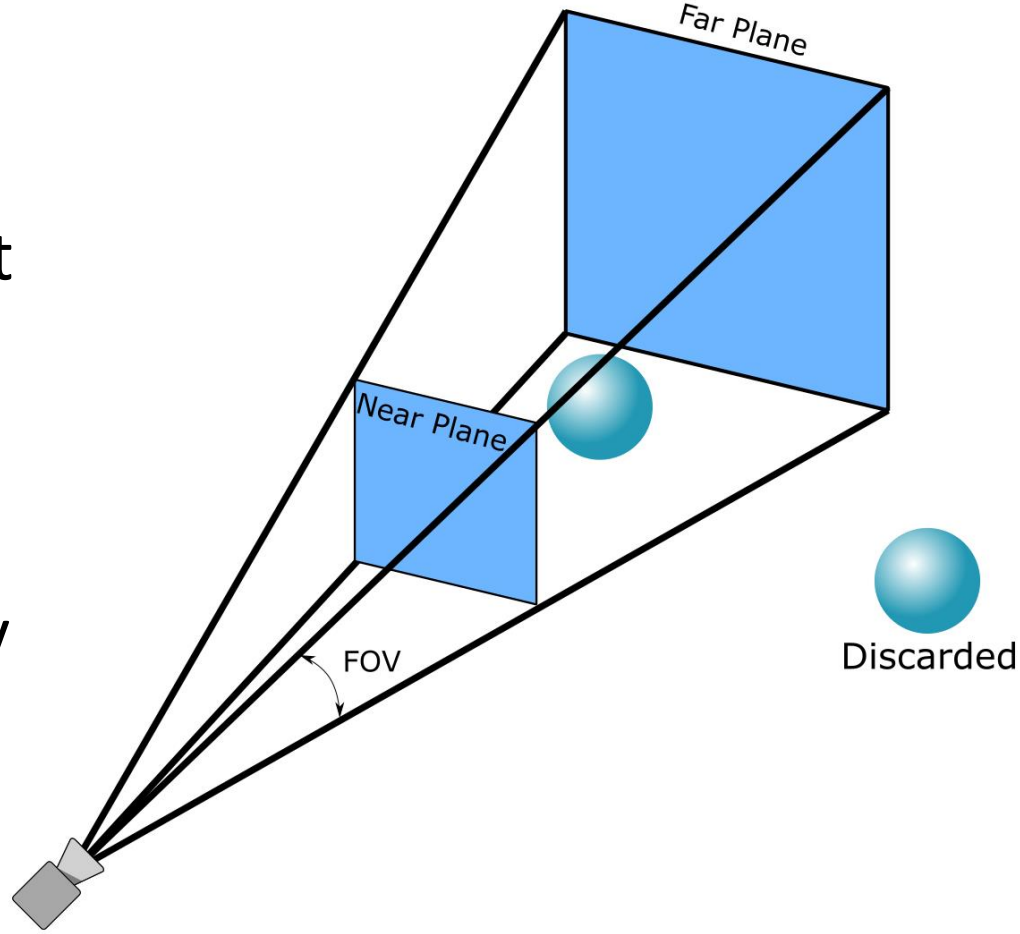
Perspective Projection

- A perspective frustum can be visualized as a non-uniformly shaped box from where each coordinate inside this box will be mapped to a point in clip space



Perspective Projection

- 1. parameter: fov (field of view) value and sets how large the viewspace is (usually set to 45°)
- 2. parameter: aspect ratio (calculated by dividing the viewport's width by its height)
- 3./4. parameter: near and far plane(usually 0.1f and 100.0f)
- All the vertices between the near and far plane and inside the frustum will be rendered




```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH
/ (float)SCR_HEIGHT, 0.01f, 100.0f);
```

Perspective Projection

Too high near values (like 10.0f) results in clipping coordinates close to the camera (between 0.0f and 10.0f)

Gives a familiar visual result in videogames in that you can see through certain objects if you move too close to them

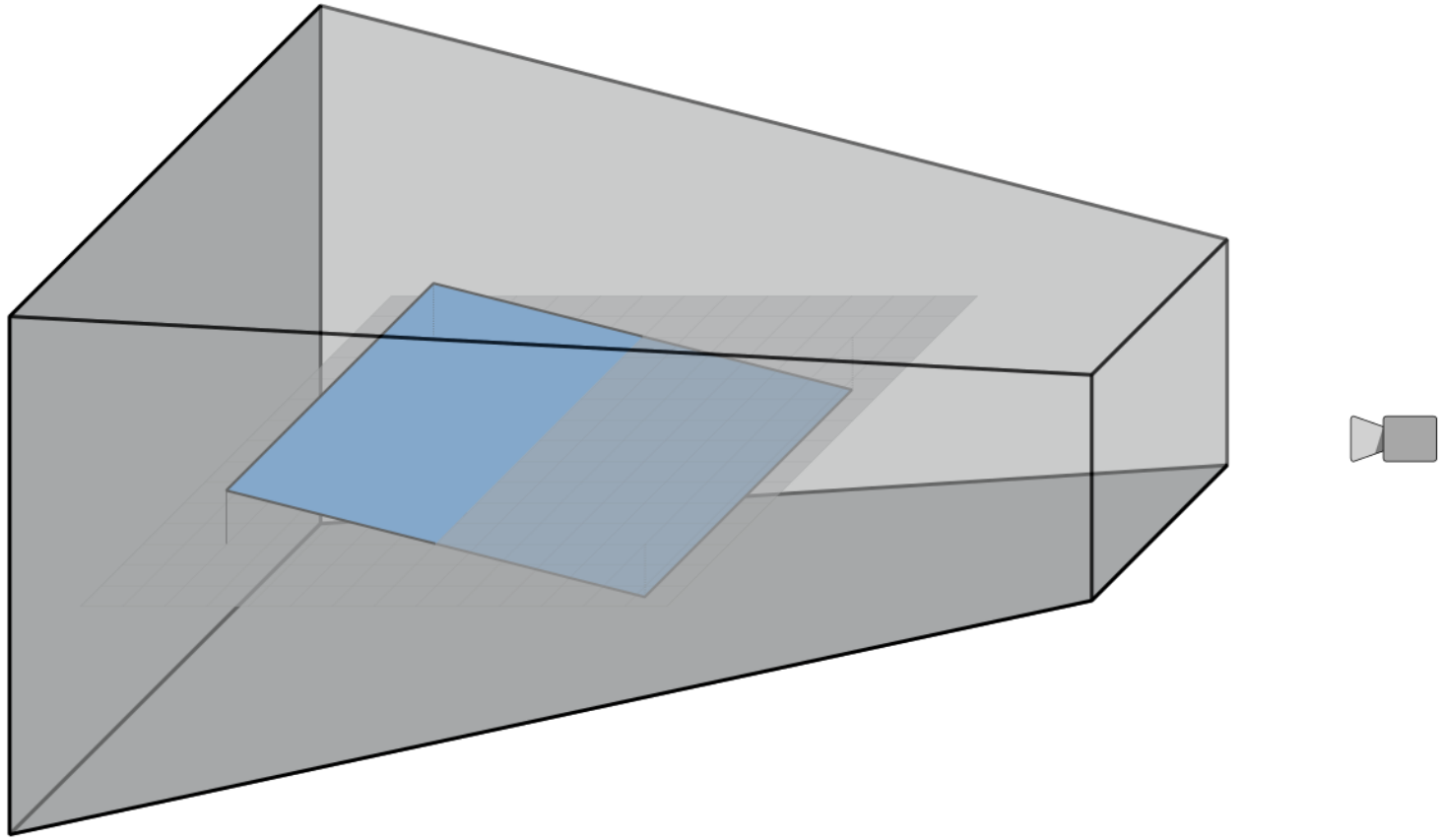
Perspective Projection

- Let us go back to our example:
  - We want translate, scale, and rotate the plane
  - We want to translate the midpoint to the origin: translate by $-\bar{p}$
  - Scale it by 5
  - Rotate around the x-axis by -30°
  - Translate it along the y-axis by 35 units

$$p_1 = \begin{pmatrix} 5 \\ 5 \\ 10 \end{pmatrix}, p_2 = \begin{pmatrix} 10 \\ 5 \\ 10 \end{pmatrix}, p_3 = \begin{pmatrix} 5 \\ 10 \\ 10 \end{pmatrix}, p_4 = \begin{pmatrix} 10 \\ 10 \\ 10 \end{pmatrix}$$

Perspective

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ -24.17 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ -24.17 \end{pmatrix}$$



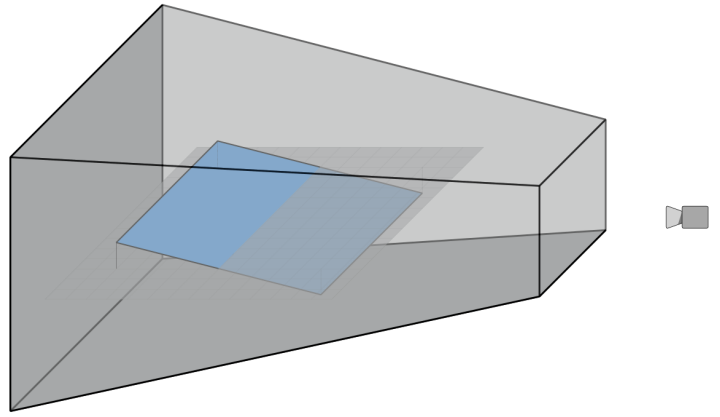
Perspective

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ -24.17 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ -24.17 \end{pmatrix}$$

- Projection matrix (aspect=800/600=4/3, fov=45°, far=-near=12):

$$proj = \begin{pmatrix} \frac{1}{aspect \cdot \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{4/3(\sqrt{2}-1)} & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}-1} & 0 & 0 \\ 0 & 0 & 0 & 12 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



Perspective

$$vmp_1 \approx \begin{pmatrix} -12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_2 \approx \begin{pmatrix} 12.5 \\ 6.25 \\ -45.83 \end{pmatrix}, vmp_3 \approx \begin{pmatrix} -12.5 \\ -6.25 \\ -24.17 \end{pmatrix}, vmp_4 \approx \begin{pmatrix} 12.5 \\ -6.25 \\ -24.17 \end{pmatrix}$$

- We get: $pvmp_i := proj \cdot vmp_i$

$$pvmp_1 \approx \begin{pmatrix} -22.63 \\ 15.09 \\ 12 \\ 45.82 \end{pmatrix}, pvmp_2 \approx \begin{pmatrix} 22.63 \\ 15.09 \\ 12 \\ 45.82 \end{pmatrix}, pvmp_3 \approx \begin{pmatrix} -22.63 \\ -15.09 \\ 12 \\ 24.17 \end{pmatrix}, pvmp_4 \approx \begin{pmatrix} 22.63 \\ -15.09 \\ 12 \\ 24.17 \end{pmatrix}$$

Perspective

$$pvmp_1 \approx \begin{pmatrix} -22.63 \\ 15.09 \\ 12 \\ 45.82 \end{pmatrix}, pvmp_2 \approx \begin{pmatrix} 22.63 \\ 15.09 \\ 12 \\ 45.82 \end{pmatrix}, pvmp_3 \approx \begin{pmatrix} -22.63 \\ -15.09 \\ 12 \\ 24.17 \end{pmatrix}, pvmp_4 \approx \begin{pmatrix} 22.63 \\ -15.09 \\ 12 \\ 24.17 \end{pmatrix}$$

- In clip space the coordinates are finally divided by w:

$$outp_1 \approx \begin{pmatrix} -0.49 \\ 0.33 \\ 0.26 \end{pmatrix}, outp_2 \approx \begin{pmatrix} 0.49 \\ 0.33 \\ 0.26 \end{pmatrix}, outp_3 \approx \begin{pmatrix} -0.94 \\ -0.62 \\ 0.50 \end{pmatrix}, outp_4 \approx \begin{pmatrix} 0.94 \\ -0.62 \\ 0.50 \end{pmatrix}$$

Perspective Projection

- All together:

```
glm::mat4 model= glm::mat4(1.0f);
glm::mat4 view= glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);

model = glm::translate(model, glm::vec3(0.0f, -35.0f, 0.0f));
model = glm::rotate(model, glm::radians(-30.0f), glm::vec3(1.0, 0.0, 0.0));
model = glm::scale(model, glm::vec3(5.0));
model = glm::translate(model, glm::vec3(-7.5f, -7.5f, -10.0f));

glm::vec4 r1 = glm::vec4(1,0,0,0);
glm::vec4 r2 = glm::vec4(0, 0, 1, 0);
glm::vec4 r3 = glm::vec4(0, 1, 0, 0);
glm::vec4 r4 = glm::vec4(0, 0, 0, 1);
view = glm::mat4(r1, r2, r3, r4);

projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH /
(float)SCR_HEIGHT, -12.0f, 12.0f);
```

F5

- …done

More 3D

Introduction

- So far working with a 2D plane in 3D space
- Now extend it to a 3D cube
- To render a cube, need a total of 36 vertices (6 faces * 2 triangles * 3 vertices each)

```
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};
```

Introduction

- Now, we want to rotate the cube:

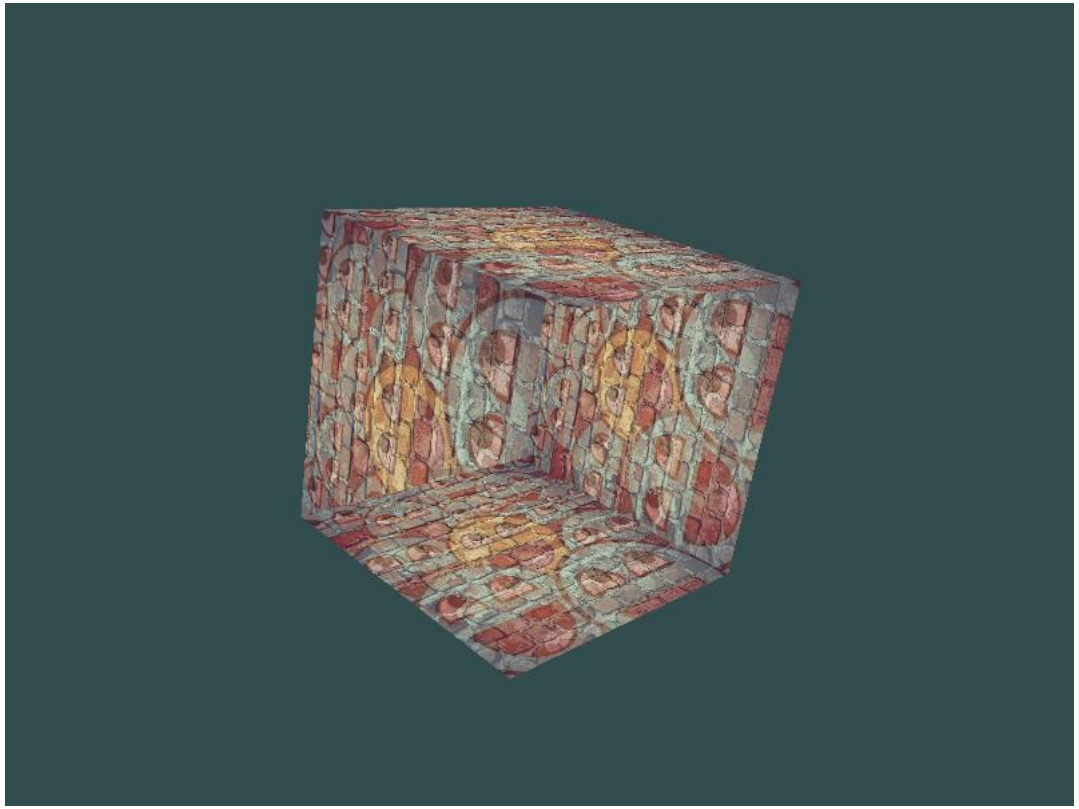
```
model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
```

- And draw the cube using glDrawArrays, but this time with a count of 36 vertices:

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

F5

- A rotating cube



Introduction

- Some sides of the cubes are being drawn over other sides
- This happens because when OpenGL draws the cube triangle-by-triangle, it will overwrite its pixels even though something else might’ve been drawn there before
- Because of this, some triangles are drawn on top of each other while they are not supposed to overlap

Introduction

- Luckily, OpenGL stores depth information in a buffer called the z-buffer that allows OpenGL to decide when to draw over a pixel and when not to
- Using the z-buffer we can configure OpenGL to do depth-testing

Z–Buffer

- GLFW automatically creates such a buffer
- The depth is stored within each fragment (z value) and 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
- This process is called depth testing and is done automatically by OpenGL

Z-Buffer

- To make sure OpenGL actually performs the depth testing we need to enable it (disabled by default) by using glEnable
- glEnable and glDisable functions allow to enable/disable certain functionalities in OpenGL
- That functionality is then enabled/disabled until another call is made to disable/enable it
- To enable depth testing by enabling GL_DEPTH_TEST:

```
glEnable(GL_DEPTH_TEST);
```

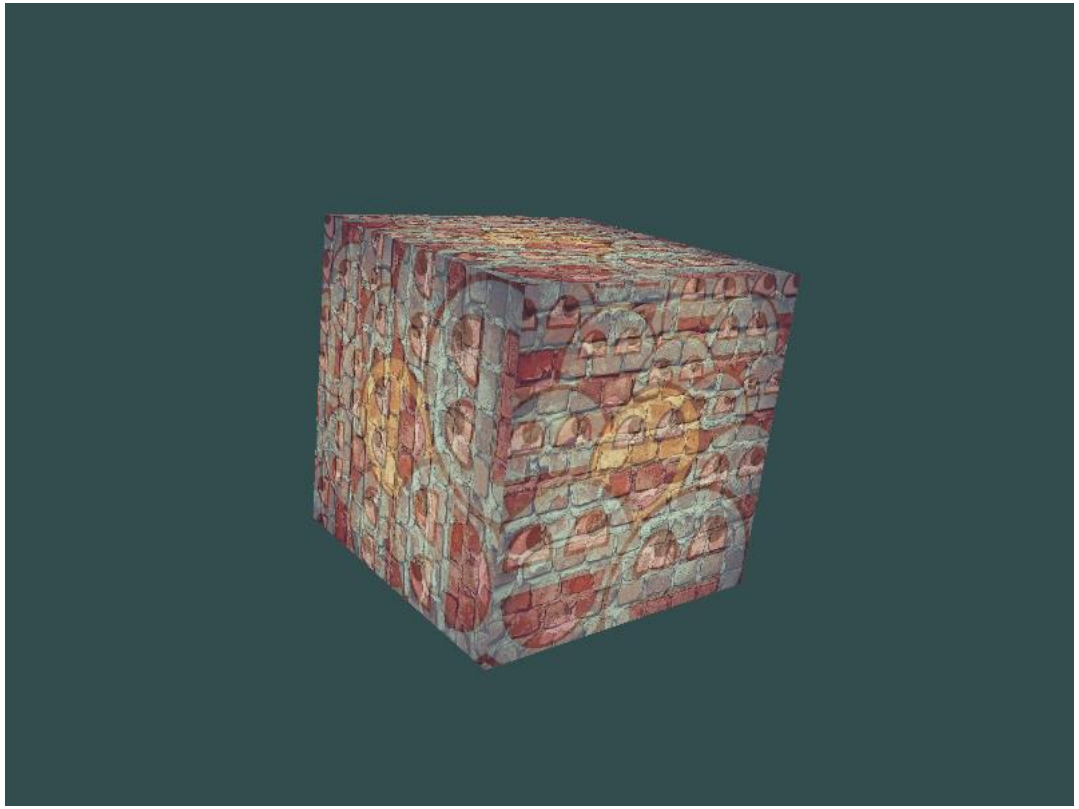
Z-Buffer

- By using a depth buffer, also want to clear the depth buffer before each render iteration (otherwise the depth information of the previous frame stays in the buffer)
- Just like clearing the color buffer, clear the depth buffer by specifying the DEPTH_BUFFER_BIT bit in the glClear function:

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

F5

- A rotating cube with correct depth



More Cubes

- Let's display 10 cubes on screen with different positions and rotations
- Only thing we have to change for each object is its model matrix
- First, define a translation vector for each cube that specifies its position in world space.
- Define 10 cube positions in 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)
    };
```

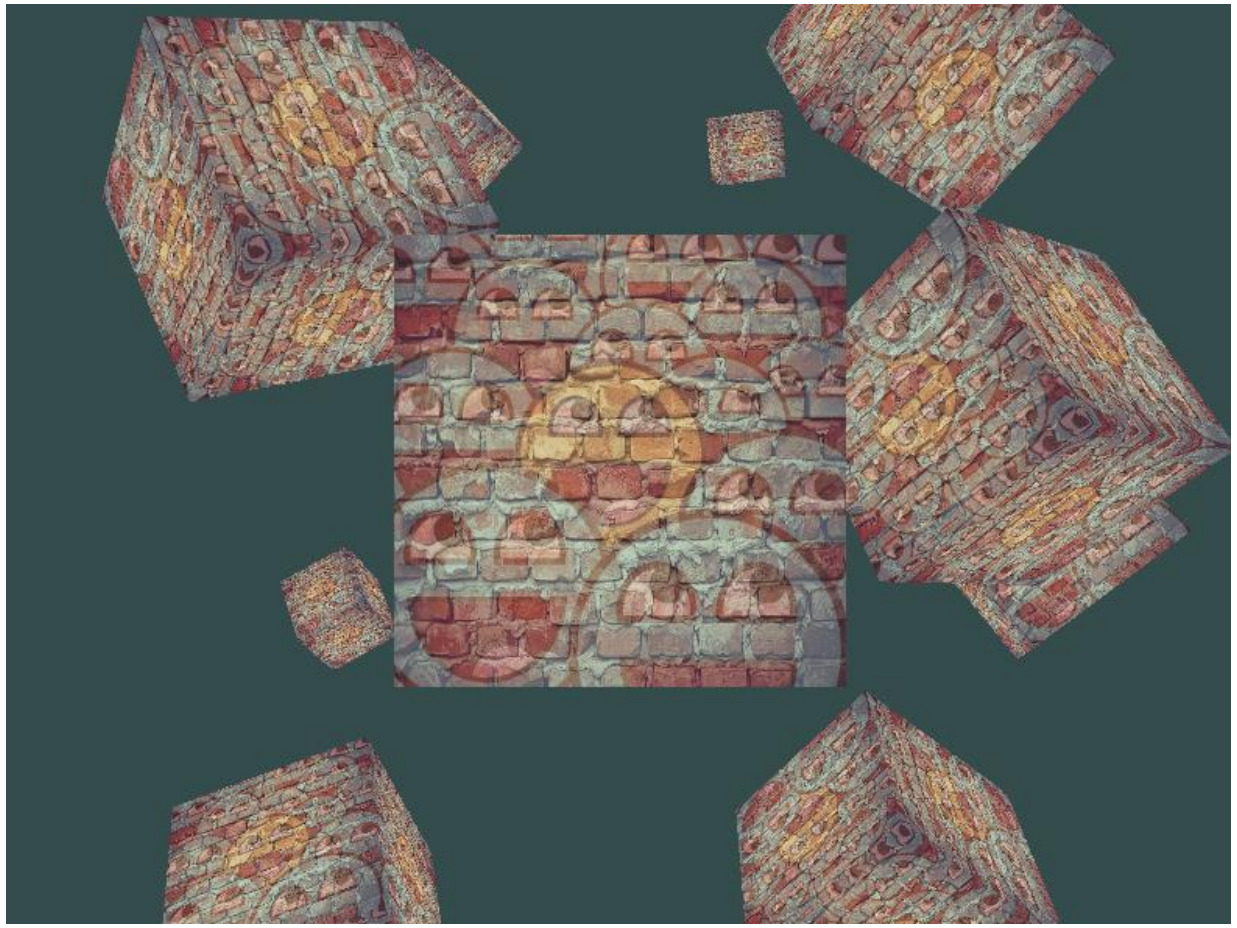
More Cubes

- Within the game loop call the glDrawArrays function 10 times, but each with a different model matrix

```
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);
}
```

F5

- … so we are many now!



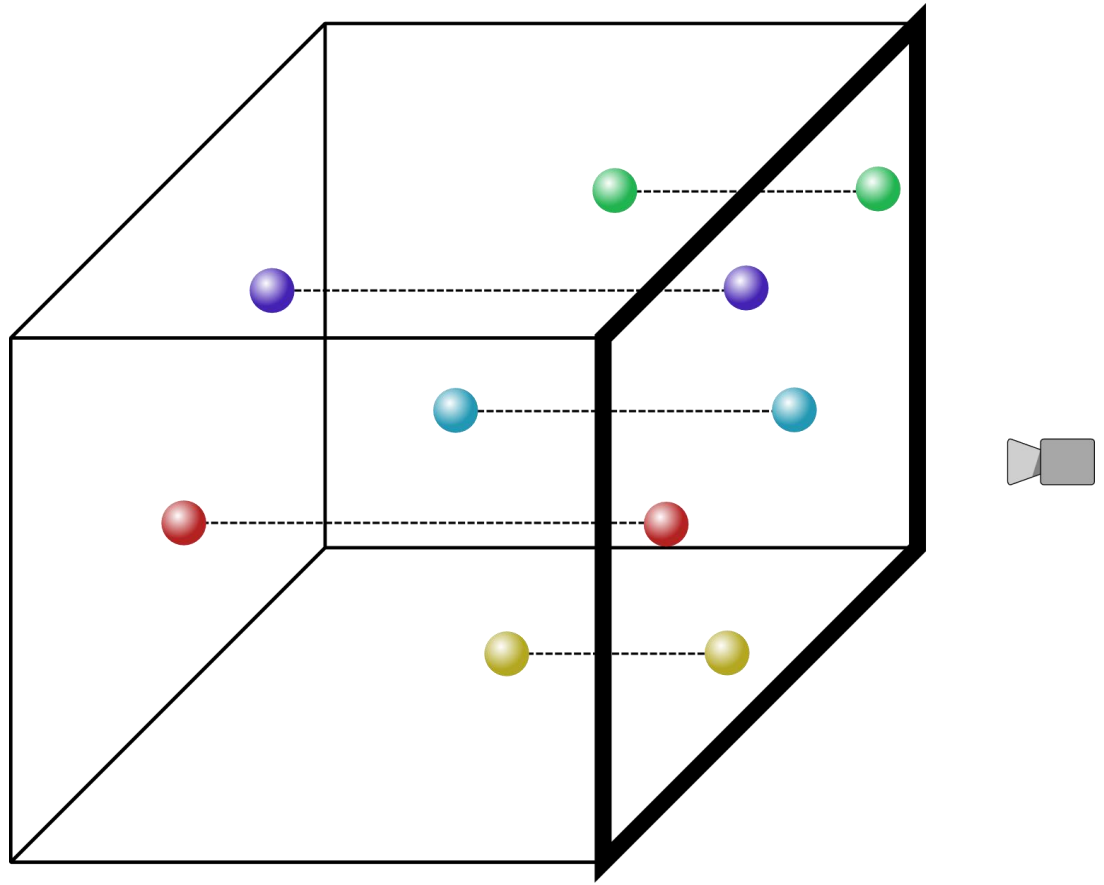
Projections*

Introduction

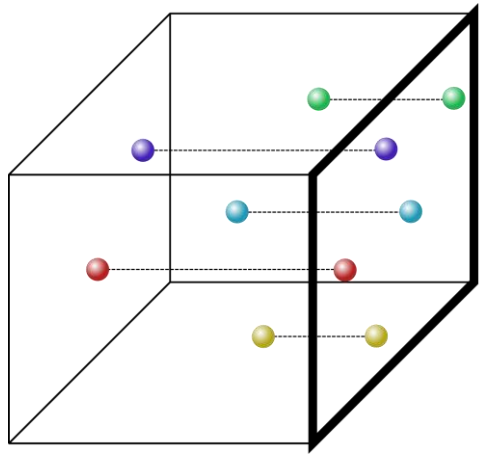
- Prerequisite: Camera in origin, view along the (negative) z-axis
- Goal: 2D coordinates in the view plane
- Near and far defined along z-axis
- Orthographic:
  - The z-values are of no relevance
- Perspective:
  - The Pixel values result from the intercept theorem (later)

Orthographic

- We have parallel rays from the point on the view plane
- With a given point, what are the NDC in the view plane?
- Again, the box is defined with left, right; bottom, top; near, far



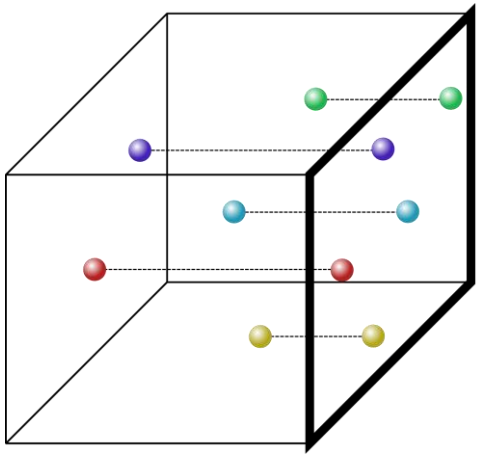
Orthographic



- Exemplarily, we have a closer look a the x-coordinate
- If the x component is equal the left value, then it should become -1
  - $NDC_X(left) = -1$
- If the x component is equal the right value, then it should become 1
  - $NDC_X(right) = 1$
- In between it should be linearized
  - $NDC_X(x) = mx + n$

Orthographic

$$NDC_X(left) = -1$$
$$NDC_X(right) = 1$$
$$NDC_X(x) = mx + n$$

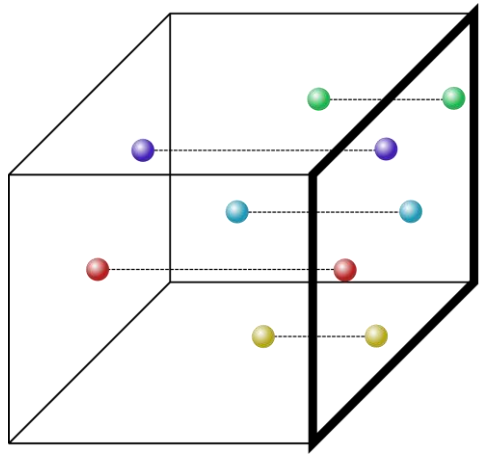


$$m = \frac{1 - (-1)}{right - left}$$
$$= \frac{2}{right - left}$$

$$NDC_X(left) = -1 \Rightarrow n = -1 - \frac{2 \cdot left}{right - left}$$
$$\Rightarrow n = \frac{-right + left - 2 \cdot left}{right - left}$$
$$\Rightarrow n = \frac{-right - left}{right - left}$$

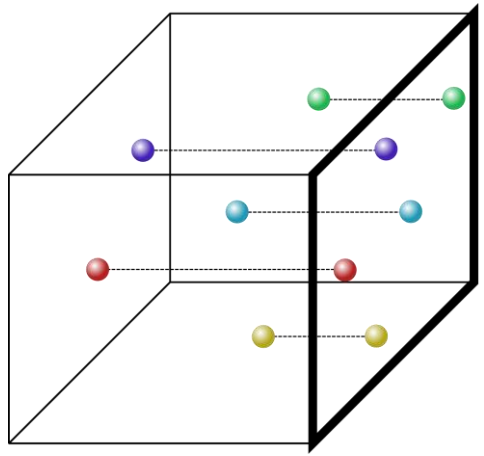
Orthographic

$$NDC_X(left) = -1$$
$$NDC_X(right) = 1$$
$$NDC_X(x) = mx + n$$



$$NDC_X(x) = \frac{2}{right - left}x - \frac{right + left}{right - left}$$

Orthographic

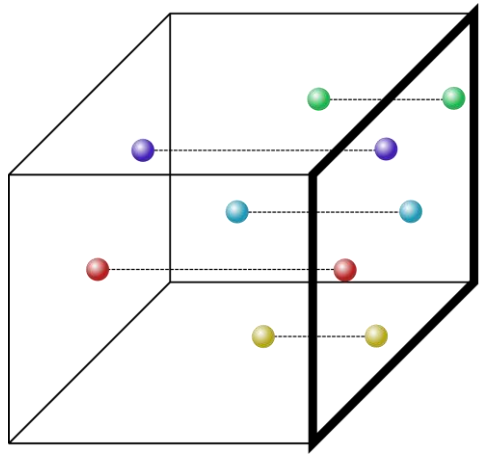


$$NDC_X(x) = \frac{2}{right - left}x - \frac{right + left}{right - left}$$

$$NDC_Y(x) = \frac{2}{top - bottom}x - \frac{top + bottom}{top - bottom}$$

$$NDC_Z(x) = \frac{-2}{far - near}x - \frac{far + near}{far - near}$$

Orthographic



$$NDC_X(x) = \frac{2}{right - left}x - \frac{right + left}{right - left}$$

$$NDC_Y(x) = \frac{2}{top - bottom}x - \frac{top + bottom}{top - bottom}$$

$$NDC_Z(x) = \frac{-2}{far - near}x - \frac{far + near}{far - near}$$

$$NDC_Z(-near) = -1$$

$$[z_{min}, z_{max}] \rightarrow [-near, -far]$$

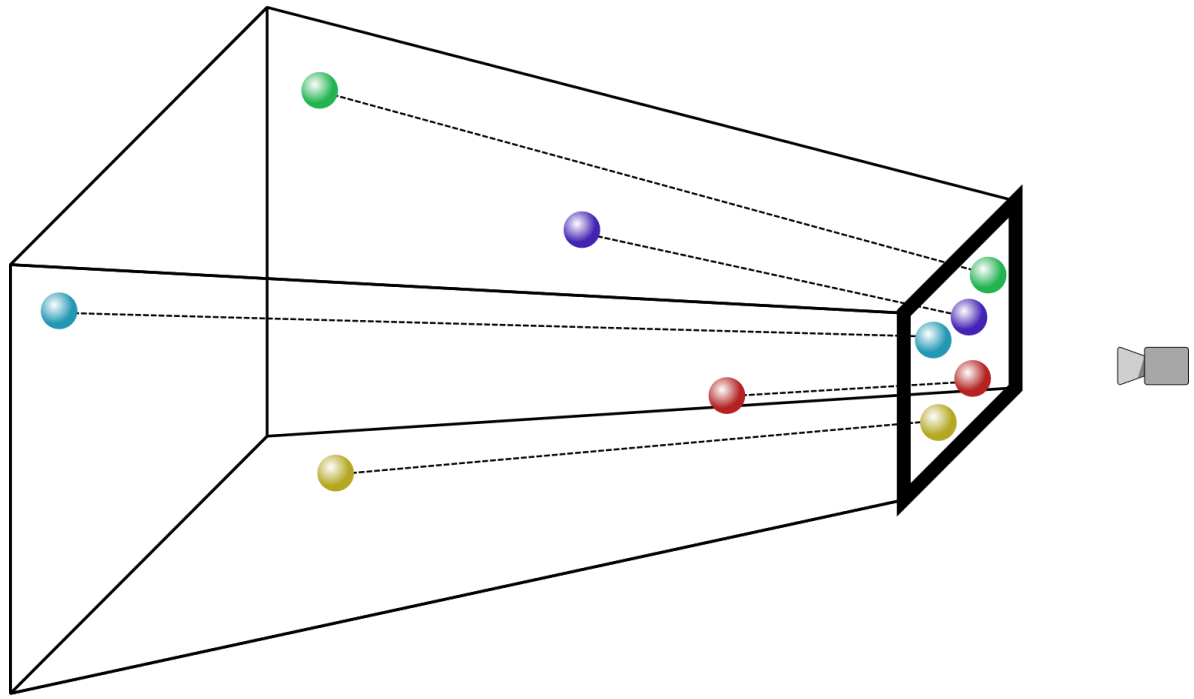
Orthographic

$$NDC_X(x) = \frac{2}{right - left}x - \frac{right + left}{right - left}$$
$$NDC_Y(x) = \frac{2}{top - bottom}x - \frac{top + bottom}{top - bottom}$$
$$NDC_Z(x) = \frac{-2}{far - near}x - \frac{far + near}{far - near}$$

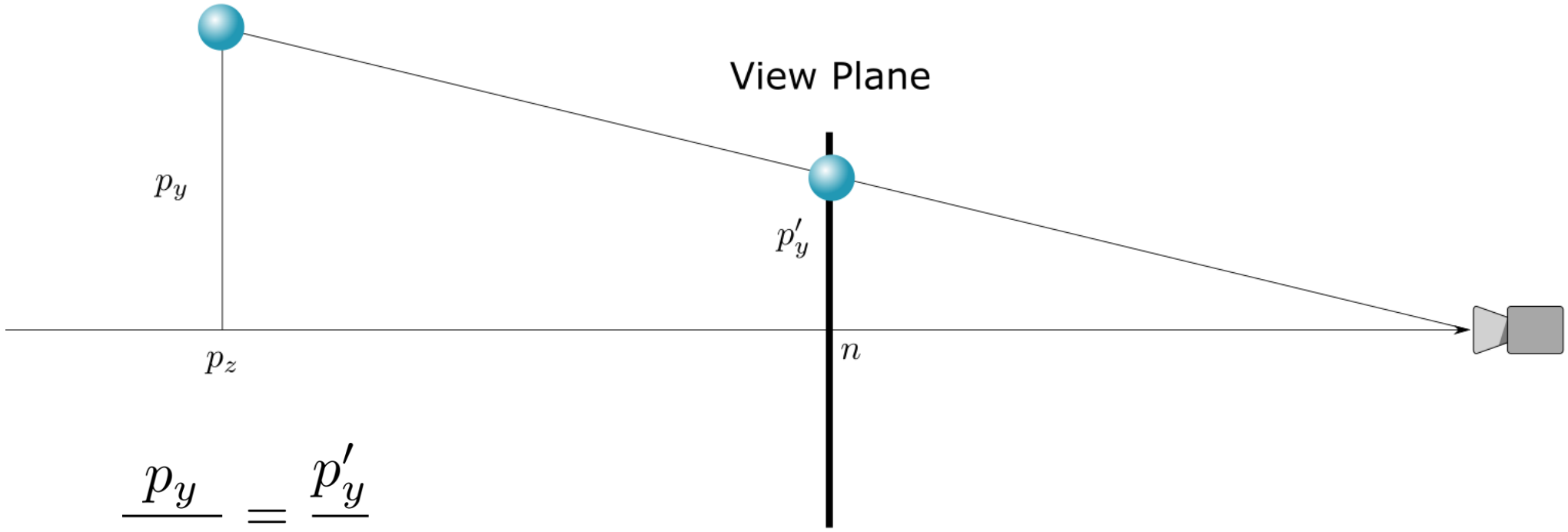
$$proj = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective

- We have rays starting from one origin (camera)
- With a given point, what are the NDC in the view plane?
- Again, the box is defined with left, right; bottom, top; near, far



Perspective



$$\frac{p_y}{-p_z} = \frac{p'_y}{n}$$

$$p'_y = n \cdot \frac{p_y}{-p_z}$$

Perspective

$$\frac{p_y}{-p_z} = \frac{p'_y}{n}$$

$$p'_y = n \cdot \frac{p_y}{-p_z}$$

- Note, that we want the coordinates to be in NDC [-1,1] and that the values are divided by p_w at the end

$$NDC_Y(x) = \frac{2}{top - bottom}x - \frac{top + bottom}{top - bottom}$$

$$NDC_Y(-n \cdot p_y/p_z) = \frac{2}{top - bottom} \cdot \frac{n \cdot p_y}{-p_z} - \frac{top + bottom}{top - bottom}$$

$$= \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_y}{top - bottom} + \frac{p_z \cdot (top + bottom)}{top - bottom}\right)$$

Perspective

$$NDC_X(p_x, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_x}{right - left} + \frac{p_z \cdot (right + left)}{right - left}\right)$$

$$NDC_Y(p_y, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_y}{top - bottom} + \frac{p_z \cdot (top + bottom)}{top - bottom}\right)$$

Perspective

$$NDC_X(p_x, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_x}{right - left} + \frac{p_z \cdot (right + left)}{right - left}\right)$$

$$NDC_Y(p_y, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_y}{top - bottom} + \frac{p_z \cdot (top + bottom)}{top - bottom}\right)$$

- Dividing the values by w is not possible with a matrix, thus, the projection matrix is of the form

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & ? & ? \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective

$$NDC_X(p_x, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_x}{right - left} + \frac{p_z \cdot (right + left)}{right - left}\right)$$

$$NDC_Y(p_y, p_z) = \frac{1}{-p_z}\left(\frac{2 \cdot n \cdot p_y}{top - bottom} + \frac{p_z \cdot (top + bottom)}{top - bottom}\right)$$

- Dividing the values by w is not possible with a matrix, thus, the projection matrix is of the form

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective

$NDC_Z(-n)/n = -1$
$NDC_Z(-f)/f = 1$
$NDC_Z(z) = (Az + B)/(-z)$

$$NDC_Z(-n) = \frac{-A \cdot n + B}{n} = -1$$

$$NDC_Z(-f) = \frac{-A \cdot f + B}{f} = 1$$

$$\Rightarrow -A \cdot n + n = -A \cdot f - f$$

$$\Rightarrow A = -\frac{f + n}{f - n}$$

$$\Rightarrow B = f + A \cdot f = -\frac{2 \cdot f \cdot n}{f - n}$$

Perspective

- Finally:

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective

$$proj = \begin{pmatrix} \frac{2n}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2n}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2 \cdot f \cdot n}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

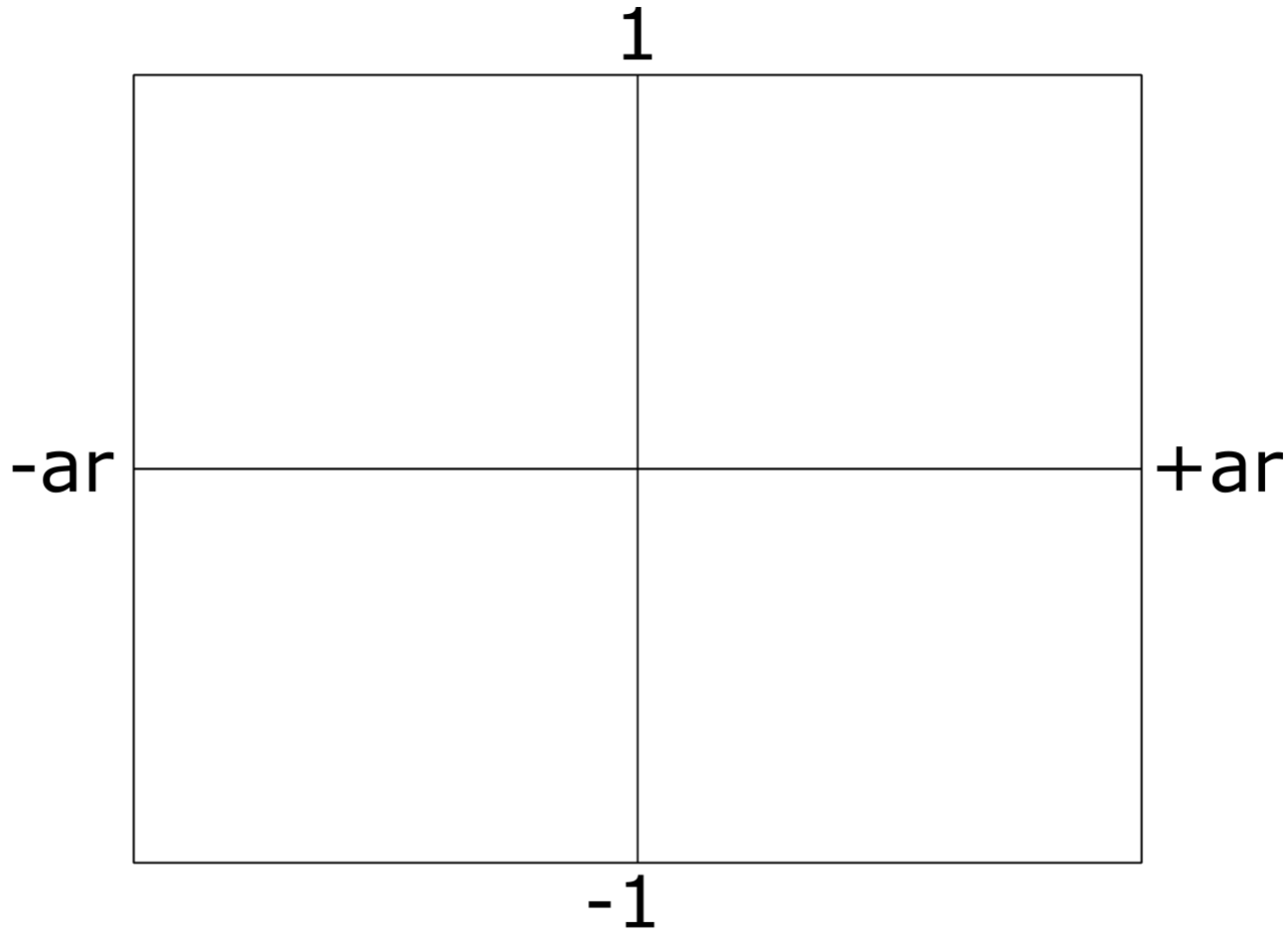
- …but we knew another representation?

$$proj = \begin{pmatrix} \frac{1}{aspect \cdot \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective

- View plane

$$ar = \frac{width}{height}$$



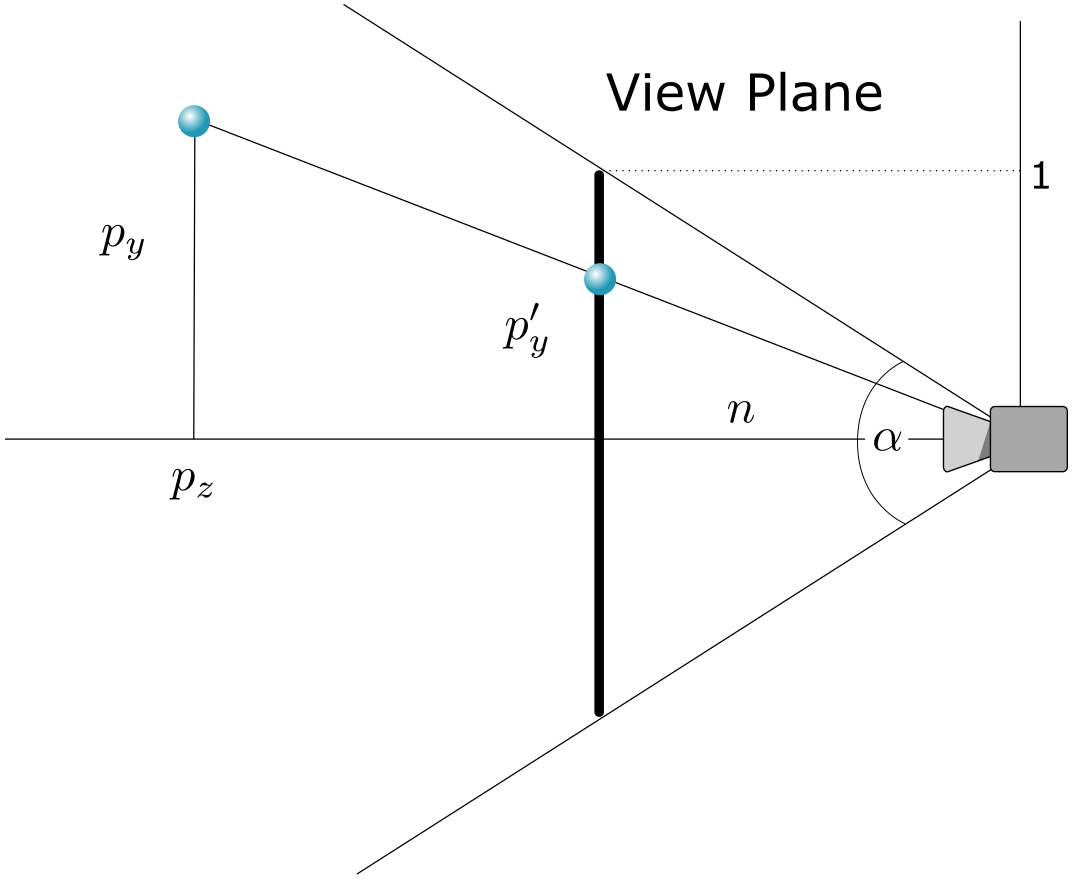
Perspective

$$\tan(\alpha/2) = \frac{1}{n} \Rightarrow n = \frac{1}{\tan(\alpha/2)}$$

$$\frac{p_y}{-p_z} = \frac{p'_y}{n}$$

$$p'_y = -n \cdot \frac{p_y}{p_z}$$

$$= \frac{p_y}{-p_z \cdot \tan(\alpha/2)}$$

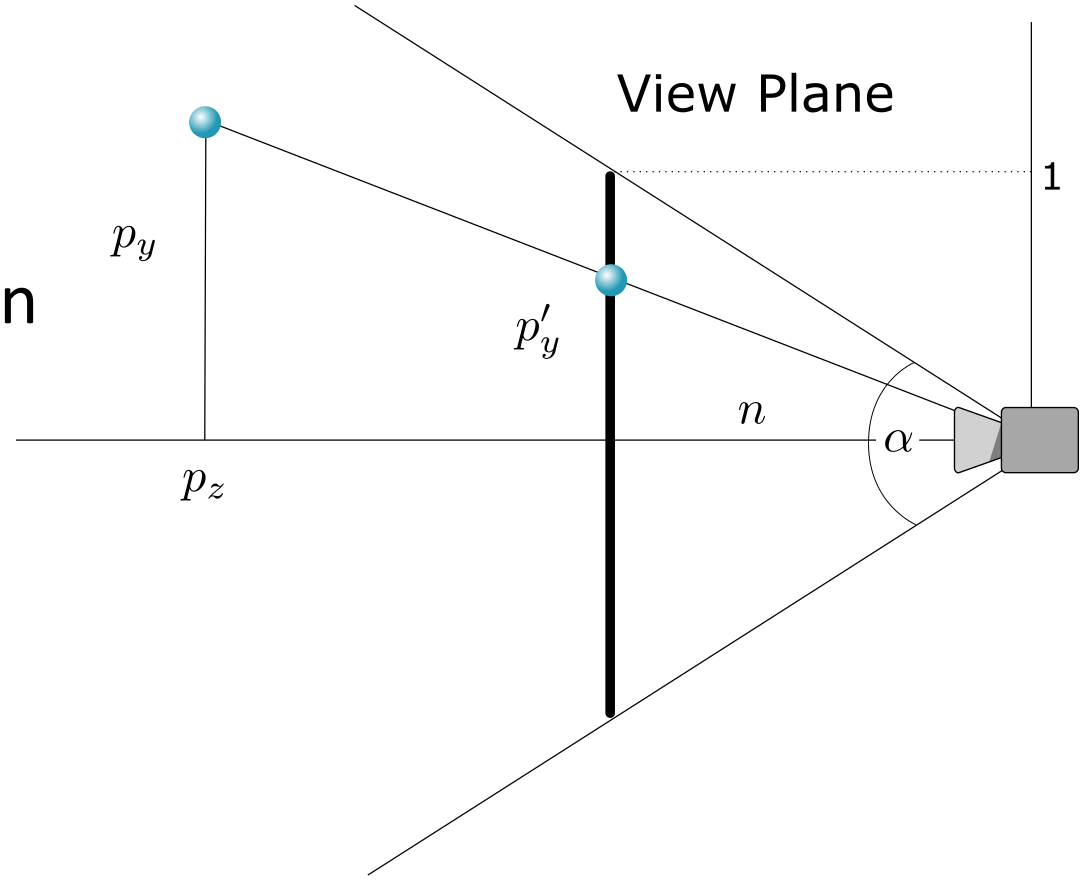


Perspective

- p'_y should be in the allowed interval $[-1,1]$ so that's ok
- p'_x should be in the allowed interval $[-ar, ar]$, we divide it by ar then it is in $[-1,1]$ so that's ok

$$p'_y = \frac{1}{-p_z} \cdot \frac{p_y}{\tan(\alpha/2)}$$

$$p'_x = \frac{1}{-p_z} \cdot \frac{p_x}{ar \cdot \tan(\alpha/2)}$$



Perspective

$$p'_x = \frac{1}{-p_z} \cdot \frac{p_x}{ar \cdot \tan(\alpha/2)}$$

$$p'_y = \frac{1}{-p_z} \cdot \frac{p_y}{\tan(\alpha/2)}$$

- This yields:

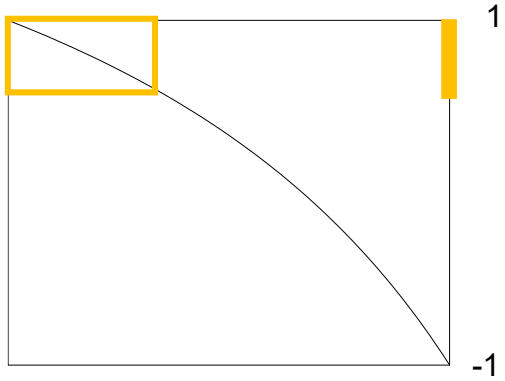
$$proj = \begin{pmatrix} \frac{1}{aspect \cdot \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

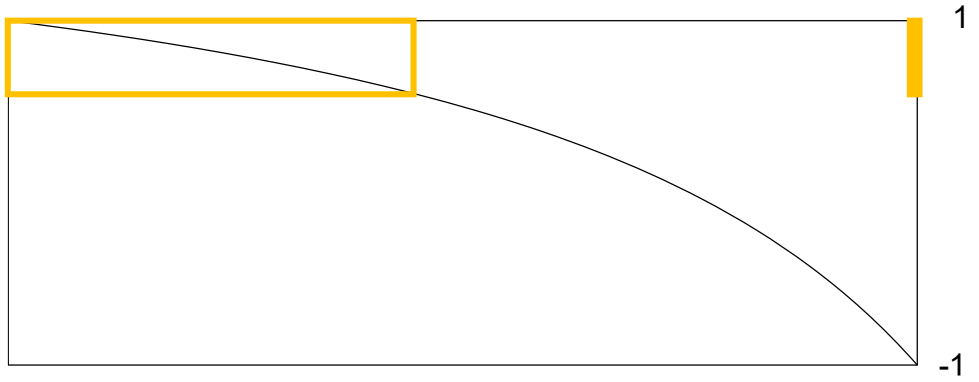
Depth

$$proj = \begin{pmatrix} \frac{1}{aspect \cdot \tan(fov/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(fov/2)} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

- Non-linear relationship of z' and z
- High values → little precision

$$proj \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow z' = \frac{f+n}{f-n} + \frac{2fn}{z(f-n)}$$





Questions???