(2) Mathematics

GPU Programming

Thorsten Grosch

TU Clausthal

Mathematics

- Today: repeat most important transformations and projections from Computer Graphics 1
- Show how they can be implemented with **glm**

- Transformations
- Projections
- Viewport

Transformations

Transformations

- The most important transformations can be implemented with 4x4 matrices (with homogeneous coordinates)
  - Translation, Rotation, Scaling
  - In OpenGL, we work (in most cases) with 4D vectors and 4x4 matrices
- In modern OpenGL, these matrices can not be generated, but we need an external library, e.g. glm

glm Matrices

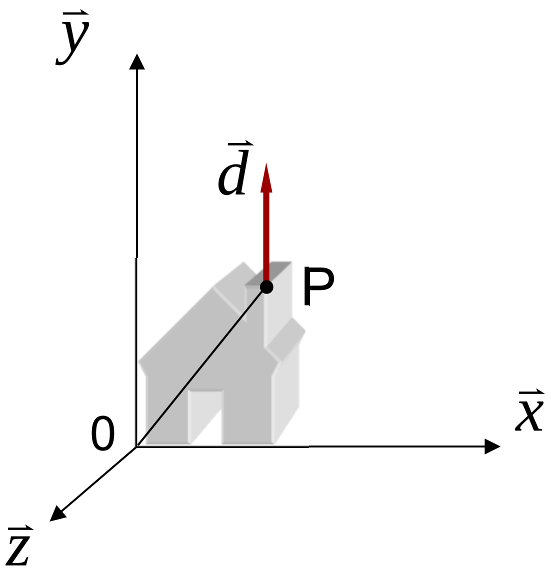
```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

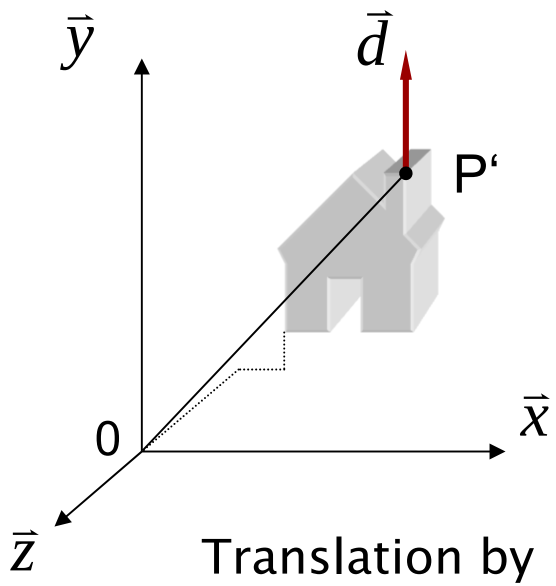
- Create unity matrix

```
mat4 matrix = mat4(1.0);

// create transformations…
```

Translation

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$





Translation by t_x, t_y, t_z

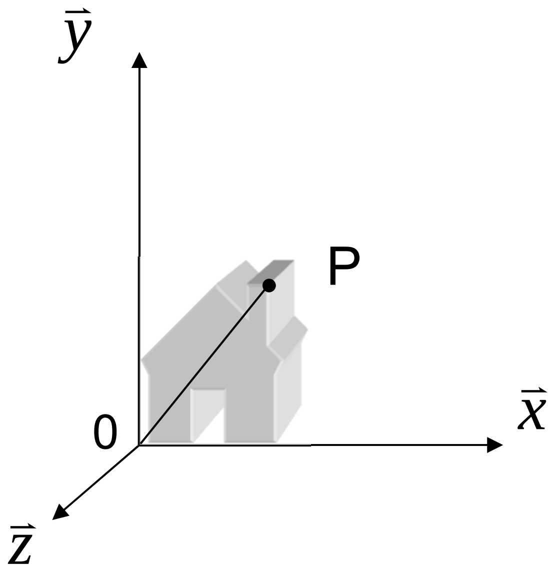
glm Translation Matrix

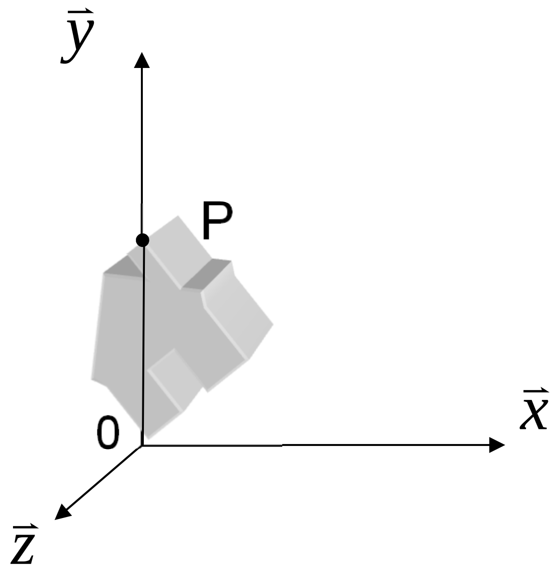
```
glm::mat4 glm::translate(glm::mat4& M,
                         glm::vec3& translation);
```

- Generates a translation matrix for the translation vector **`translation`**
- If the first parameter is set to the unity matrix (**`M = mat4(1.0)`**), we simply compute the translation matrix **`T`**
- Otherwise we compute the product **`M * T`**
- The translation matrix is multiplied from the right to the existing transformation

Rotation (z axis)

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$





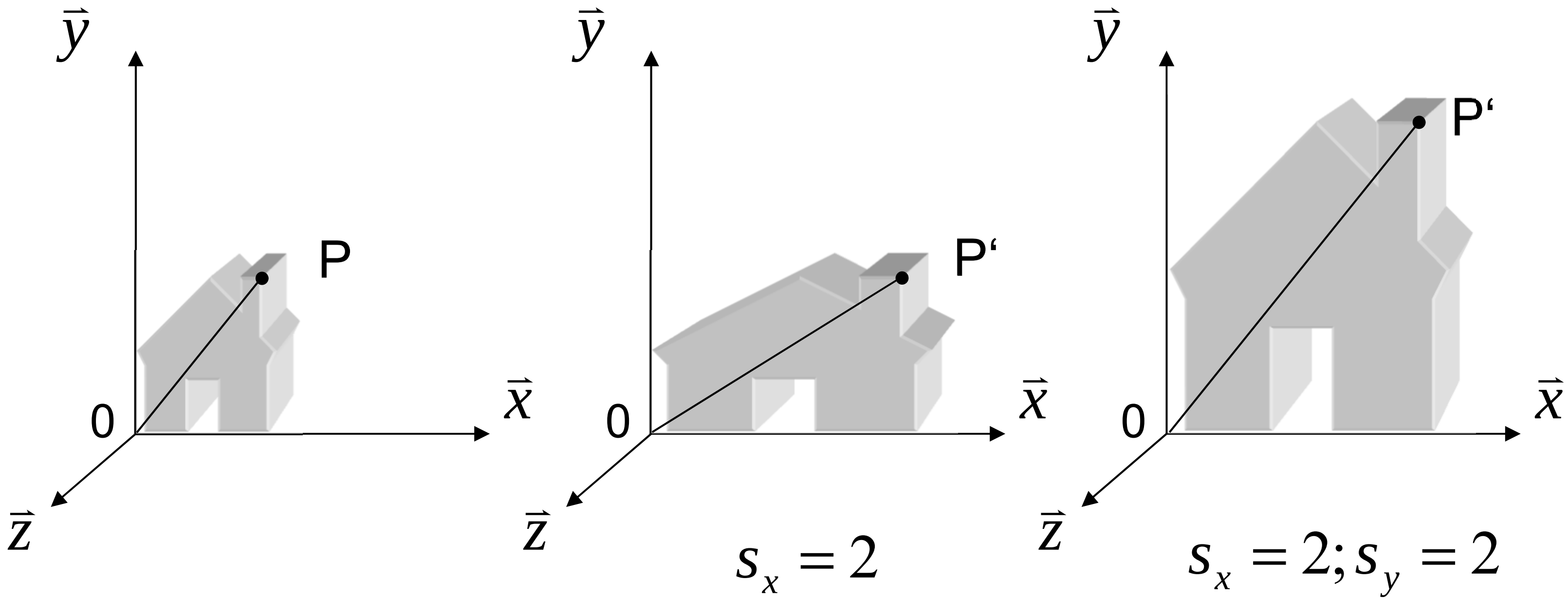
glm Rotation Matrix

```
glm::mat4 glm::rotate(glm::mat4& M, float angle,
                      glm::vec3& axis);
```

- Generates a rotation matrix for a rotation by the angle **`angle`** (in degrees) around the axis **`axis`**
- If the first parameter is set to the unity matrix (**`M = mat4(1.0)`**), we simply compute the rotation matrix **`R`**
- Otherwise we compute the product **`M * R`**
- The rotation matrix is multiplied from the right to the existing transformation

Scaling

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \end{pmatrix}$$



glm Scaling Matrix

```
glm::mat4 glm::scale(glm::mat4& M,
                     glm::vec3& scaleFactors);
```

- Generates a scaling matrix for the three scaling factors **`scaleFactors`**
- If the first parameter is set to the unity matrix (**`M = mat4(1.0)`**), we simply compute the scaling matrix **`S`**
- Otherwise we compute the product **`M * S`**
- The scaling matrix is multiplied from the right to the existing transformation

glm Matrices Combinations

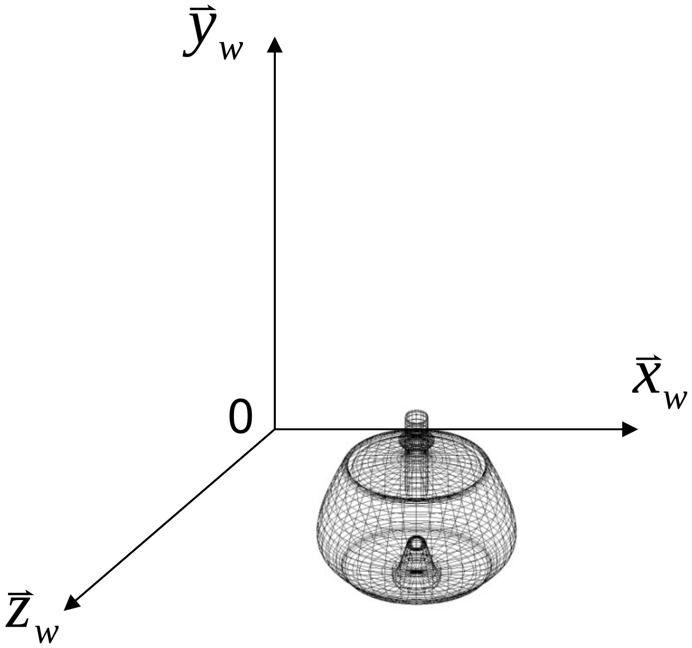
```
mat4 T = translate(mat4(1.0f), vec3(1.0f, 2.0f, 3.0f));
mat4 R = rotate(mat4(1.0f), 45.0f, vec3(1.0f, 0.0f, 0.0f));
mat4 S = scale(mat4(1.0f), vec3(2.0f, 2.0f, 2.0f));
mat4 model = T * R * S;
```

The same matrix can be created like this:

```
mat4 model = mat4(1.0f);
model = translate(model, vec3(1.0f, 2.0f, 3.0f));
model = rotate(model, 45.0f, vec3(1.0f, 0.0f, 0.0f));
model = scale(model, vec3(2.0f, 2.0f, 2.0f));
```

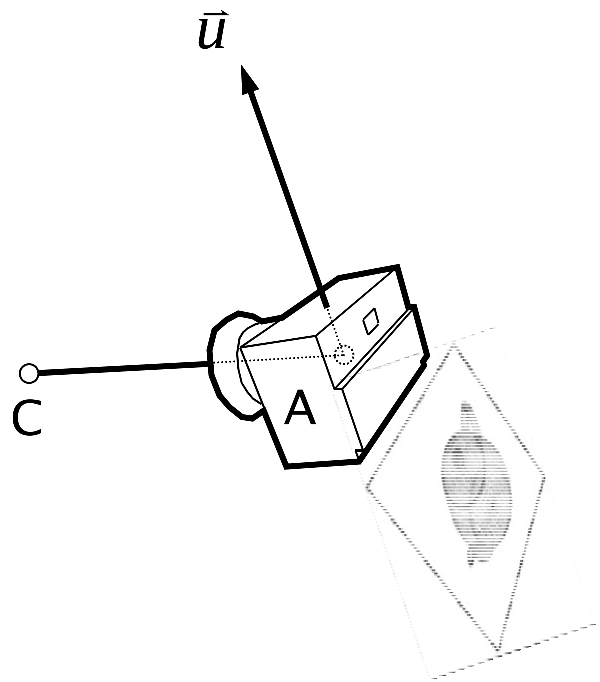
View Transformation

Camera: Position and Orientation



World coordinate system:

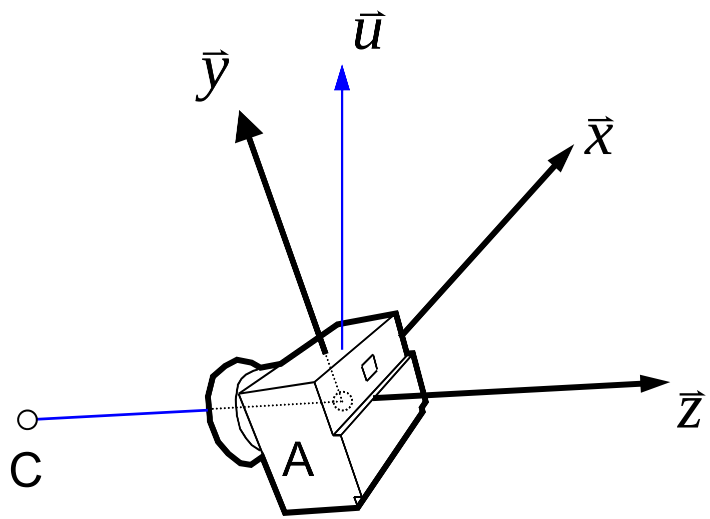
The world in which objects can be moved around



Definition of a camera:

Eye position A, center point C, vector pointing „up“

Camera Coordinate System



$$\vec{z} = \overrightarrow{CA}^0 = \frac{A-C}{|A-C|}$$

$$\vec{x} = \frac{\vec{u}\times\vec{z}}{|\vec{u}\times\vec{z}|}$$

$$\vec{y} = \vec{z}\times\vec{x}$$

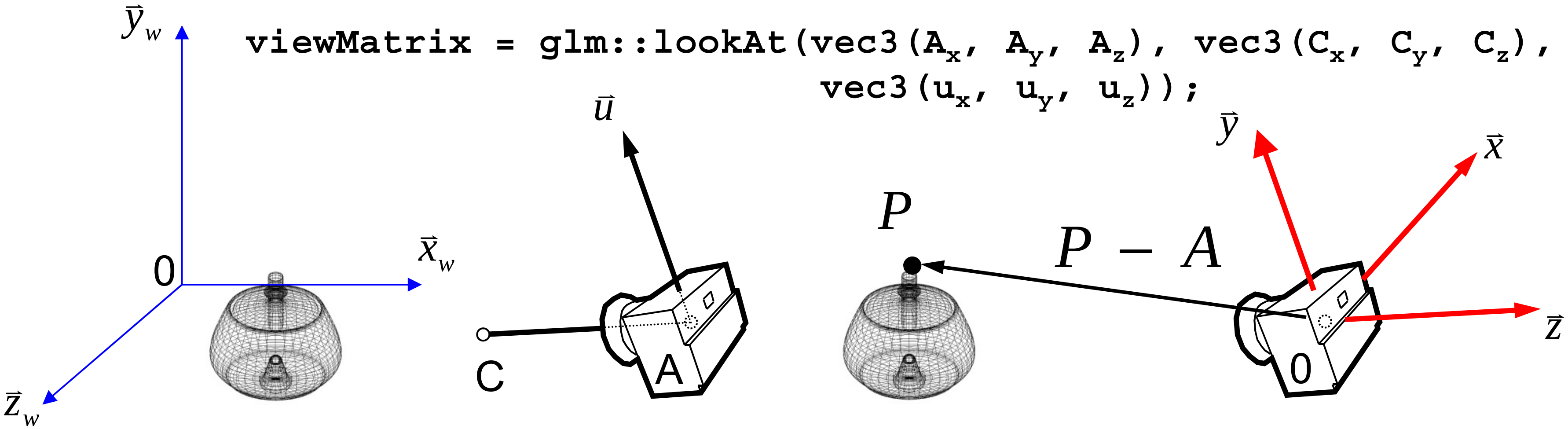
In general we have

$$\vec{y} \neq \vec{u}$$

- The camera looks along the negative z axis, the y axis points upwards and the x axis to the right (OpenGL)
- This is an ortho-normal right-handed coordinate system
- Transformation world coordinates → camera coordinates
- What happens if we look upwards (along u vector) ?

View Transformation

```
viewMatrix = glm::lookAt(vec3(Ax, Ay, Az), vec3(Cx, Cy, Cz),
                         vec3(ux, uy, uz));
```



Define Camera in glm with lookAt(…)

The camera coordinate system is then constructed from the parameters A, C, u (from, at, up) (see slide Camera Coordinate System)

The world coordinates are then transformed in camera coordinates

$$p_x = (P - A) \circ \vec{x}$$

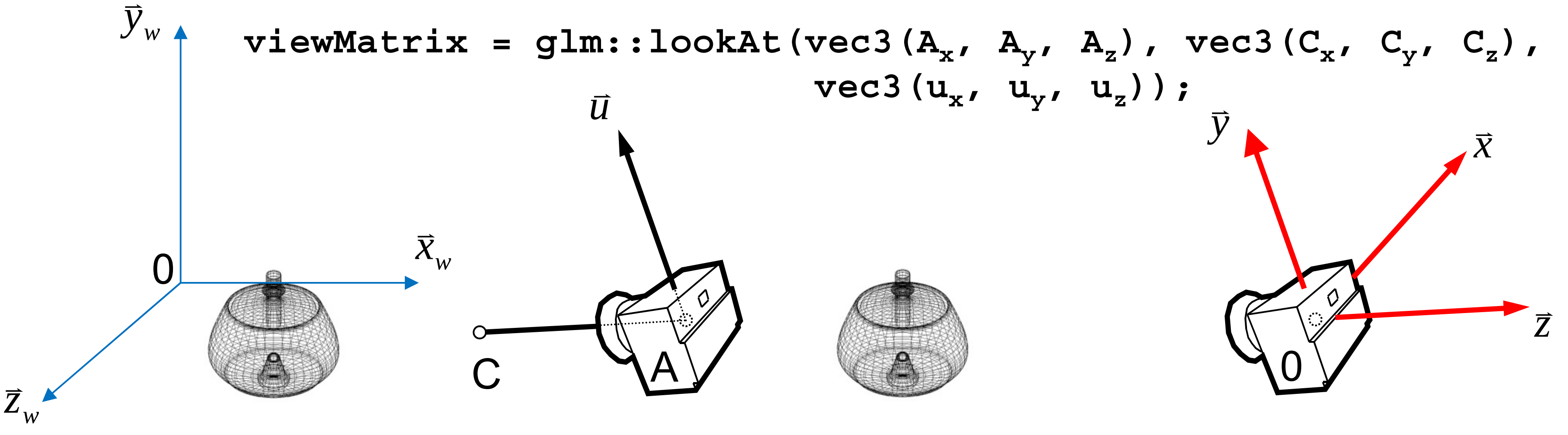
$$p_y = (P - A) \circ \vec{y}$$

$$p_z = (P - A) \circ \vec{z}$$

Camera World

View Transformation

```
viewMatrix = glm::lookAt(vec3(Ax, Ay, Az), vec3(Cx, Cy, Cz),
                          vec3(ux, uy, uz));
```



glm calculates the matrix V (=View) for this transformation

This Matrix can be used in the Vertex Shader

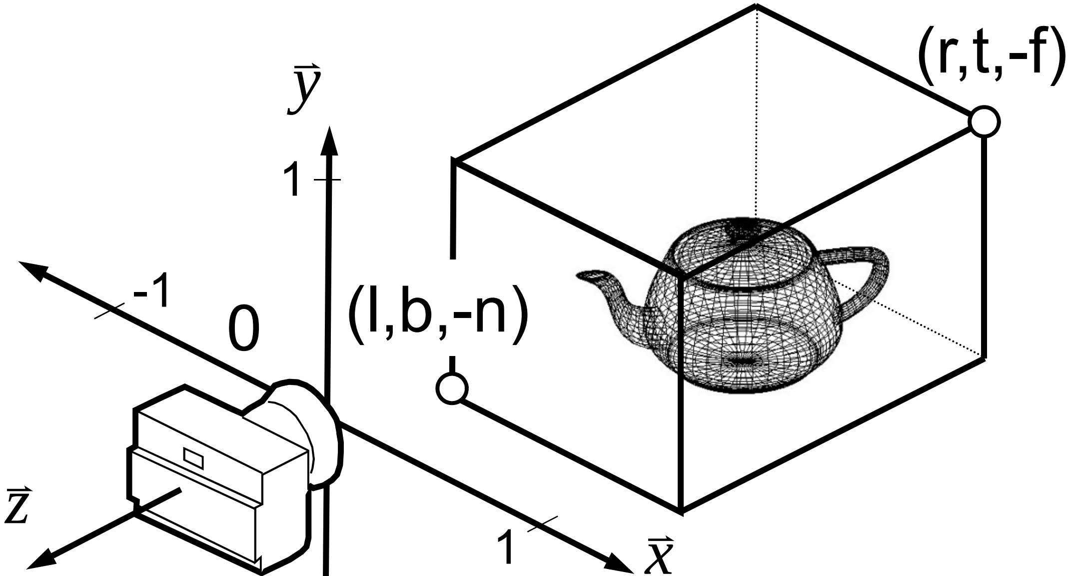
Do we multiply the V matrix from the left of from the right to the transformation matrix ?

$$V = \begin{pmatrix} x_{x_w} & x_{y_w} & x_{z_w} & 0 \\ y_{x_w} & y_{y_w} & y_{z_w} & 0 \\ z_{x_w} & z_{y_w} & z_{z_w} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -A_x \\ 0 & 1 & 0 & -A_y \\ 0 & 0 & 1 & -A_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Base vectors as rows

Orthographic Projection

Parallel Projection



„Orthographic Volume“

$$(x_{\min}, x_{\max}) \triangleq (left, right)$$

$$(y_{\min}, y_{\max}) \triangleq (bottom, top)$$

$$(z_{\min}, z_{\max}) \triangleq (-near, -far)$$

-1

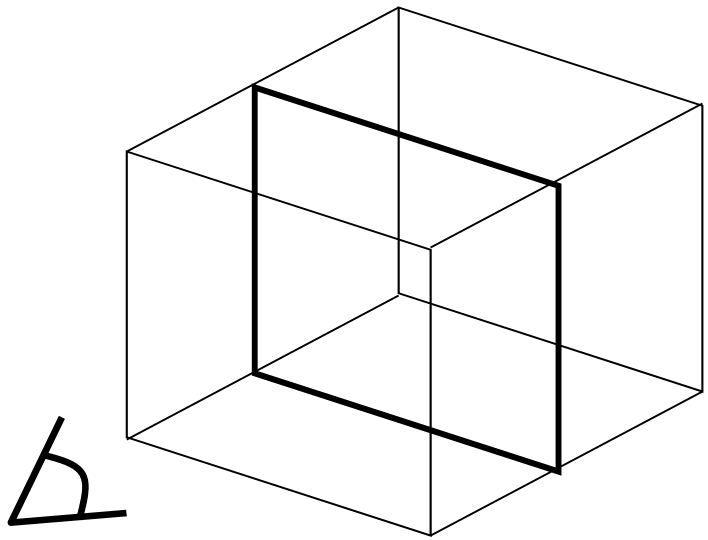
```
projectionMatrix =
glm::ortho(left, right, bottom, top, near, far);
```

e.g. ortho(-1, 1, -1, 1, 3, 10);

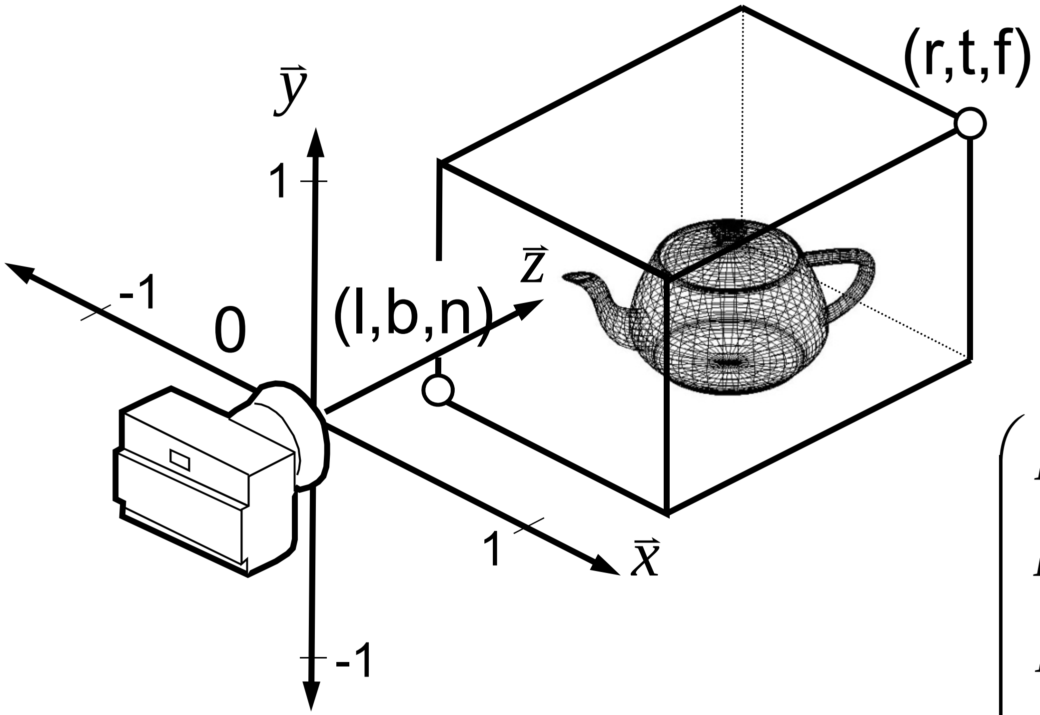
- ortho() defines a cuboid (a box).
- Parallel projection is applied to all points inside the cuboid
- Points outside the cuboid are not drawn
- The projection is applied along the z axis onto the plane z = −n (near plane)
- The parameters left, right, bottom, top are in camera coordinates
- The viewing direction is therefore perpendicular to the near plane
- Attention: The sign is inverted for near and far !

Parallel Projection

- A matrix can be used for an orthographic projection
- The projection is applied in three steps



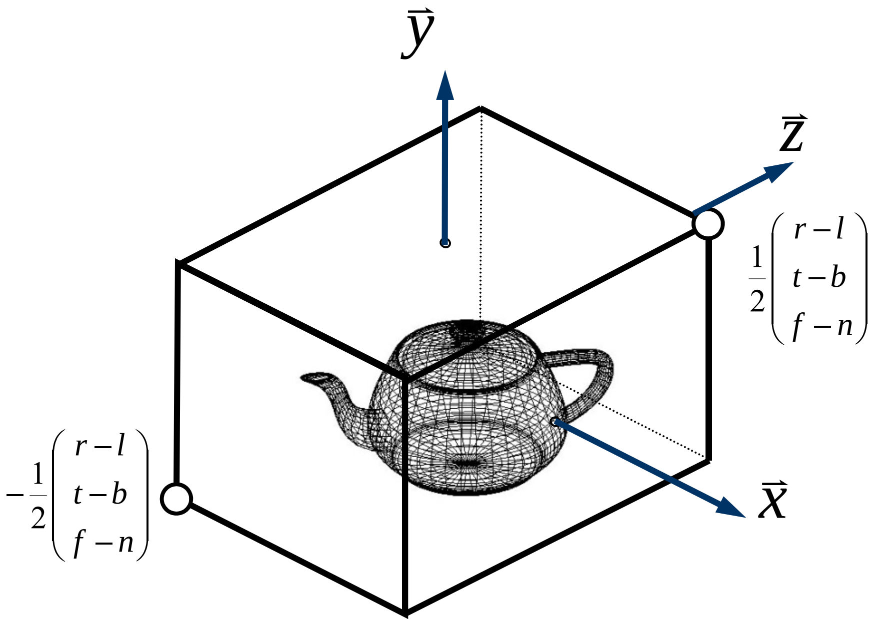
Parallel Projection



1. Negate the z coordinate
(right handed → left handed coordinate system)

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{R \to L}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Parallel Projection

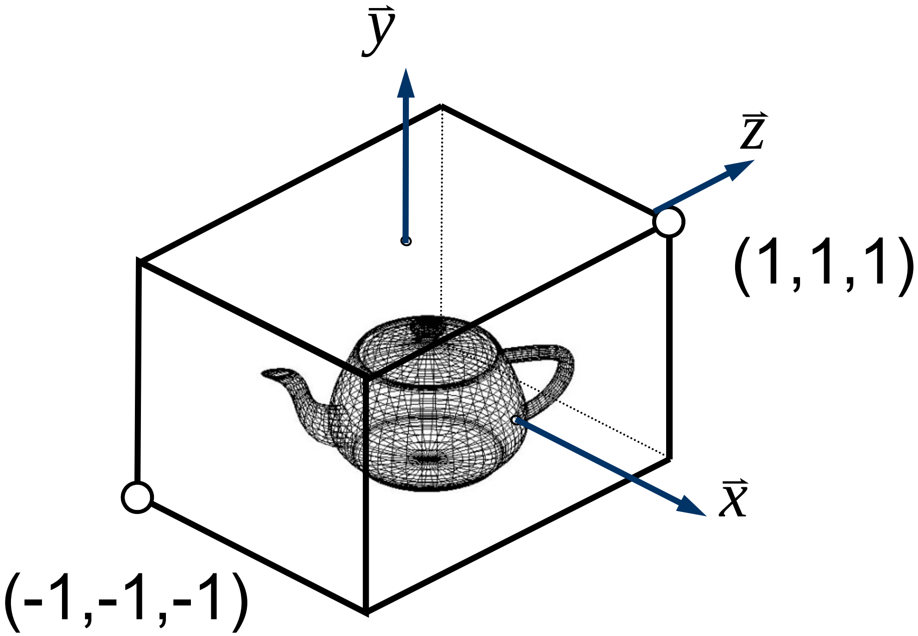


2. Translation: Move center of the volume into the origin

$$l + \frac{1}{2}(r-l) = \frac{l+r}{2}$$

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Parallel Projection



The „canonical" Volume

`ortho(…)` transforms the original cuboid in a unit cube around the origin of the camera coordinate system

3. Scale along the axes

$$\frac{p'_x}{p_x} = \frac{2}{(r-l)}$$

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix for Orthographic Projection

$$\underbrace{\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{ORTHO}} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{R \to L}}$$

Matrix for Orthographic Projection

Multiplying the three matrices results in:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{b+t}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
projectionMatrix = glm::ortho(left, right, bottom, top, near, far);
```

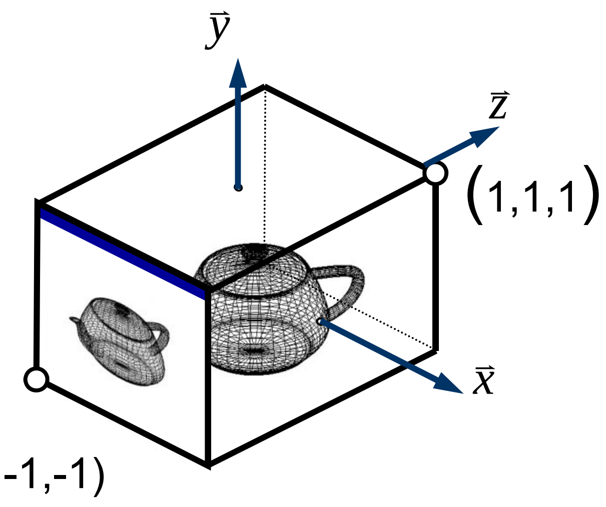
Transformation: Point in camera coordinates → Point in canonical volume

Sequence of Matrices

Afterwards, transformation in camera coordinate system

First, arbitrary affine transformations

$$\vec{p}\,' = \underbrace{M_{ORTHO} \cdot M_{R \to L}}_{M_{PROJECTION}} \cdot \underbrace{V \cdot T \cdot \ldots \cdot S \cdot T \cdot R}_{M_{MODELVIEW}} \cdot \vec{p}$$



After the orthographic projection:

- Object lies around the coordinate origin
- view along the z axis (left handed)
- display volume (–1,1)x(–1,1)x(–1,1)

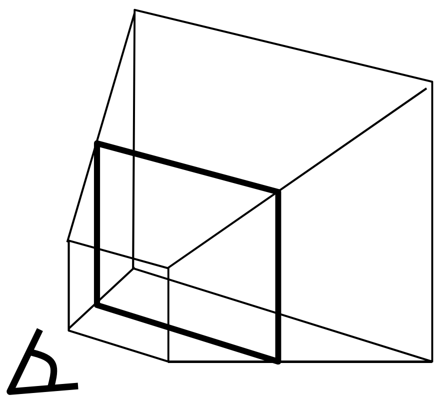
Afterwards: Viewport–Transformation (later)

- only for xy values
- z values for depth buffer

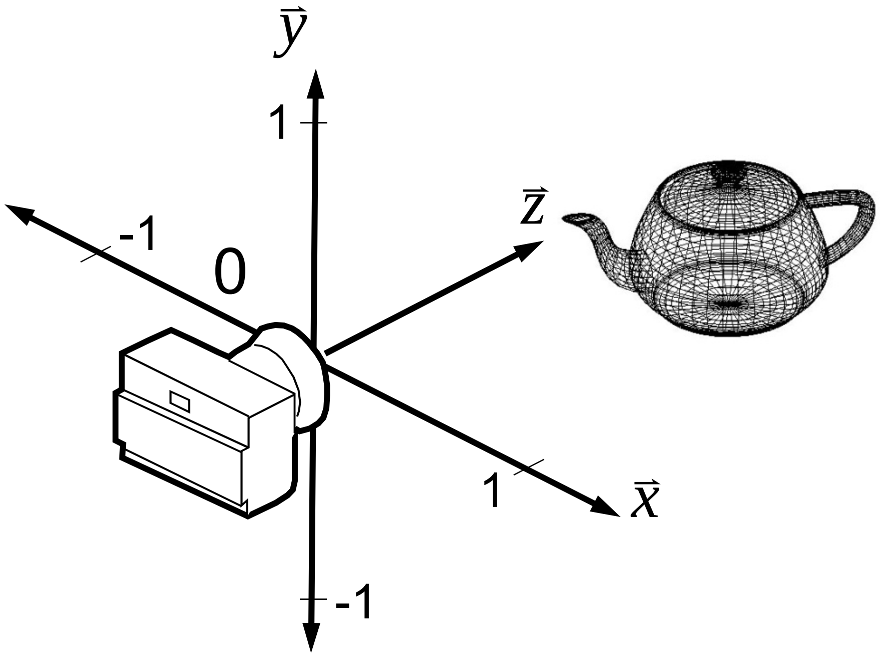
Perspective Projection

Perspective Projection in OpenGL

- For the perspective projection a matrix is used
- Multiple steps
- More difficult than parallel projection
- A perspective transformation is used, afterwards a parallel projection leads to a perspective image



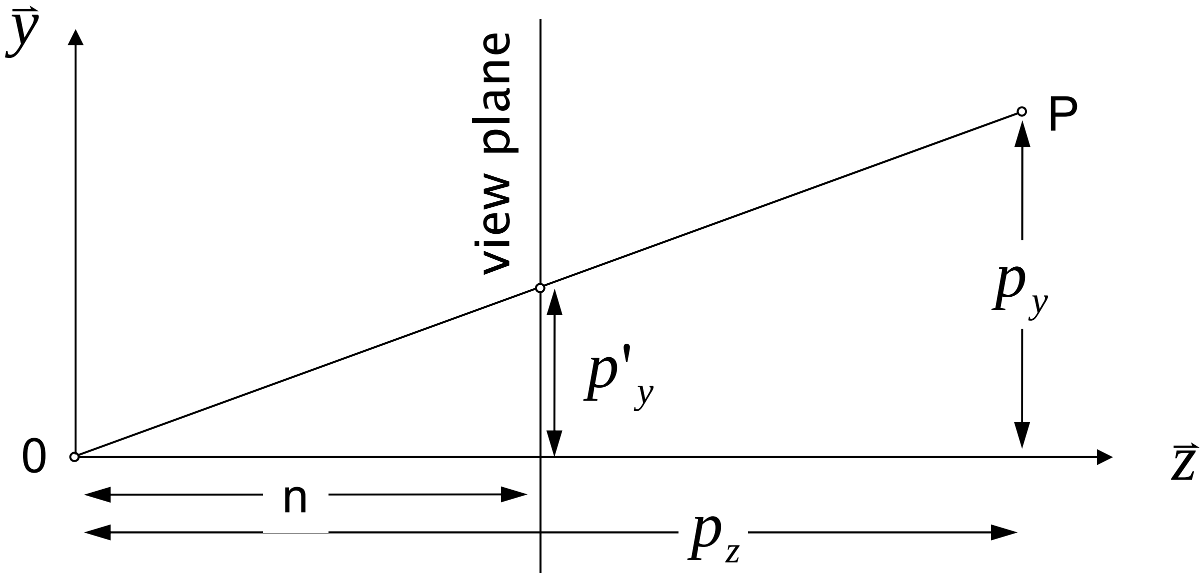
Perspective Projection



1. Negate the z coordinate (right handed → left handed coordinate system)

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{R \to L}} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Perspective Projection



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

Central Projection

- Division by a coordinate (here: z) is not possible with „normal" matrices
- Solution: Use 4D instead of 3D and use homogeneous coordinates
- Here we have:

$$\begin{pmatrix} wx \\ wy \\ wz \\ w \end{pmatrix} \equiv \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \begin{pmatrix} 6 \\ -2 \\ 8 \\ 2 \end{pmatrix} \xrightarrow{\text{hom}} \begin{pmatrix} 3 \\ -1 \\ 4 \\ 1 \end{pmatrix}$$

analog to:

$$\frac{wx}{wy} = \frac{x}{y}$$

- Important: Homogenization at the end (Division by 4th component)

Homogeneous Coordinates

- Homogeneous Point in 4D $(x, y, z, w)^T$ corresponds to 3D Euclidian Point $(x/w, y/w, z/w)^T$
- w=0: no Euclidian Point, but an idealized „point at infinity" in direction (x,y,z) of a line

- Sequence of points in homogeneous space:

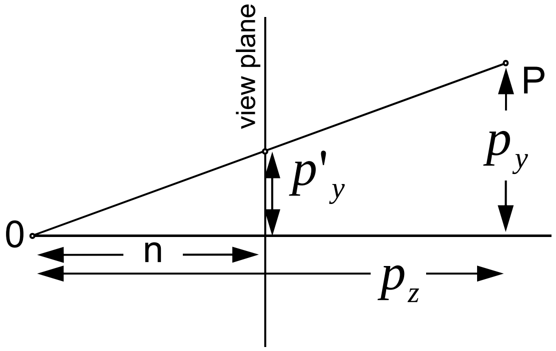
$$\begin{pmatrix} 1 \\ 2 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 0 \\ 0.01 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 0 \\ 0.0001 \end{pmatrix}, \quad \ldots$$

- Corresponds to Euclidean points:

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 100 \\ 200 \end{pmatrix}, \begin{pmatrix} 10000 \\ 20000 \end{pmatrix}, \quad \ldots$$

TU Clausthal

Central Projection: First Try



$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/n & 0 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_z/n \end{pmatrix}$$

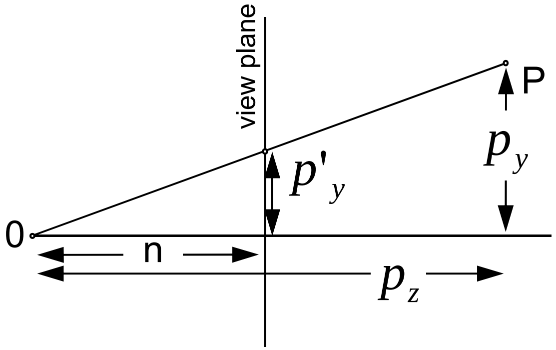
homogenize:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} np_x/p_z \\ np_y/p_z \\ n \\ 1 \end{pmatrix}$$

Goal: $p'_x = \frac{n}{p_z} \cdot p_x \qquad p'_y = \frac{n}{p_z} \cdot p_y \qquad p'_z = n$

Problem: z coordinate „vanishes“, no more depth information…
Several objects can be projected onto the same point:
Which object is in front, which in the back ?

Central Projection: Second Try



$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & ? & ? \\ 0 & 0 & 1/n & 0 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

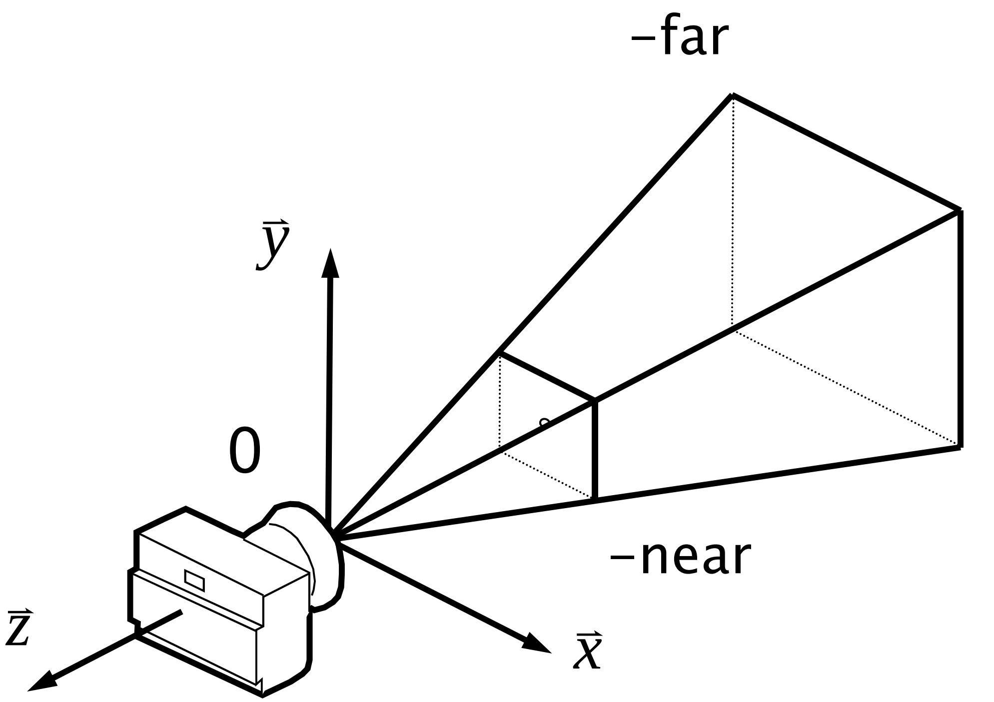
$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \cdot p_z / n \\ p_z / n \end{pmatrix}$$

homogenize:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} np_x / p_z \\ np_y / p_z \\ p_z \\ 1 \end{pmatrix}$$

Goal: $p'_x = \frac{n}{p_z} \cdot p_x \qquad p'_y = \frac{n}{p_z} \cdot p_y \qquad p'_z = p_z$

Solution

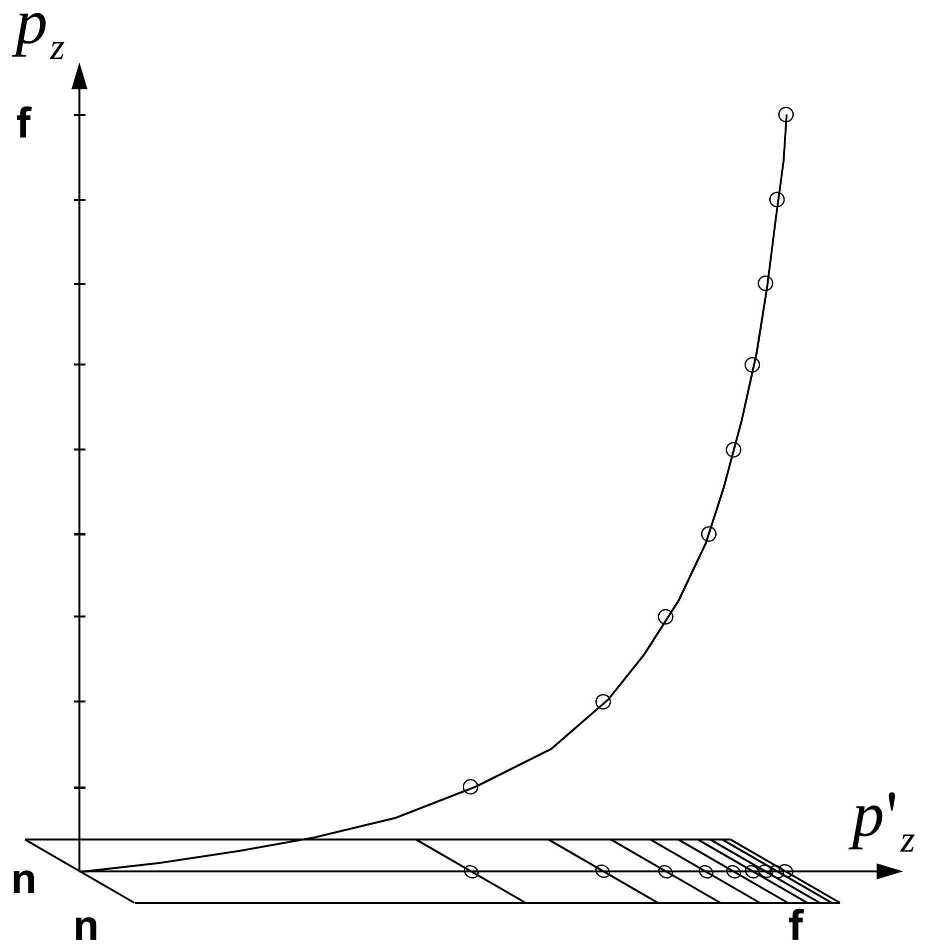


- Define a near- and far clipping plane

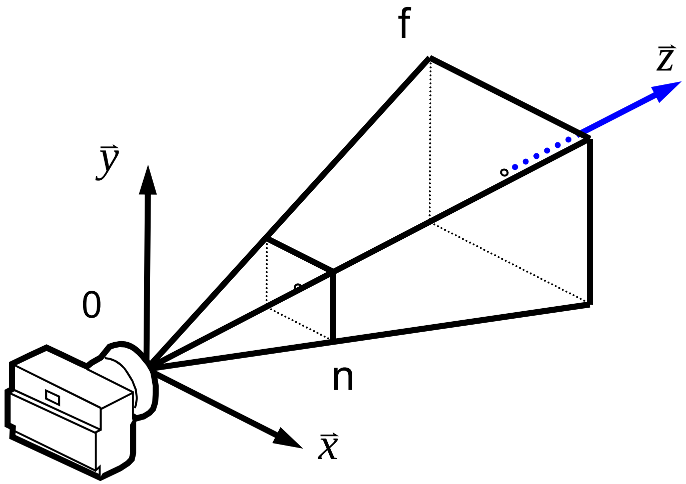
Solution

- **Non-linear scaling of the z coordinates**
- Function properties:
  - Increasing (monotonically)
  - n is mapped to n
  - f is mapped to f
  - The z values in the front get larger „distances“ then the z values in the back
    - better z buffer precision for points in the front
  - z values can be interpolated linearly
- Works with homogeneous coordinates

$$p'_z = n + f - nf / p_z$$



Solution



$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{n+f}{n} & -f \\ 0 & 0 & 1/n & 0 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \cdot \frac{n+f}{n} - f \\ p_z/n \end{pmatrix}$$

homogenize:

$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} np_x/p_z \\ np_y/p_z \\ n+f-nf/p_z \\ 1 \end{pmatrix}$$

Perspective Transformation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{n+f}{n} & -f \\ 0 & 0 & 1/n & 0 \end{pmatrix}$$

Multiplying by n
looks a bit nicer

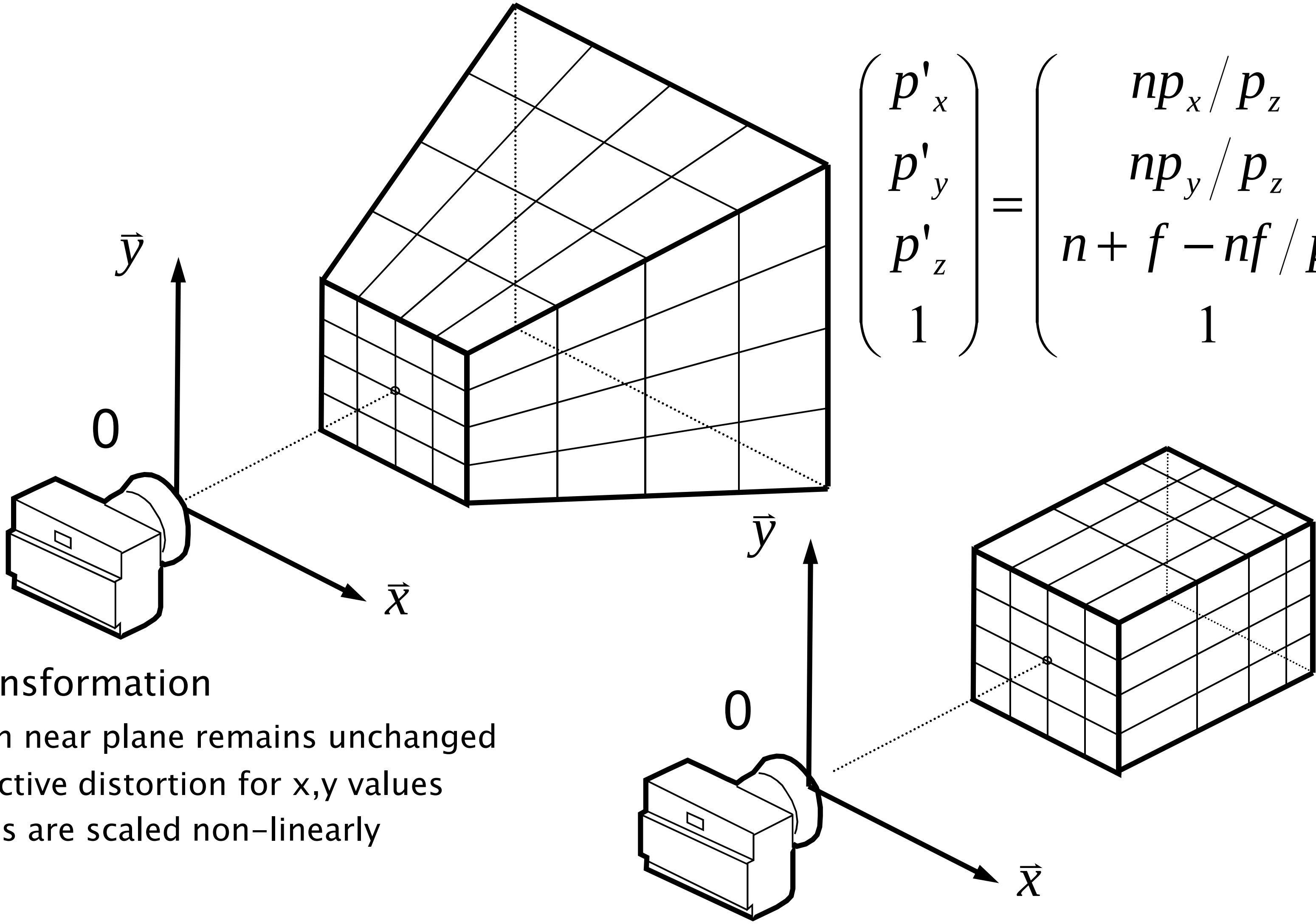
$$M_{PERSP} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

This scaling is possible due to the homogeneous coordinates:

$$\vec{p} = n \cdot \vec{p}$$

$$M \cdot (n \cdot \vec{p}) = (n \cdot M) \cdot \vec{p} = M \cdot \vec{p}$$

What is really happening ?

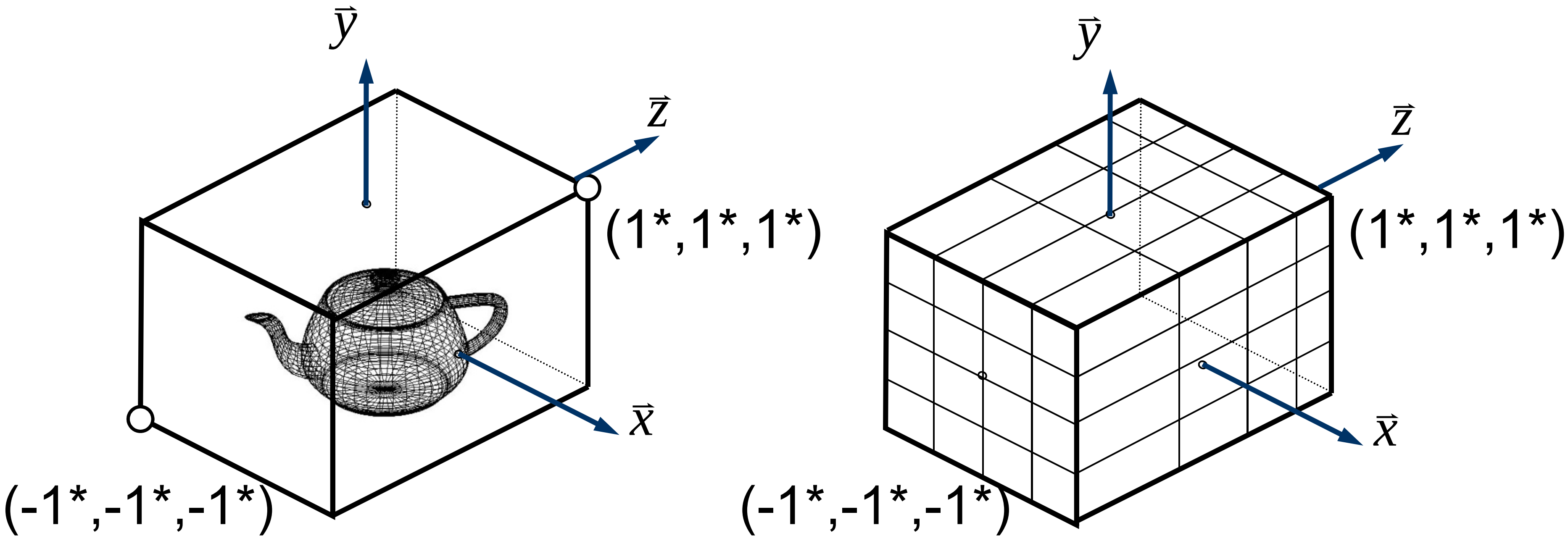


$$\begin{pmatrix} p'_x \\ p'_y \\ p'_z \\ 1 \end{pmatrix} = \begin{pmatrix} np_x / p_z \\ np_y / p_z \\ n + f - nf / p_z \\ 1 \end{pmatrix}$$

3D-Transformation

- Point in near plane remains unchanged
- perspective distortion for x,y values
- z values are scaled non-linearly

Afterwards: Orthographic Projection



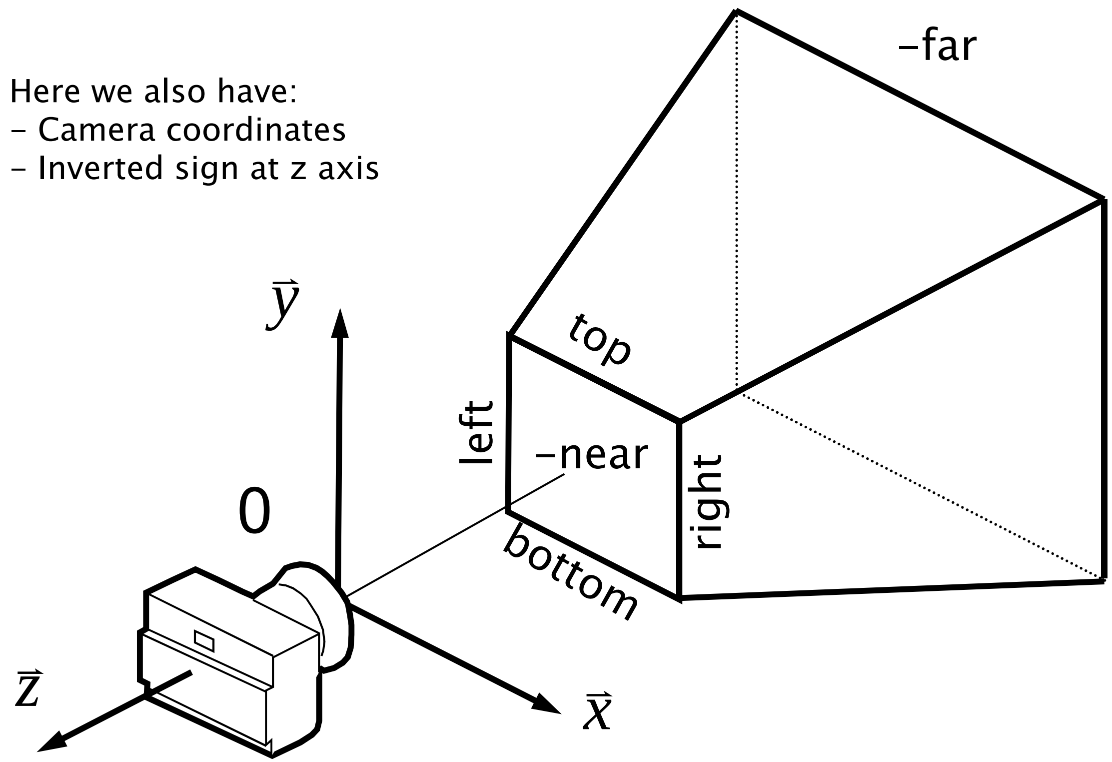
$$M_{ORTHO} \cdot M_{PERSP} \cdot M_{R \to L}$$

(* after Homogenization

Definition of a Perspective Transformation

…by defining a „view frustum“ (truncated view pyramid)

Here we also have:
- Camera coordinates
- Inverted sign at z axis



Reference point is the intersection point of the z axis with the near plane

left/right: coordinates of the borders along the x axis in the near plane

bottom/top: coordinates of the borders along the y axis in the near plane

```
glm::frustum(left, right, bottom, top, near, far);
```

e.g. `frustum(-1, 1, -1, 1, 3, 10);`

Perspective Projection

The projection matrix results from: $\underbrace{M_{ORTHO} \cdot M_{PERSP} \cdot M_{R\to L}}_{M_{PROJECTION}}$

$$\underbrace{\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{ORTHO}} \cdot \underbrace{\begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{pmatrix}}_{M_{PERSP}} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{M_{R\to L}}$$

```
glm::frustum(left, right, bottom, top, near, far);
```

Perspective Projection

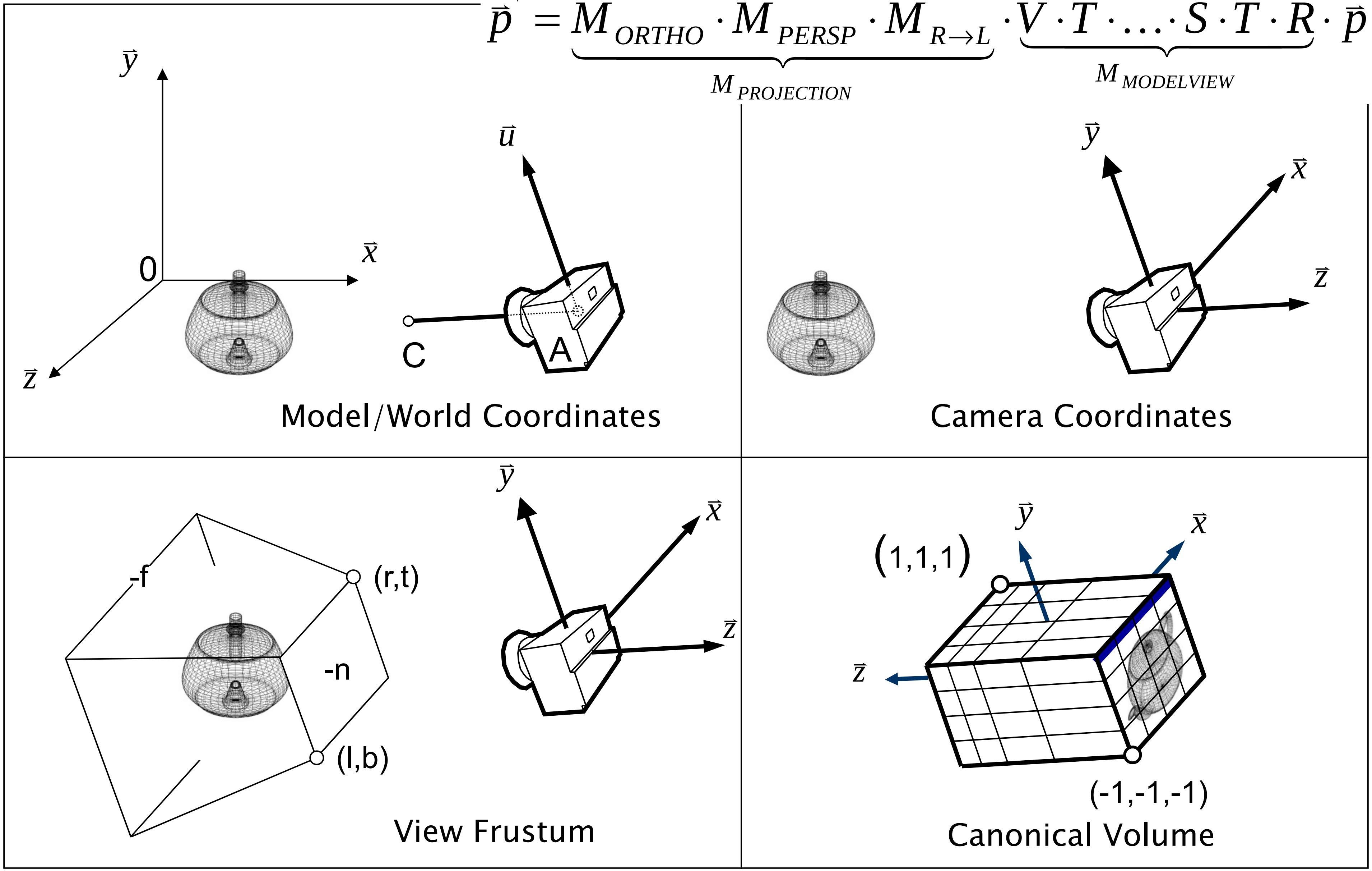
Multiplying the four matrices results in:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

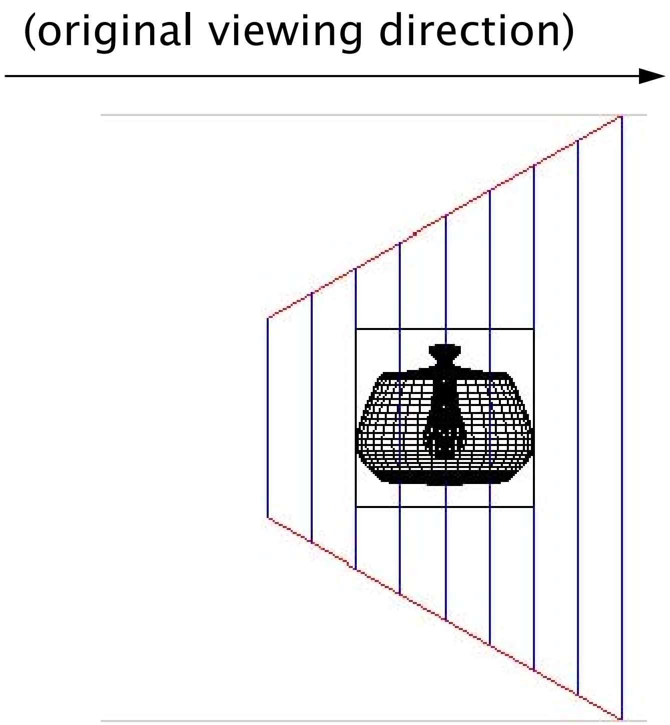
```
glm::frustum(left, right, bottom, top, near, far);
```

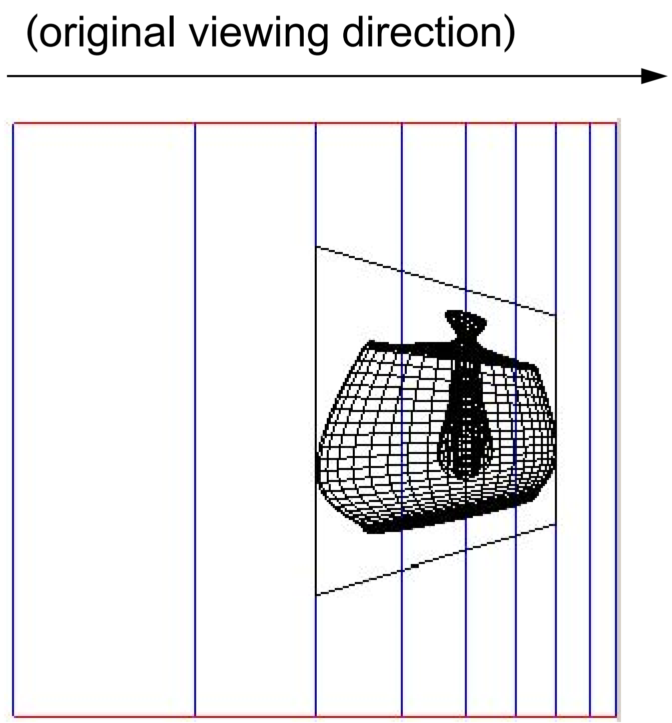
Transformation: Point in camera coordinates → Point in canonical volume

$$\vec{p}' = \underbrace{M_{ORTHO} \cdot M_{PERSP} \cdot M_{R \to L}}_{M_{PROJECTION}} \cdot \underbrace{V \cdot T \cdot \ldots \cdot S \cdot T \cdot R}_{M_{MODELVIEW}} \cdot \vec{p}$$



Rotating the Projection Matrix





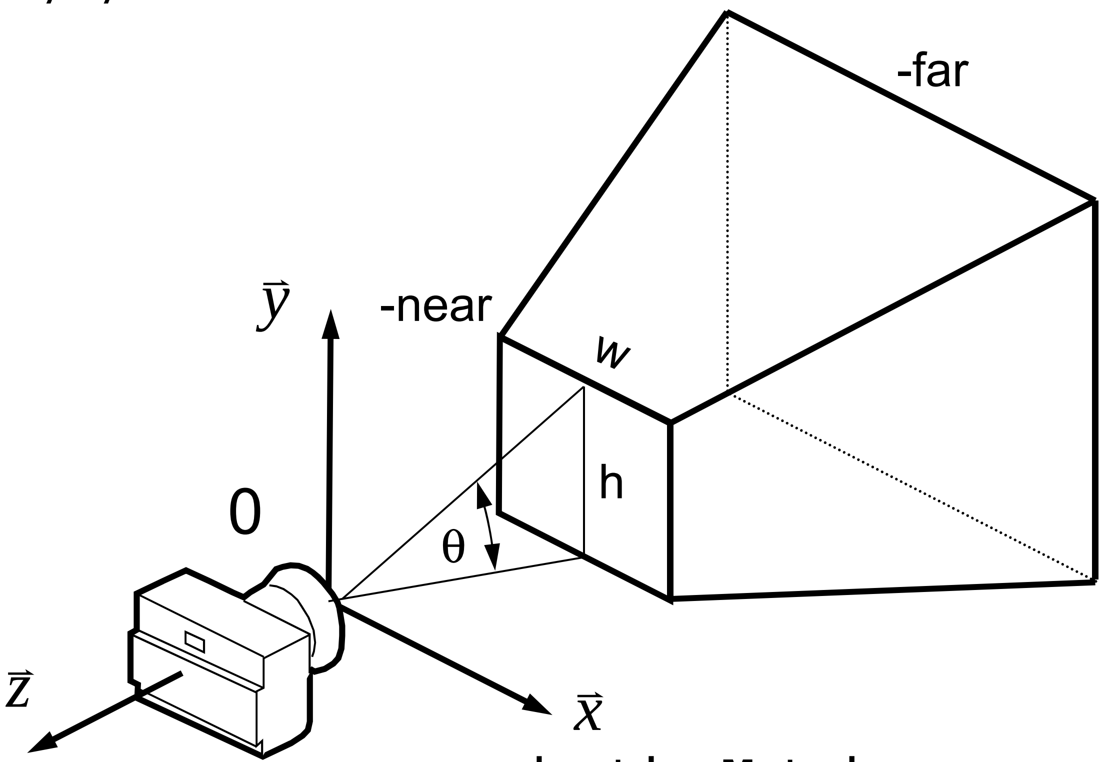
Orthographic transformation with rotation:
A view from the side of the view frustum with equidistant z values

Perspective transformation with rotation:
A view from the side of the canonical volumen (non-linear scaling of the z values)

Perspective Transformation → Parallel Projection leads to a perspective image

Alternative: glm::perspective

Only symmetrical frustum



$$aspect = \frac{w}{h}$$

```
projectionMatrix =
glm::perspective (theta, aspect, near, far)
```

Here the perspective projection is defined by the full, vertical opening angle (in degrees). Additionally, we define the aspect (width/height).

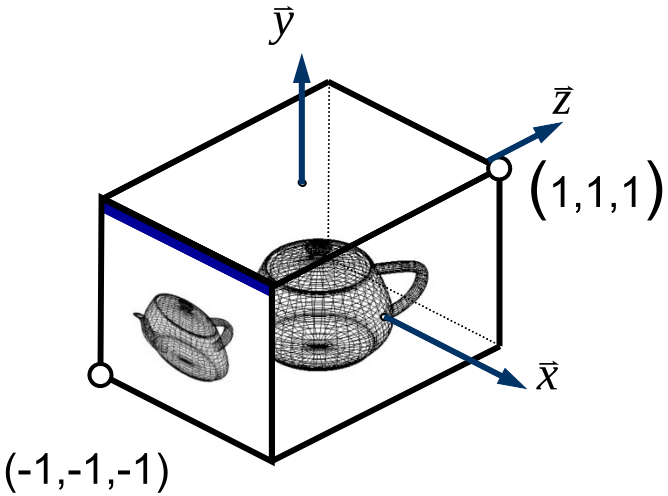
Example

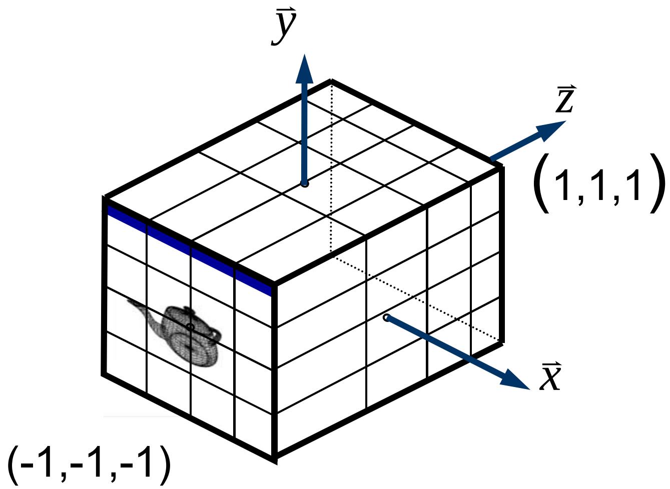
```
projectionMatrix = frustum(-1, 1, -1, 1, 0.1, 10000);

viewMatrix = lookAt(10, 10, 10, 0, 0, 0, 0, 1, 0);

modelMatrix = translate(modelMatrix, vec3(0, 0, 10));
modelMatrix = rotate(modelMatrix, alpha, vec3(0, 1, 0));


mvpMatrix = projectionMatrix * viewMatrix * modelMatrix;
```

$$\vec{p}^{\,'} = \underbrace{M_{ORTHO} \cdot M_{PERSP} \cdot M_{R \rightarrow L}}_{M_{PROJECTION}} \cdot \underbrace{V \cdot T \cdot \ldots \cdot S \cdot T \cdot R}_{M_{MODELVIEW}} \cdot \vec{p}$$

Viewport

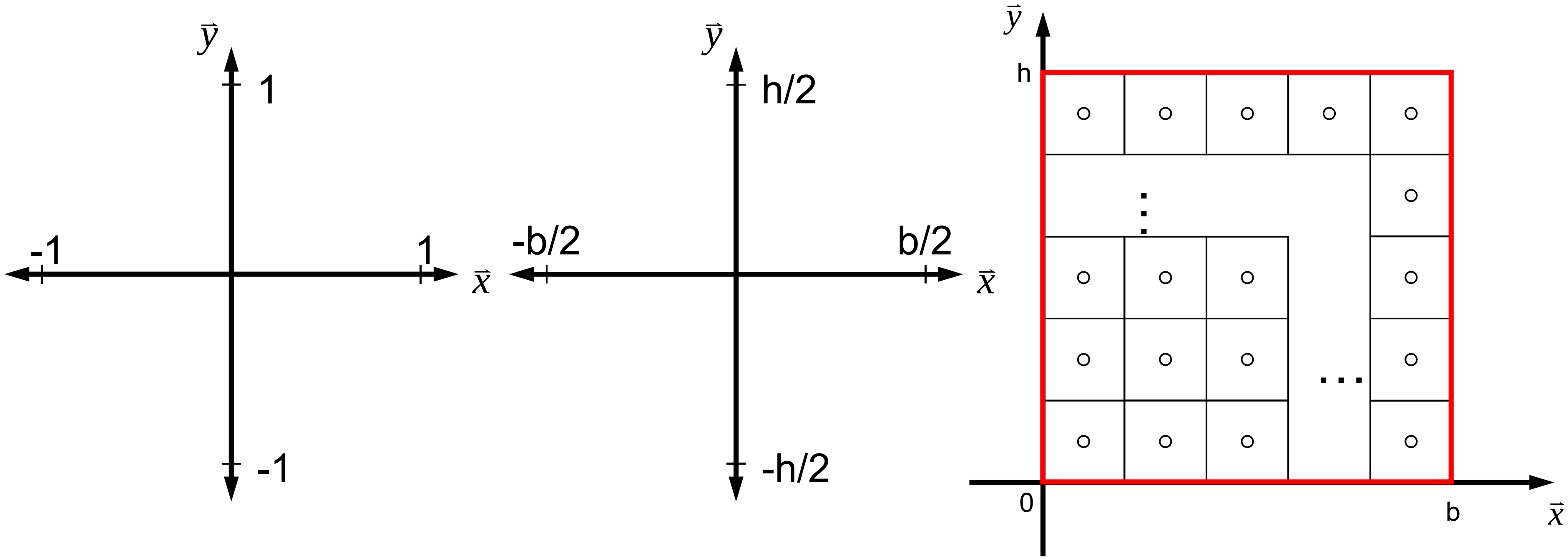
Viewport Transformation





- Transform the canonical volume
  - Transform the xy coordinates from (–1,1) to window coordinates, e.g. (0,599)x(0,599)
  - Transform the z coordinates from (–1,1) to (0,1) for the z buffer

Viewport in OpenGL



$$p'_x = b/2 \cdot p_x + b/2$$
$$p'_y = h/2 \cdot p_y + h/2$$

	$p_x = -1$	$p_x = 1$
p'_x	0	b

Viewport in General

- Window coordinates
- OpenGL command

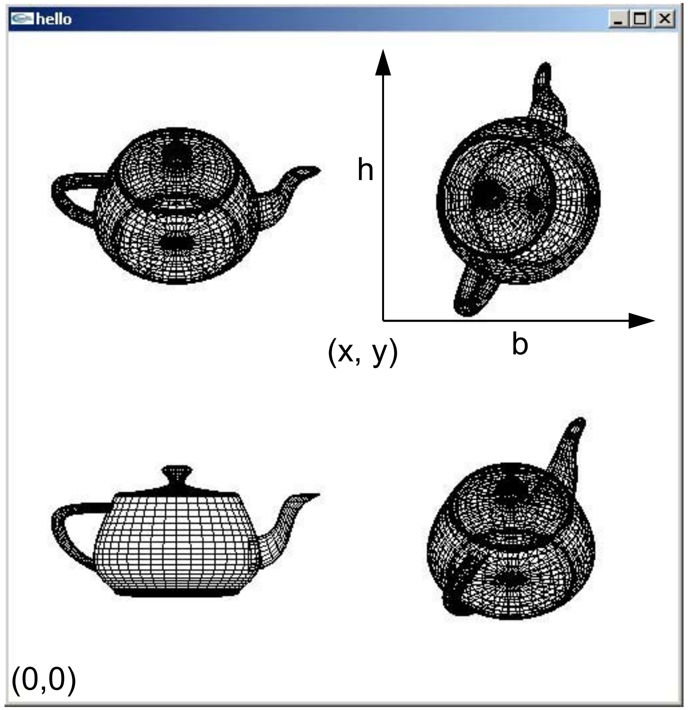
```
glViewport( GLint x, y, b, h);
```

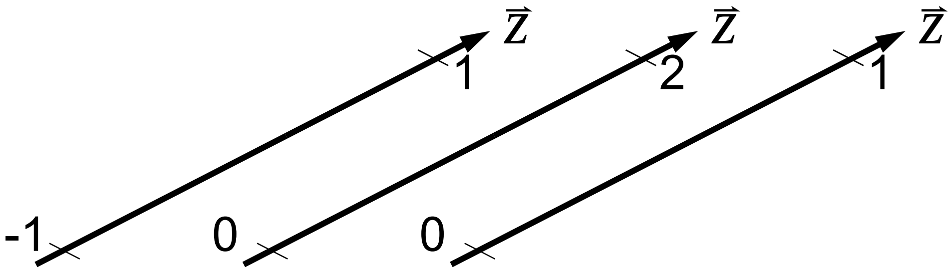
$$\textit{Viewport}:$$

$$p'_x = b/2 \cdot p_x + b/2 + x$$

$$p'_y = h/2 \cdot p_y + h/2 + y$$

$$\boxed{p'_z = \frac{1}{2} \cdot (p_z + 1)}$$





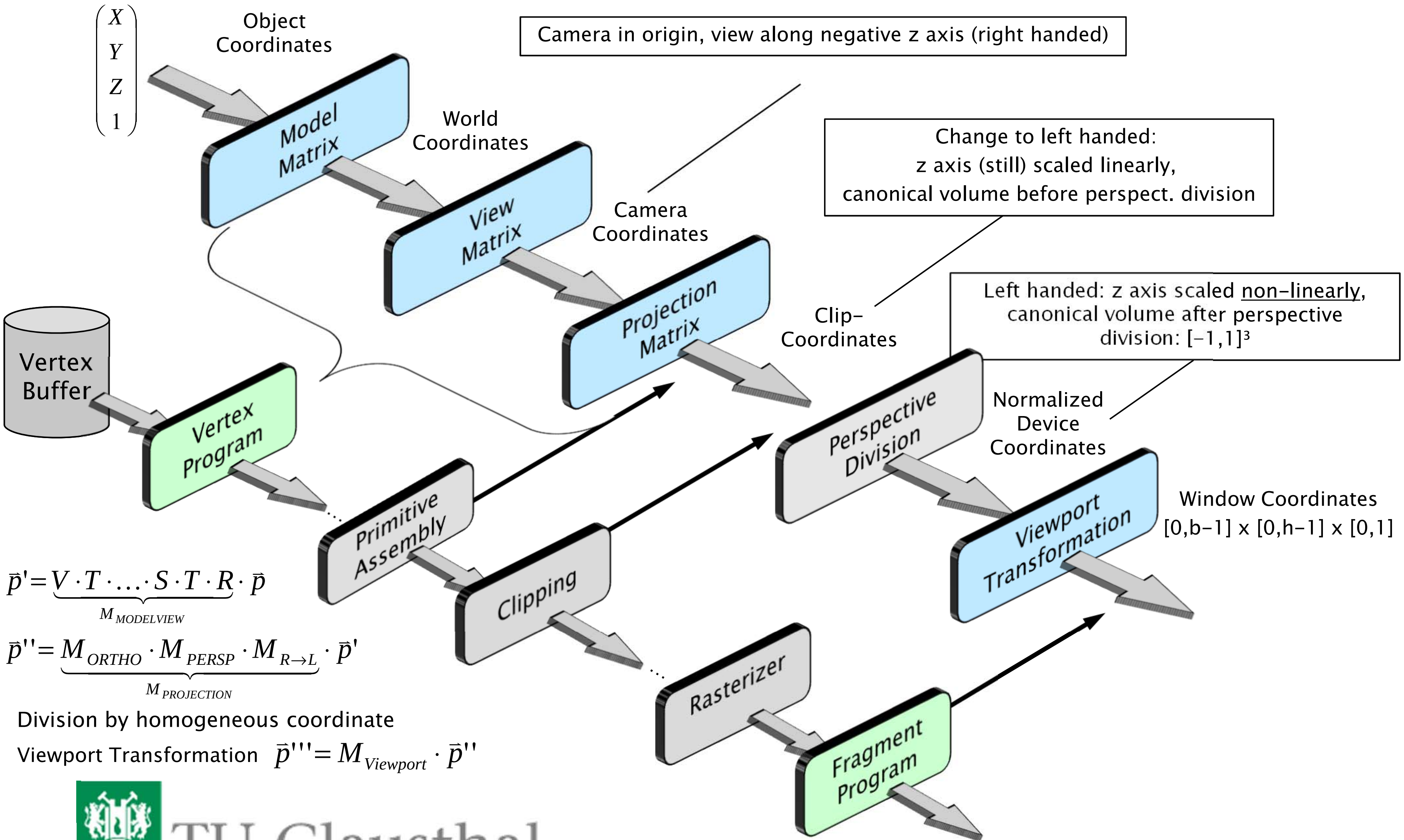
Viewport as Matrix

$$M_{Viewport}$$

$$\begin{pmatrix} \frac{b}{2} & 0 & 0 & \frac{b}{2}+x \\ 0 & \frac{h}{2} & 0 & \frac{h}{2}+y \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
glViewport( GLint x, y, b, h);
```

Rendering Pipeline (The path from vertex to pixel)



$$\vec{p}' = \underbrace{V \cdot T \cdot \ldots \cdot S \cdot T \cdot R}_{M_{MODELVIEW}} \cdot \vec{p}$$

$$\vec{p}'' = \underbrace{M_{ORTHO} \cdot M_{PERSP} \cdot M_{R \to L}}_{M_{PROJECTION}} \cdot \vec{p}'$$

Division by homogeneous coordinate

Viewport Transformation $\vec{p}''' = M_{Viewport} \cdot \vec{p}''$

That´s all for today

- Next week we will have a detailed look at buffer objects
- First exercise at 9:00 in IfI Room 116