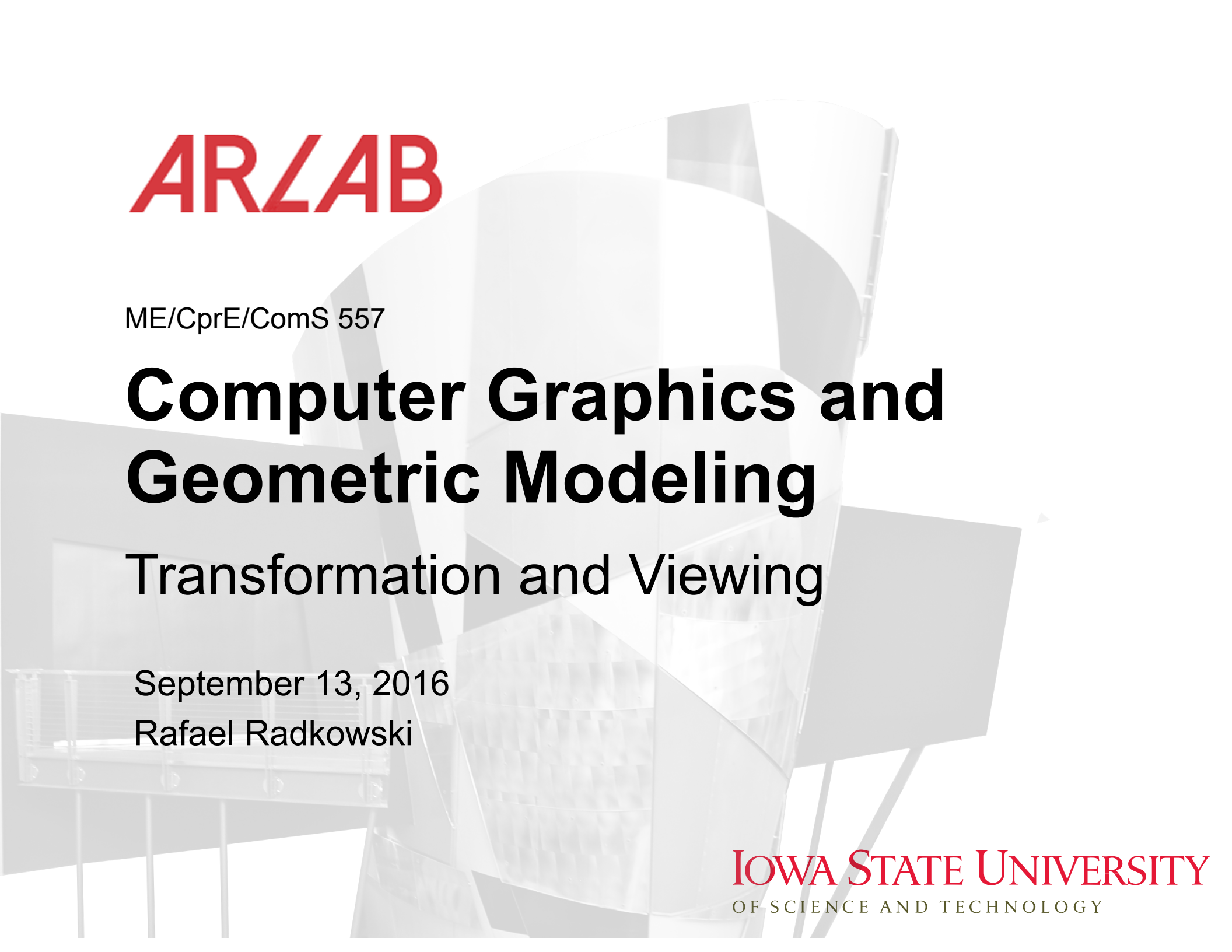


ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Transformation and Viewing

September 13, 2016

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Topics

ARLAB

- Homogenous Space
- Transformation of 3D models
- Viewing in 3D
- Transformations in OpenGL

Goals of this class:

- Understanding the transformation processes in Computer Graphics
- Transform 3D models in 3D space
- Create and move a virtual camera in 3D space

ARLAB

Homogenous Space

Affine Transformations

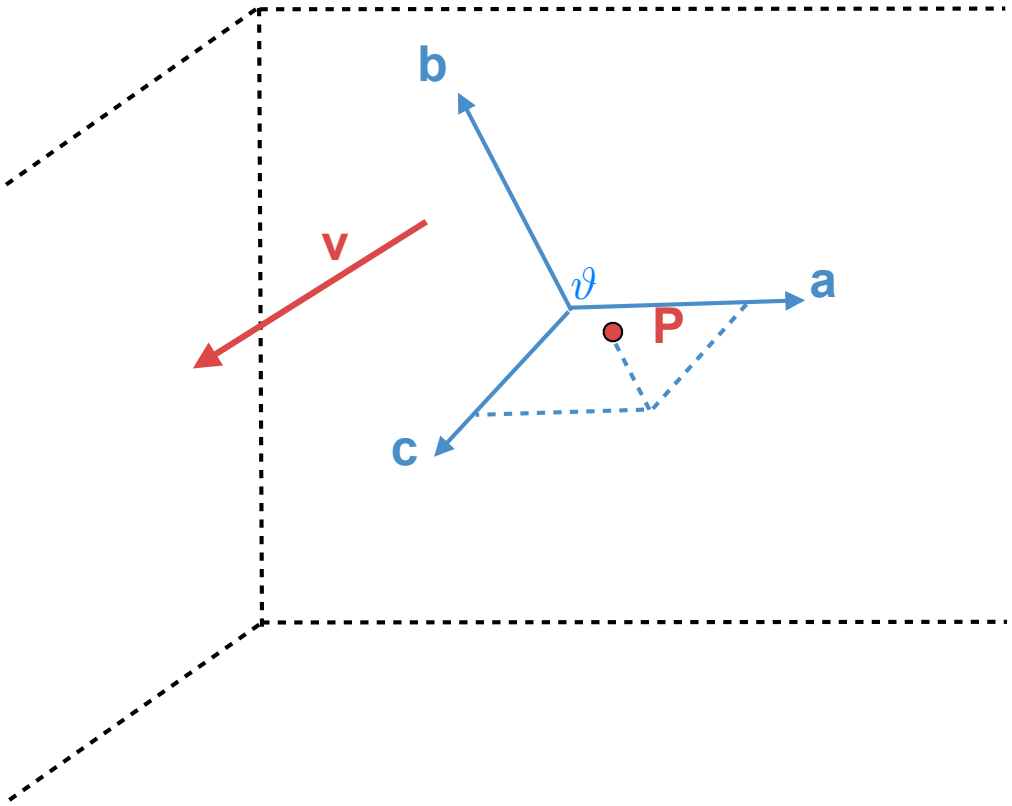
ARLAB

- Affine transformations are all mathematical transformations that preserve the geometrical structure of objects (point remains a point, a line a line, etc.)
- Cornerstone of computer graphics
- Lead to disasters and bugs because points and vectors are not the same!
- A point P has only a location, but no size and direction.
- A vector **v** has a direction and a size, but it has no location.

$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \mathbf{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

- Problems occur when points and vectors are transformed between coordinate systems

Coordinate Systems and Coordinate Frames



ϑ : origin of the coordinate frame
a,b,c: coordinate frame vector
v: vector
P: point in coordinate frame

A coordinate frame consists of an origin ϑ and three mutually perpendicular vectors **a**, **b**, **c.**

A vector in this coordinate frame is defined as:

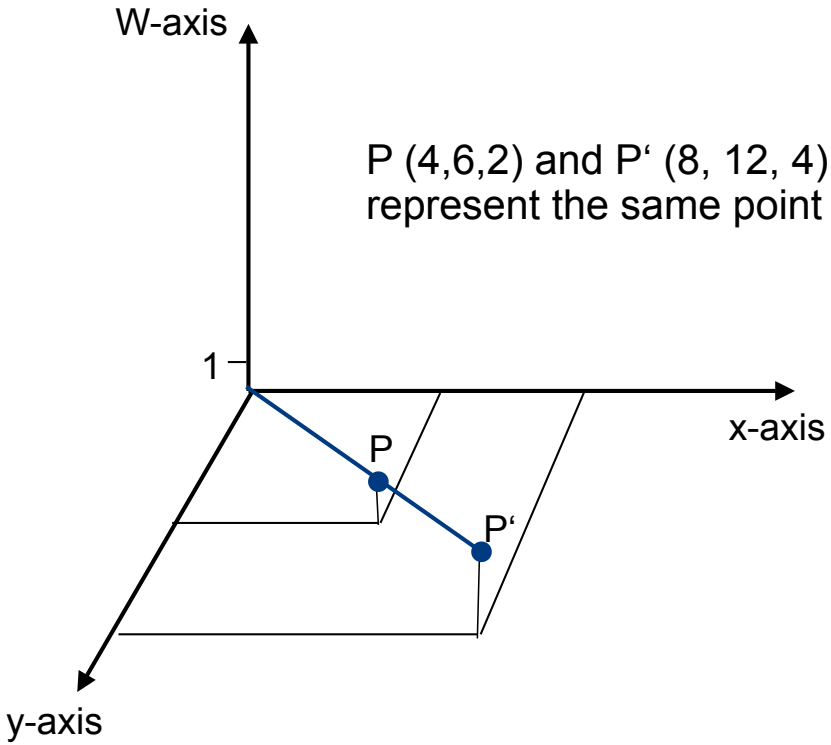
$$\mathbf{v} = v_1\mathbf{a} + v_2\mathbf{b} + v_3\mathbf{c}$$

A point in this coordinate systems is defined as:

$$P = \vartheta + p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c}$$

The basic idea is to make the origin of each coordinate system explicit.

Homogenous Coordinate System - in 2D Space



The homogeneous coordinate system or projective transformation or projective space is a geometric coordinate system to represent projective information. The advantage is that the coordinates of points and vectors can be represented using a unique matrix. A scaling parameter w

$$H(x, y) = \left(\frac{x}{w}, \frac{y}{w}, 1\right)$$

which also allows to distinguish points from vectors

- if w = 0: the matrix is a vector
- if w = 1: the matrix is a point

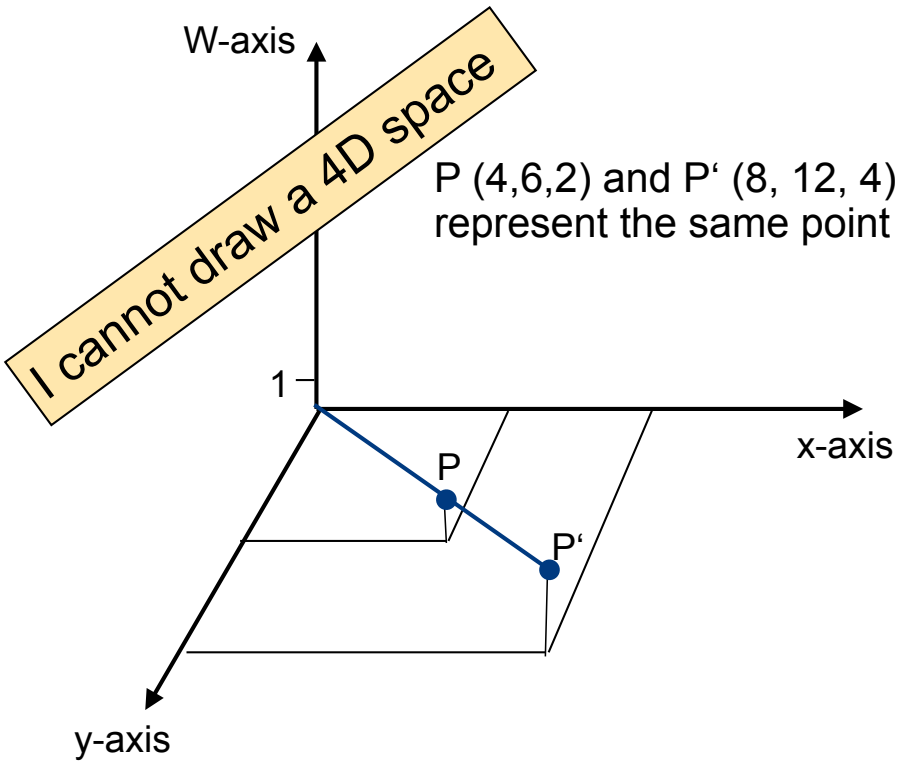
Same matrix for every transformation, here, in 2D space

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & w \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Transformation of homogenous coordinates in cartesian coordinates

$$\begin{bmatrix} x_k \\ x_k \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x_h}{W} \\ \frac{y_h}{W} \\ 1 \end{bmatrix}$$

Homogenous Coordinate System - in 3D Space



Same matrix for every transformation, here, in 3D space

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & w \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The homogeneous coordinate system or projective transformation or projective space is a geometric coordinate system to represent projective information. The advantage is that the coordinates of points and vectors can be represented using a unique matrix. A scaling parameter w

$$H(x, y, z) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$$

which also allows to distinguish points from vectors

- if w = 0: the matrix is a vector
- if w = 1: the matrix is a point

Transformation of homogenous coordinates in cartesian coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \\ 1 \end{bmatrix}$$

Homogenous Coordinate System - Projective Transformation

The homogenous representation of a vector and a point use the same set of underlying attributes a, b, c, and ϑ .

For a vector:

$$\mathbf{v} = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \vartheta) \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} a_1 & b_1 & c_1 & \vartheta_1 \\ a_2 & b_2 & c_2 & \vartheta_2 \\ a_3 & b_3 & c_3 & \vartheta_3 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

For a point:

$$P = (\mathbf{a}, \mathbf{b}, \mathbf{c}, \vartheta) \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{pmatrix}$$

$$P = \begin{pmatrix} a_1 & b_1 & c_1 & \vartheta_1 \\ a_2 & b_2 & c_2 & \vartheta_2 \\ a_3 & b_3 & c_3 & \vartheta_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ 1 \end{pmatrix}$$

- A 0 and 1 are used to distinguish a vector and a point
- Same notation for different representation of points and vectors
- Simplifies the maths.

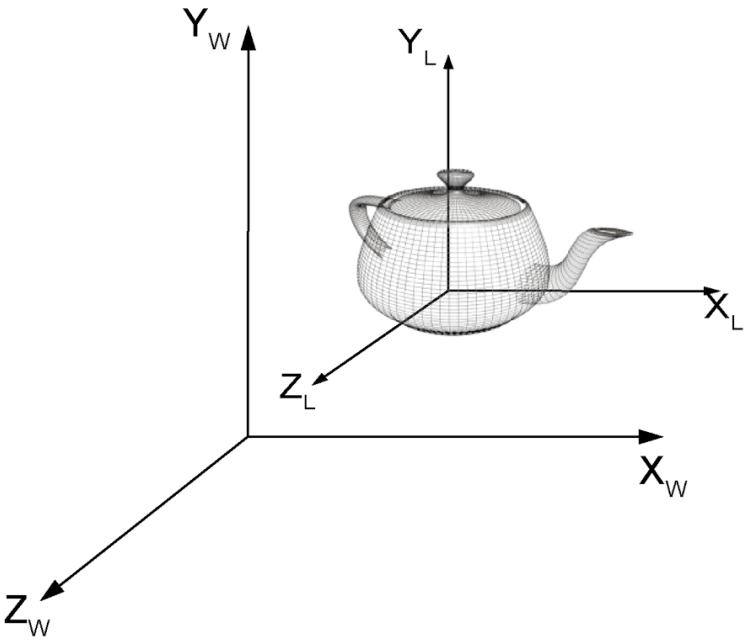
ARLAB

Transformation in 3D

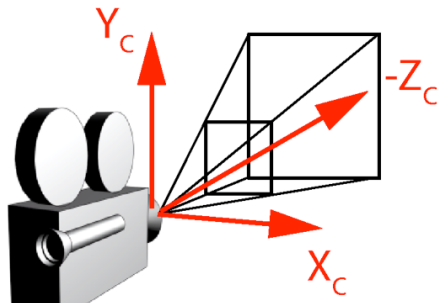
Coordinate Systems

ARLAB

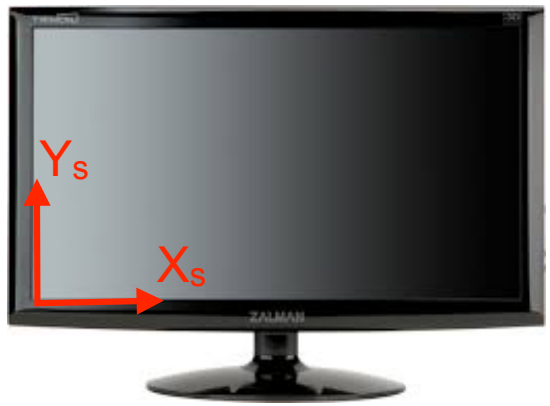
World Reference Frame and Coordinate Frame



Camera Coordinate System

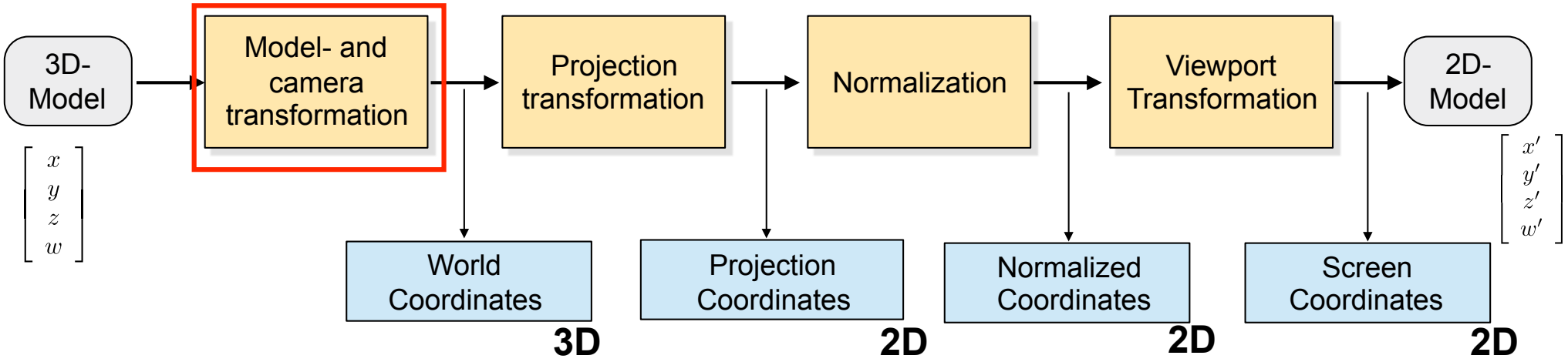


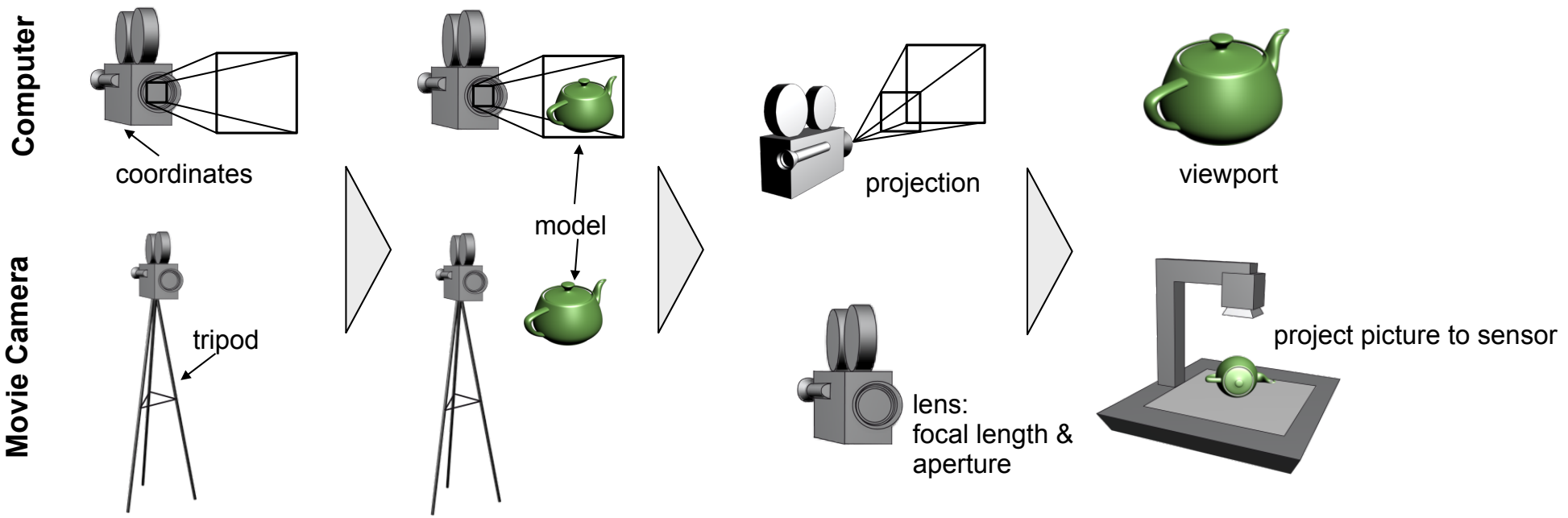
Screen Coordinate System



- The world reference coordinate system is on imaginary coordinate system of the entire virtual world.
- Object coordinate / coordinate frames are specified for each single object
- The camera coordinate system is a **3D coordinate system** with center in the camera. We typically look into the negative z axis.
- The screen coordinate system is a **2D coordinate system** in normalized coordinates or screen coordinates.

3D Transformation Pipeline





The Homogenous Matrix

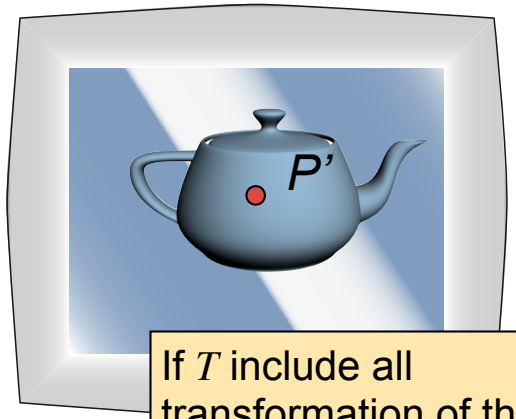
The homogenous matrix is used in computer graphics to transform objects in 3D space.

It is a 4 x 4 matrix, which elements represent translations, rotations, and the scale factor of an object.

Scale Rotation Translation

$$T := \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{11} & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

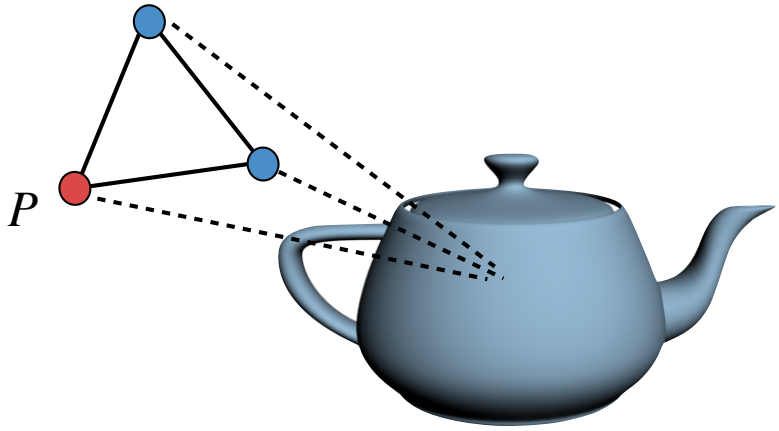
with $r_{44} = w$



If T include all transformation of the fixed function rendering pipeline

For all transformations:

$$P' = T \cdot P$$



The Homogenous Matrix

The homogenous matrix is used in computer graphics to transform objects in 3D space.

It is a 4 x 4 matrix, which elements represent translations, rotations, and the scale factor of an object.

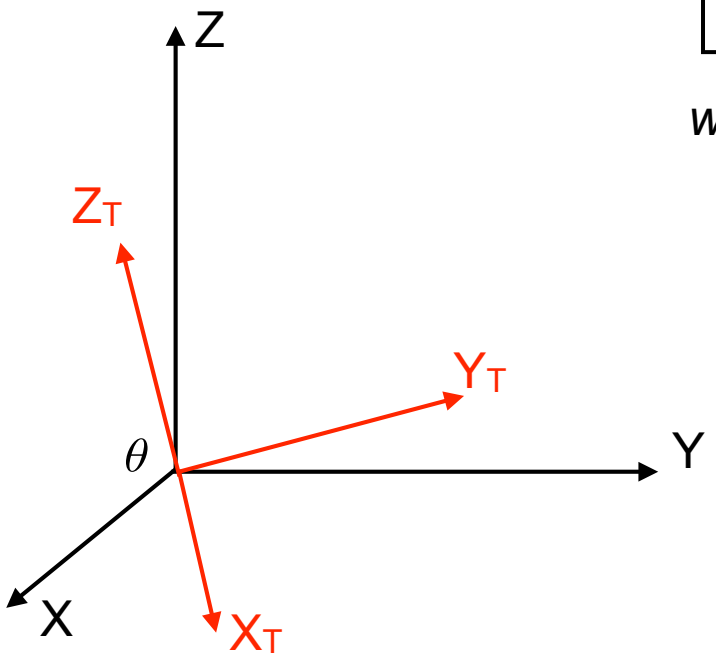
Coordinate axes

X_T Y_T Z_T

$$T := \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ r_{21} & r_{22} & r_{23} & r_{24} \\ r_{31} & r_{32} & r_{33} & r_{34} \\ r_{11} & r_{42} & r_{43} & r_{44} \end{bmatrix}$$

Translation

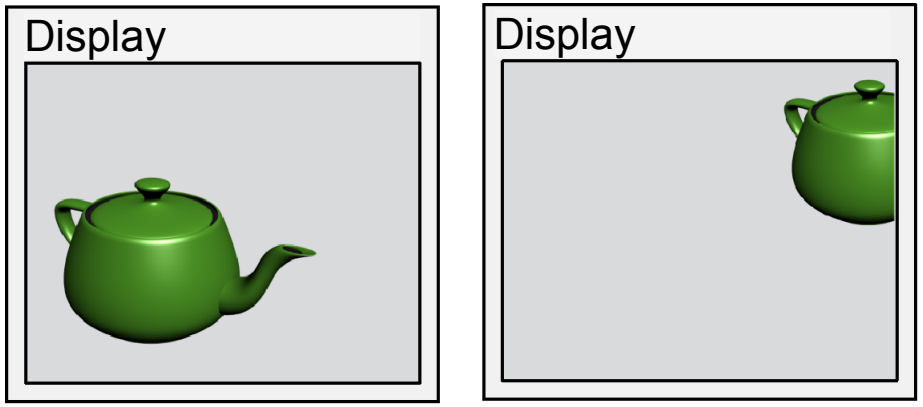
with $r_{44} = w$



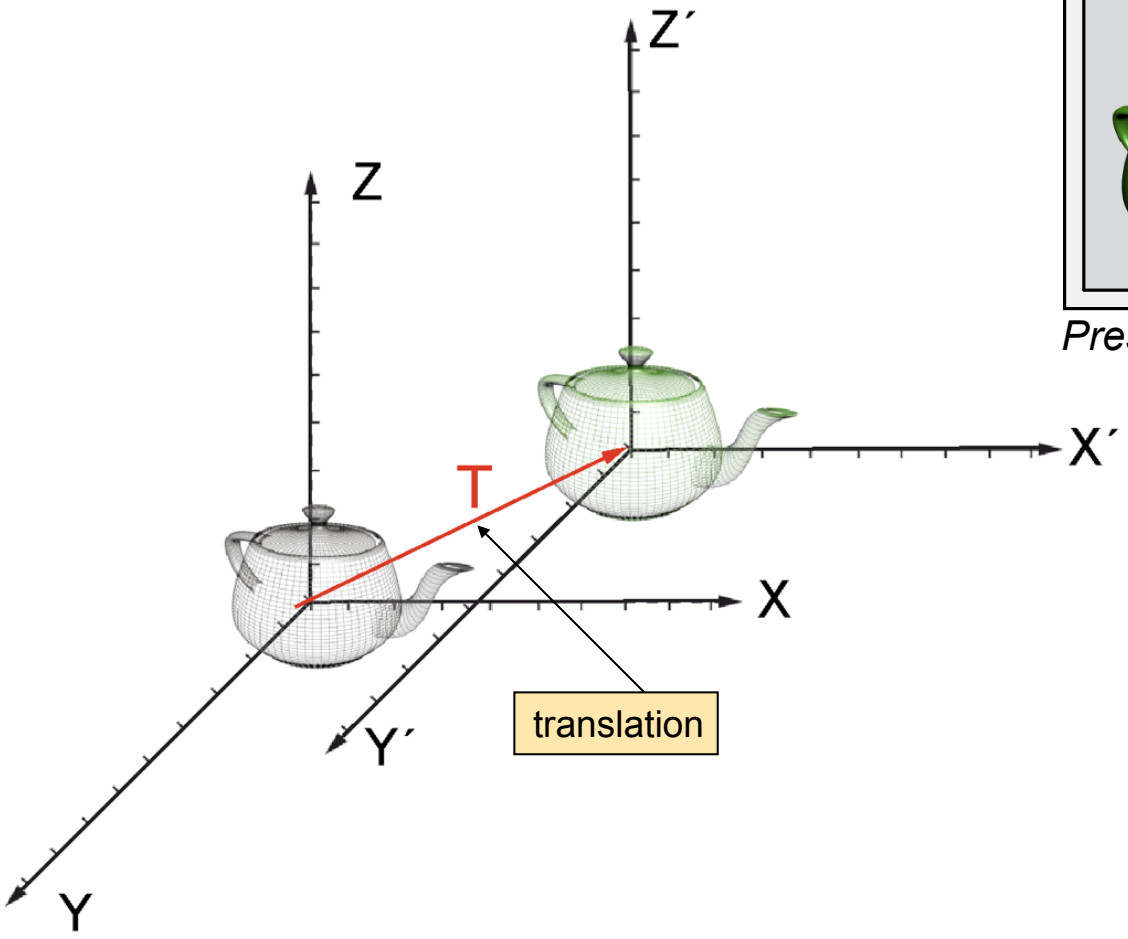
The columns of the matrix represent a coordinate system within a given reference frame θ

3D-Transformation
Translation

ARLAB



Presentation of a translated model on screen



Translation of a 3D object in 3D space

Translation of a point P(x, y, z) by T

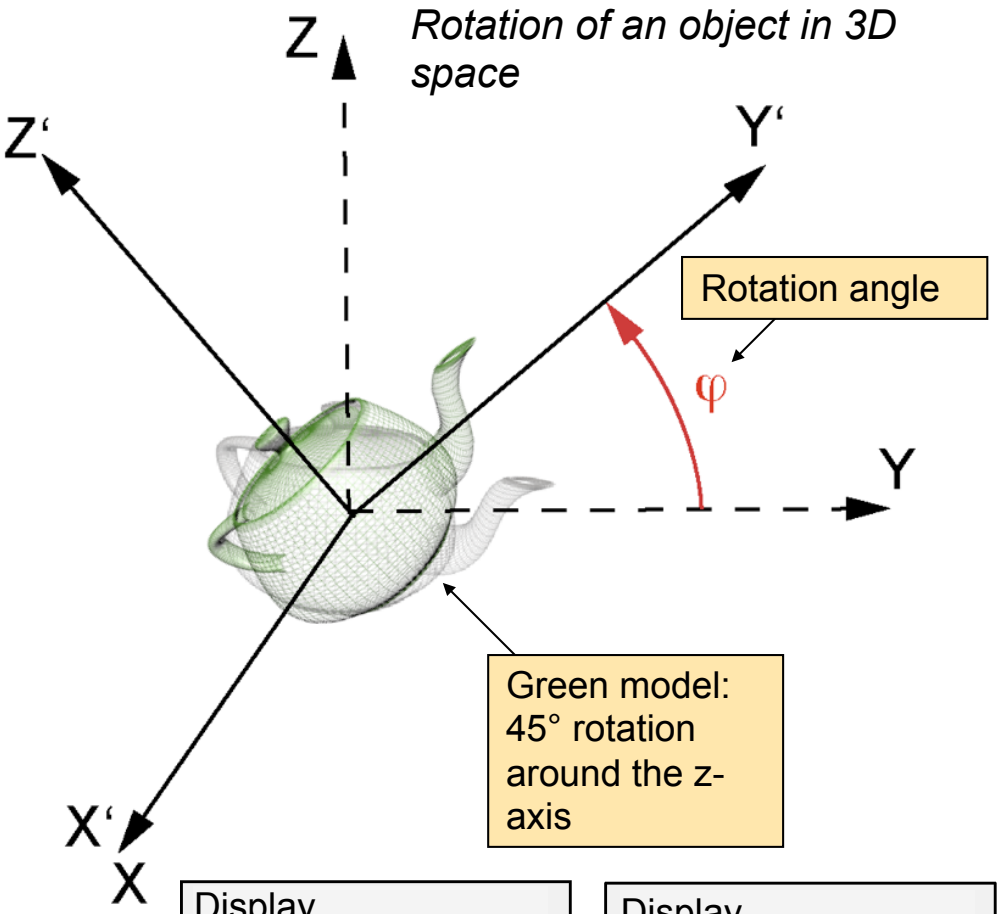
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

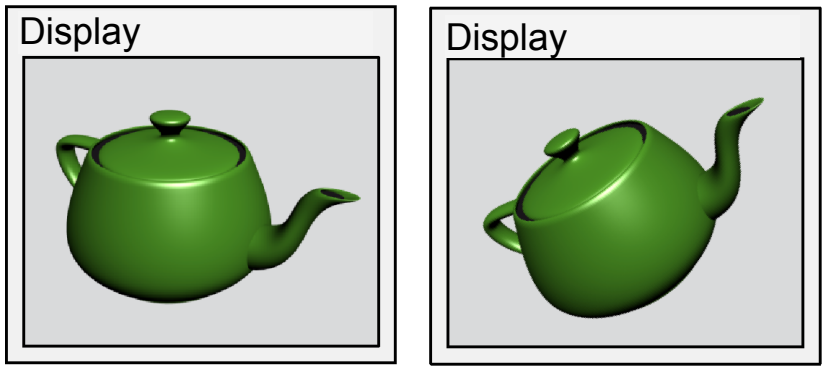
$$P' = T(t_x, t_y, t_z) \cdot P \quad \text{with}$$

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

VRAC|HCI IOWA STATE UNIVERSITY OF SCIENCE AND TECHNOLOGY

3D-Transformation
Rotation





Presentation of a rotated model on screen

Rotation around the x-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\varphi) & -sin(\varphi) & 0 \\ 0 & sin(\varphi) & cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_x(\varphi) \cdot P$$

Rotation around the y-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\varphi) & 0 & sin(\varphi) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\varphi) & 0 & cos(\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_y(\varphi) \cdot P$$

Rotation around the z-axis

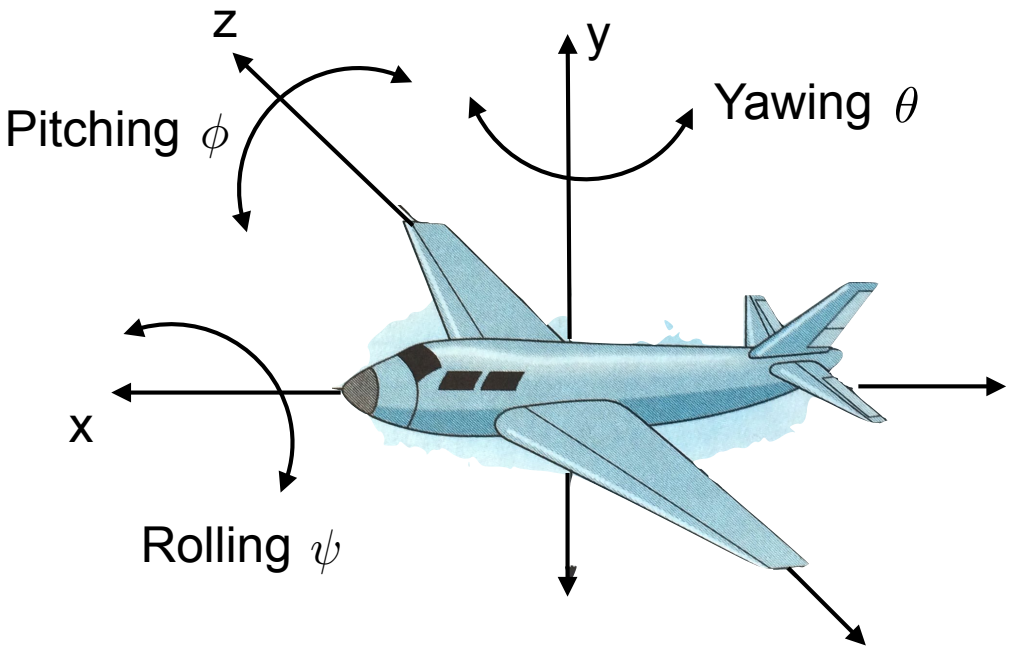
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\varphi) & -sin(\varphi) & 0 & 0 \\ sin(\varphi) & cos(\varphi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_z(\varphi) \cdot P$$

3D-Transformation
Euler Angles

ARLAB

Euler angles are a mechanism for creating a rotation through the sequence of three simple rotations:



$$\mathbf{M} = R_{yz}(\psi)\ R_{zx}(\theta)\ R_{xy}(\phi)$$

The sequence is important

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3D-Transformation
Euler Angles

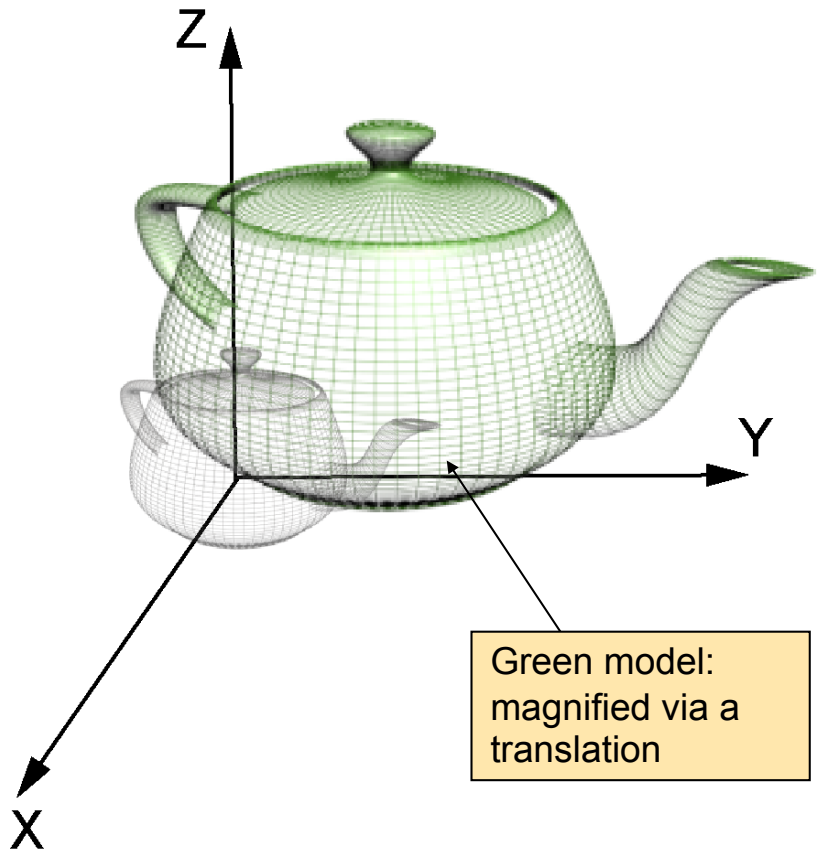
ARLAB

In homogenous coordinate system

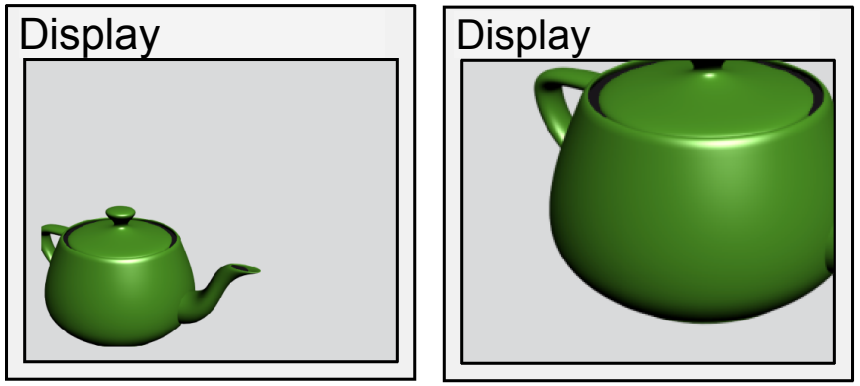
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi & 0 \\ 0 & \sin\psi & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D-Transformation
Scaling

ARLAB



Scale of an object in 3D space



Presentation of a scaled model on screen

Scaling of a point P(x, y, z) by the scaling factor S

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = S(s_x, s_y, s_z) \cdot P \quad \text{with}$$

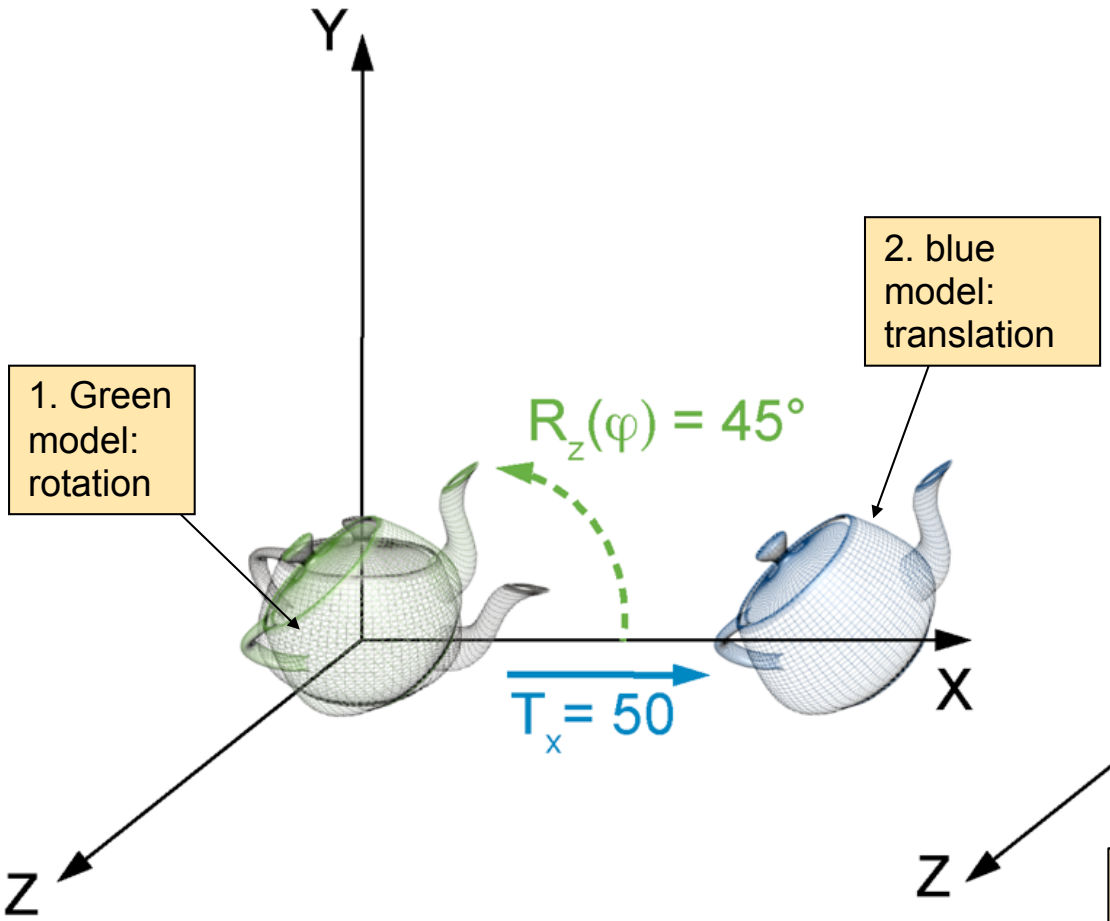
$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Every scaling incorporates a translation, if the 3D model is not at coordinate system origin.

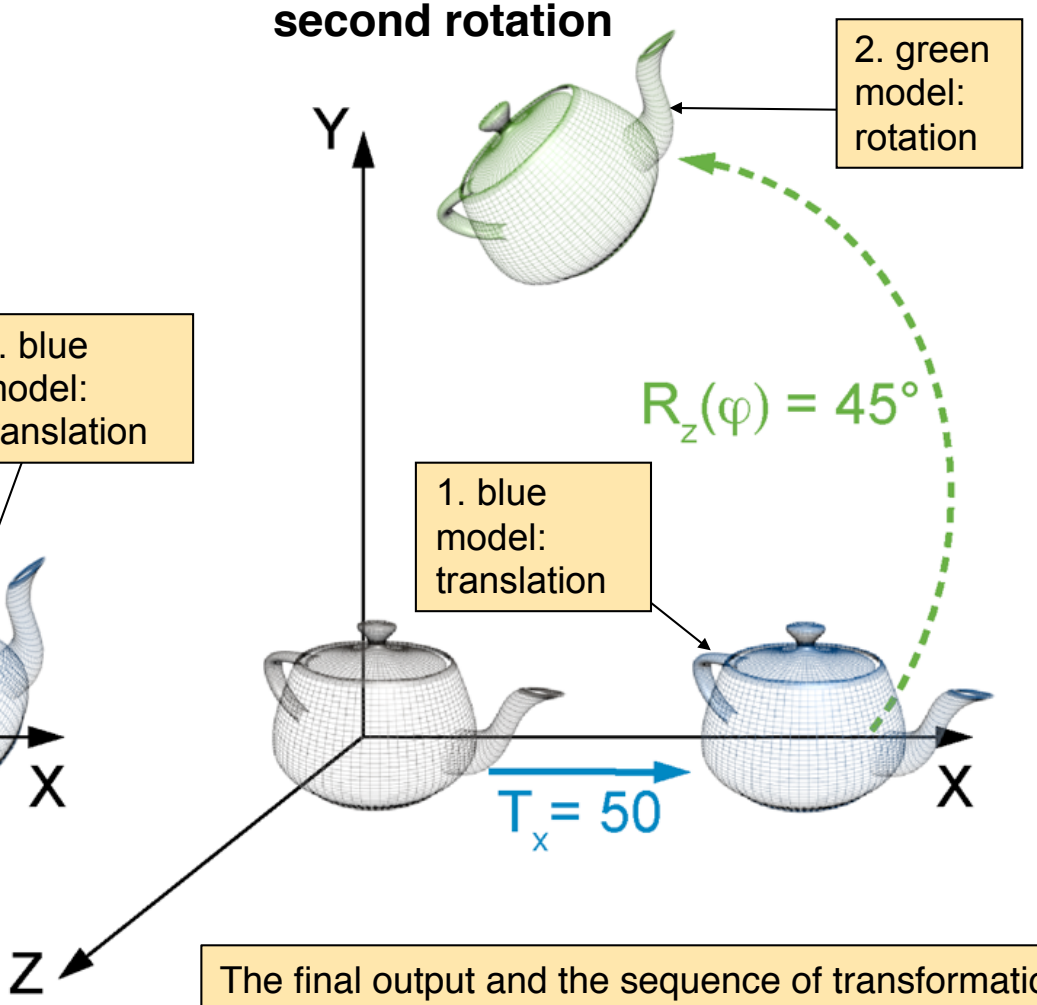
Sequence of Transformation
World Coordinates

ARLAB

First rotation, second translation



First translation, second rotation



The final output and the sequence of transformation depends on the sequence of matrix multiplications:

$$P' = T_x \cdot R_z \cdot P \quad \text{vs.} \quad P' = R_z \cdot T_x \cdot P$$

But

Matrices are associative in general, which allows us to multiple them first and then multiply a vector of points next.

$$\mathbf{M} = \mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{v})$$

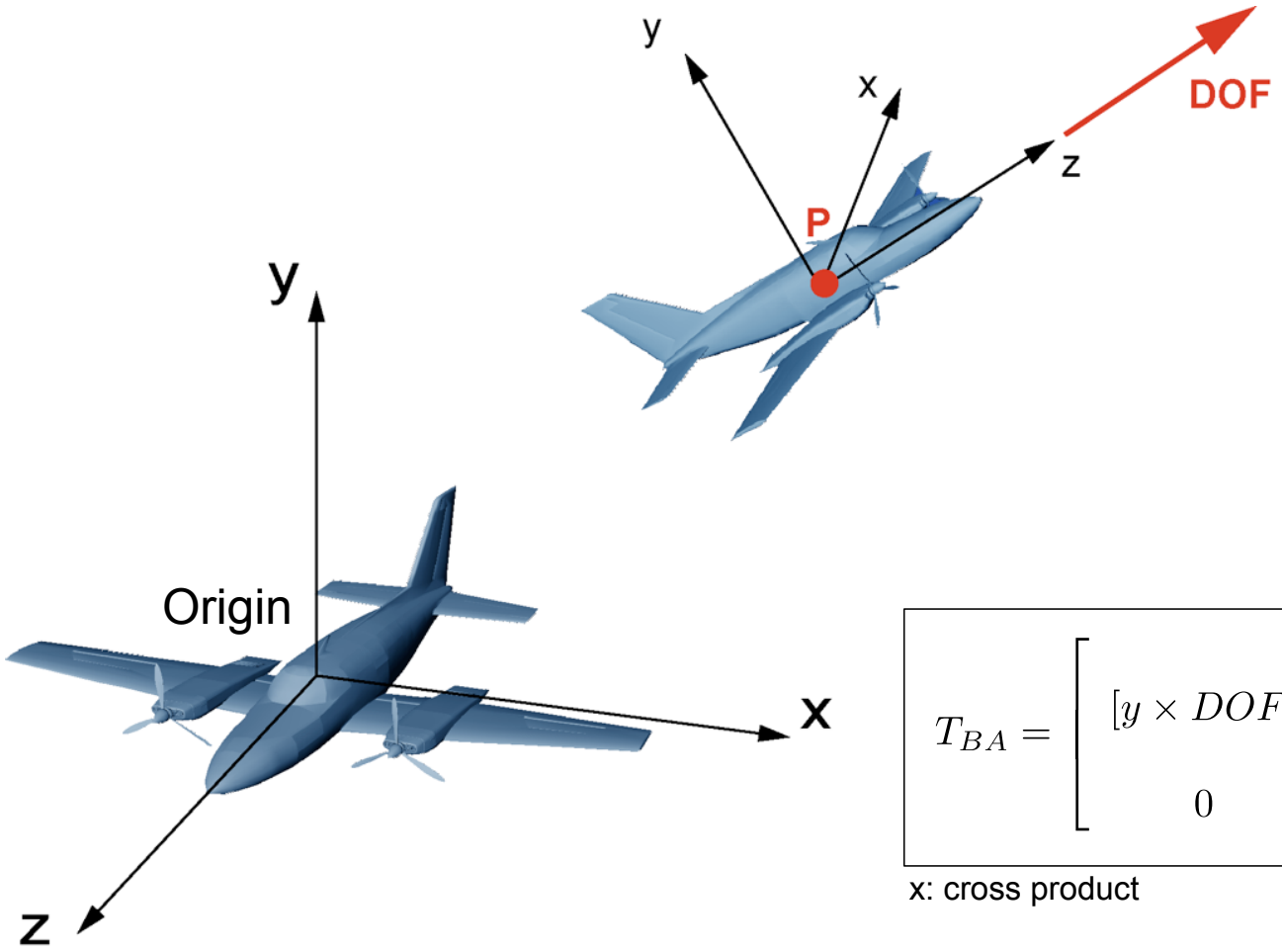
$$\mathbf{M} = (\mathbf{A} \cdot \mathbf{B}) \cdot \mathbf{v})$$

But the result may be an unexpected one when you mix translation, rotation, and scale matrices.

The final appearance of a 3D model depends greatly on the order of the matrices.

The multiplication is not commutative.

Transformation of Coordinate Systems



$$T_{BA} = \begin{bmatrix} [y \times DOF] & |DOF \times [y \times DOF]| & |DOF| & P_x \\ & & & P_y \\ & & & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

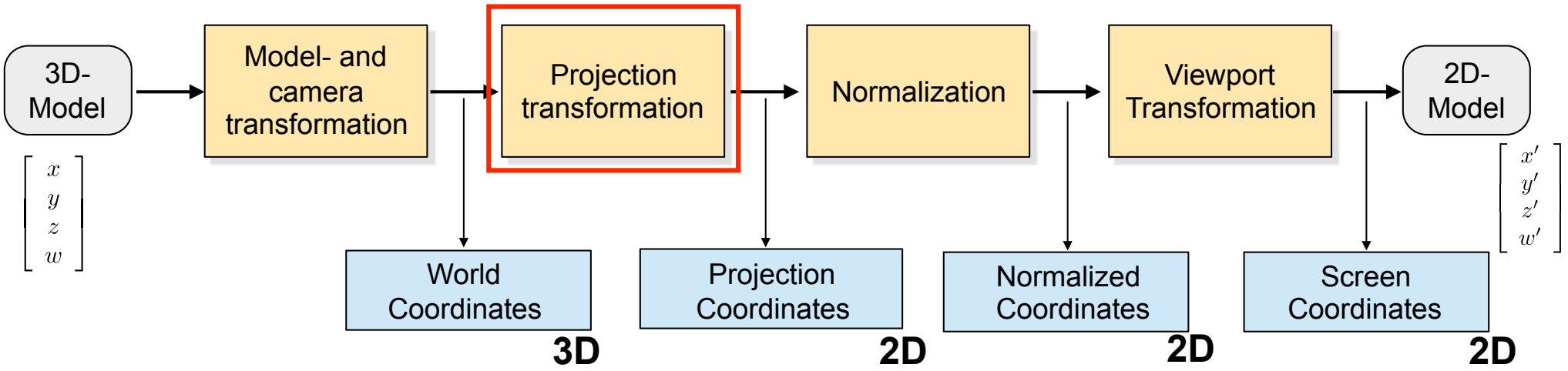
x: cross product

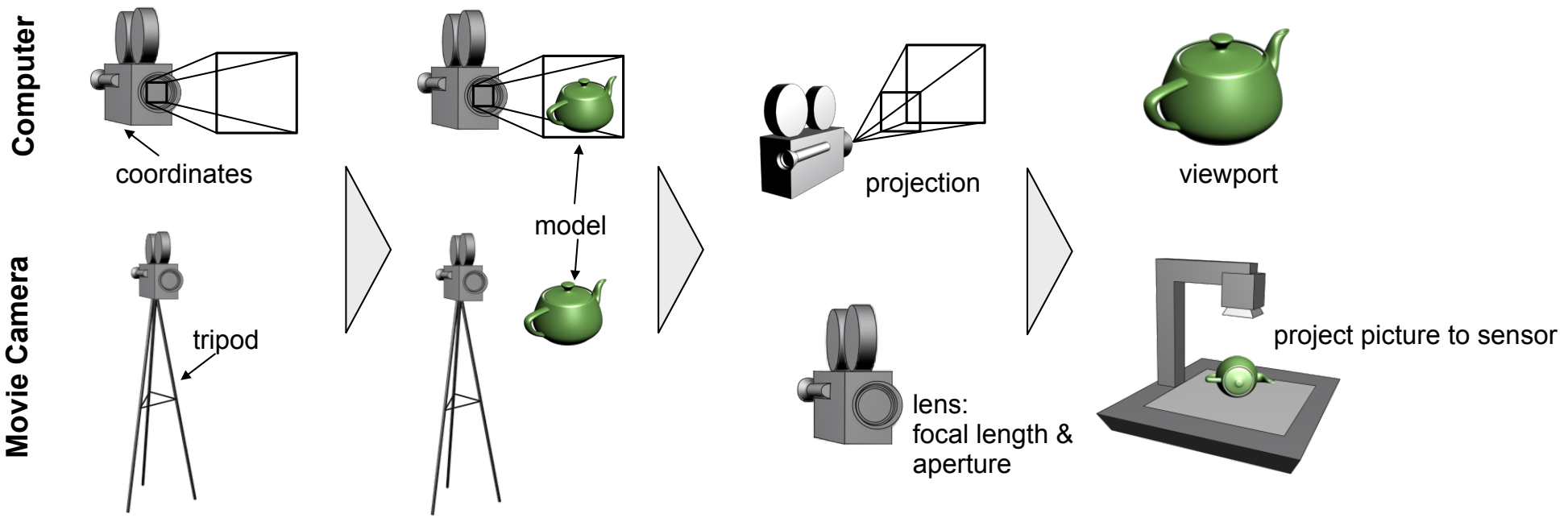
Translation of a airplane to the point P with alignment DOF (DOF.: Direction of Flight)

ARLAB

Viewing in OpenGL

3D Transformation Pipeline

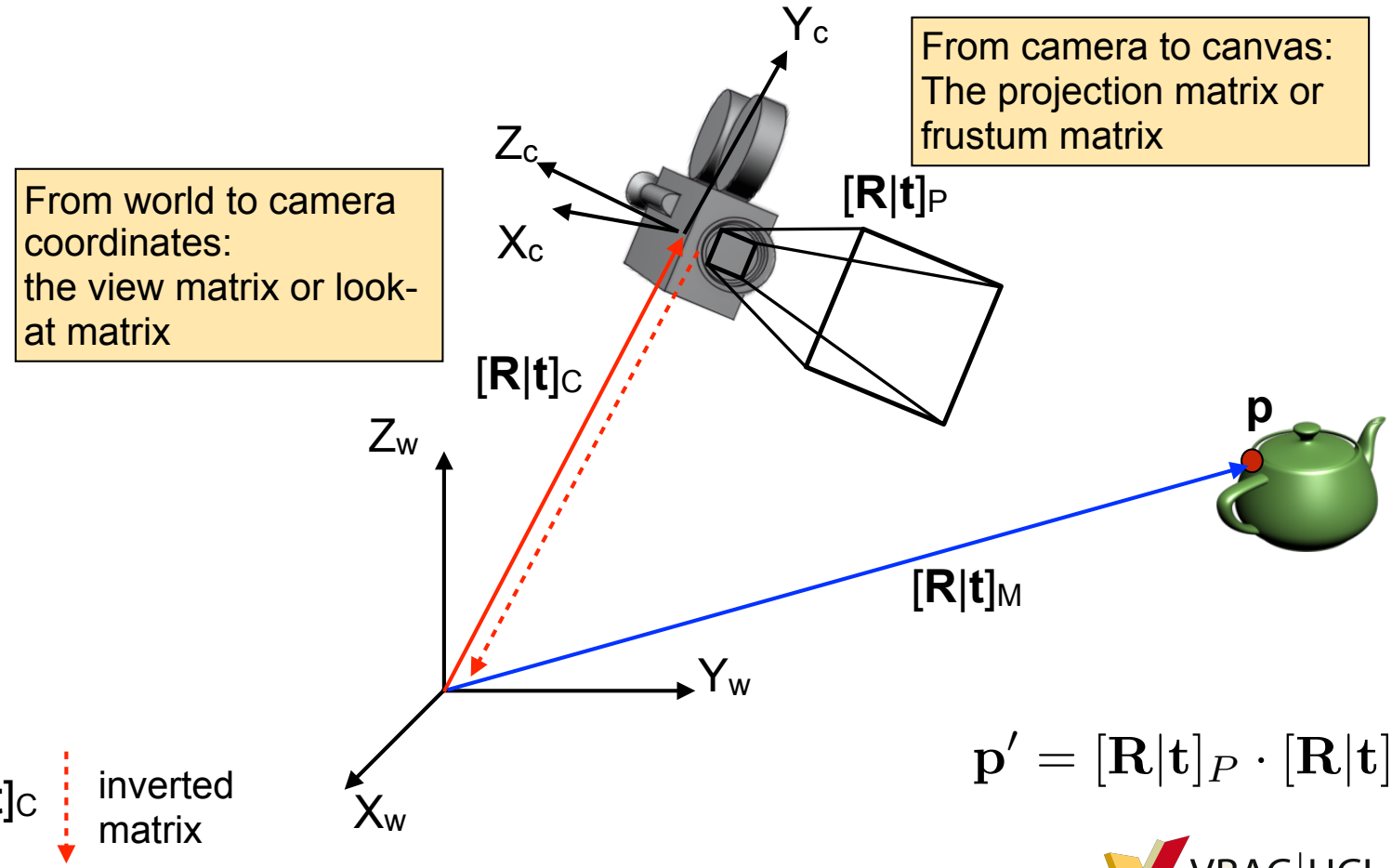




Camera Model

The camera model is used to transform every point in 3D point from 3D world coordinate system to the 2D image coordinate system. The camera model incorporates two matrices:

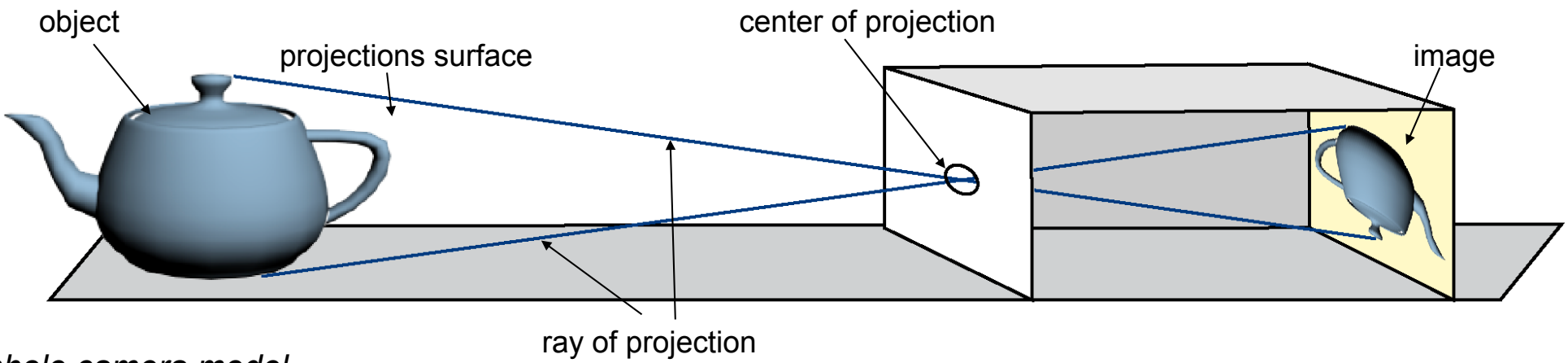
1. The view matrix or look-at matrix, which describes the position and alignment of the camera
2. The projection matrix, which defines the projection from 3D to 2D; it mimics a lens.



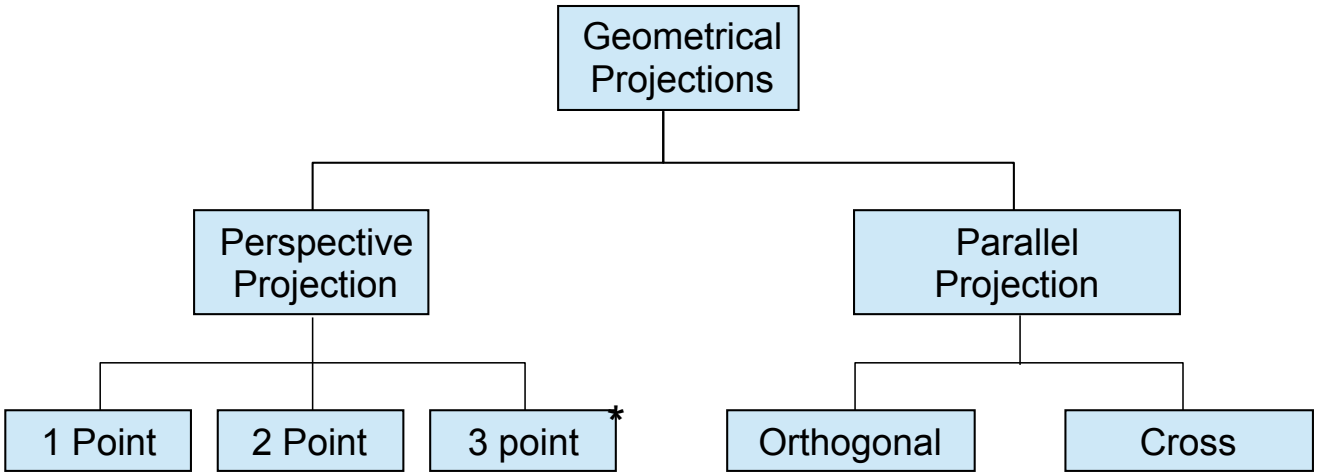
$$\mathbf{p}' = [\mathbf{R}|\mathbf{t}]_P \cdot [\mathbf{R}|\mathbf{t}]_C^{-1} \cdot [\mathbf{R}|\mathbf{t}]_M \cdot \mathbf{p}$$

[R|t]C inverted matrix

Camera Model - Projection



Pinhole camera model



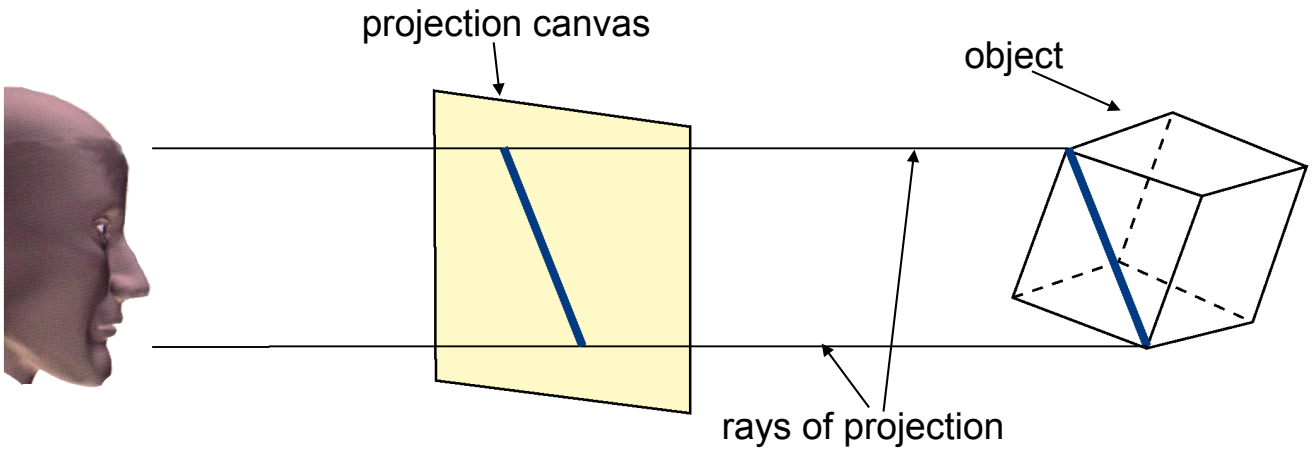
Geometric projections in Computer Graphics

In computer graphics (CG), a projection is considered of a 3-dimensional object onto a 2-dimensional canvas. Therefore, CG uses so called **geometric projections**, which calculate 2D surface coordinates for a 3D object.

Affine Transformation:
A affine projection is a linear 3D transformation (rotation and translation), which preserves the ratio and geometric structure of all lines in the image, but not the length of lines.

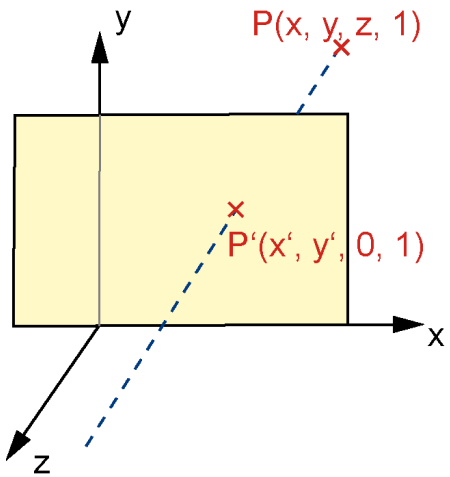
* number of vanishing points

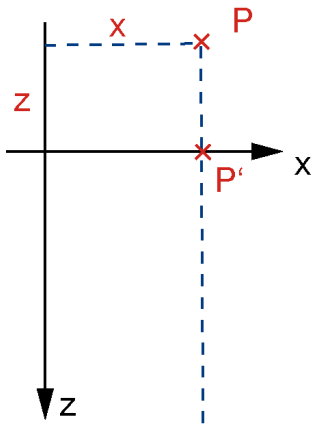
Orthogonal Projection

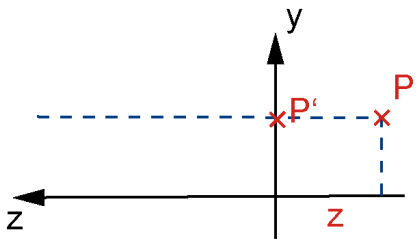


Orthogonal projection

From a mathematical point of view, a parallel projection is a theoretical projection where the rays of projections vanish at a vanishing point at infinite distance. Thus, the projection rays are considered as parallel.
The projection is an affine projection: aspect ratios, distances, and angles are kept.



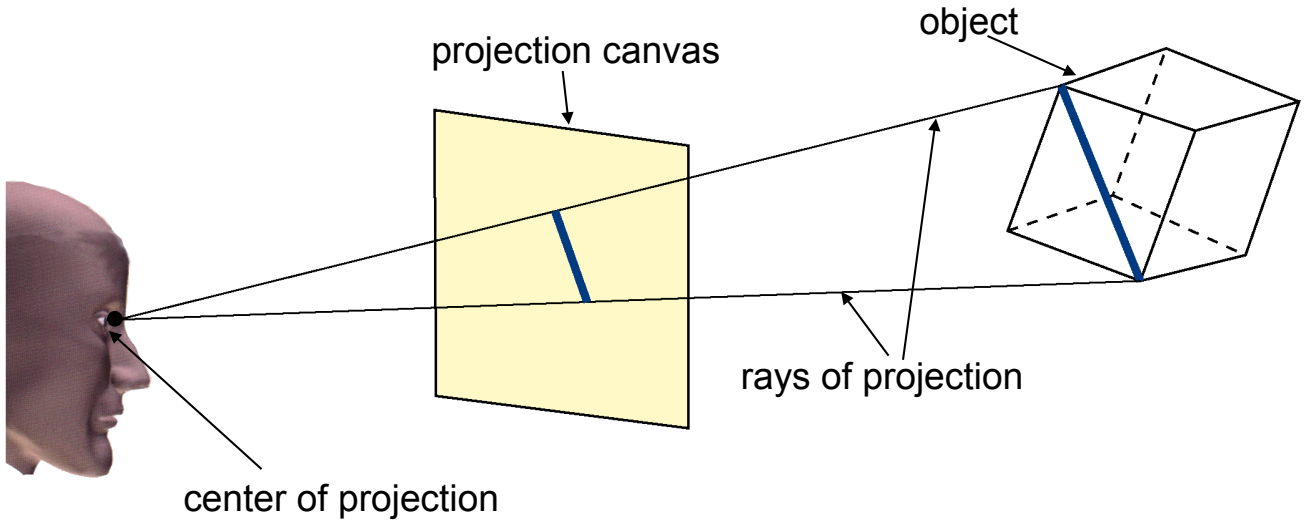




$$x' = x$$
$$y' = y$$
$$z' = 0$$

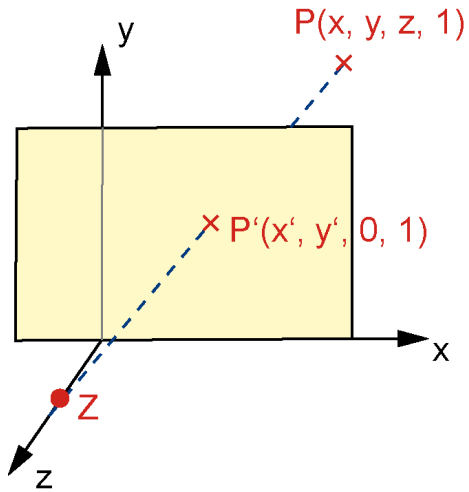
Concept of orthogonal projection

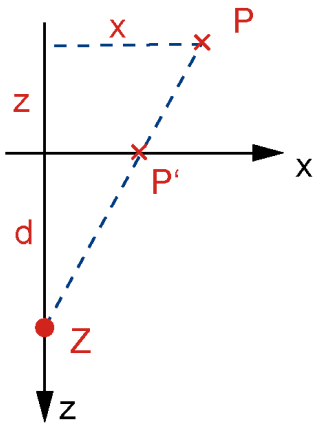
Perspective Projection

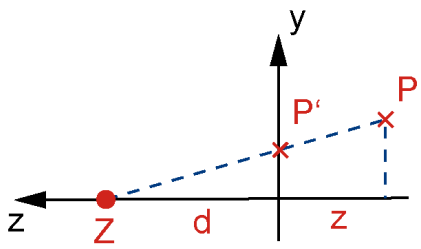


perspective projection

A perspective projection maps all objects realistically to a 2D surface. It correlates with the human viewing. But, it is not a affine projection: dimensions, aspect ratios, and angles are changed during the projection.







$$x' = \frac{d \cdot x}{z + d}$$

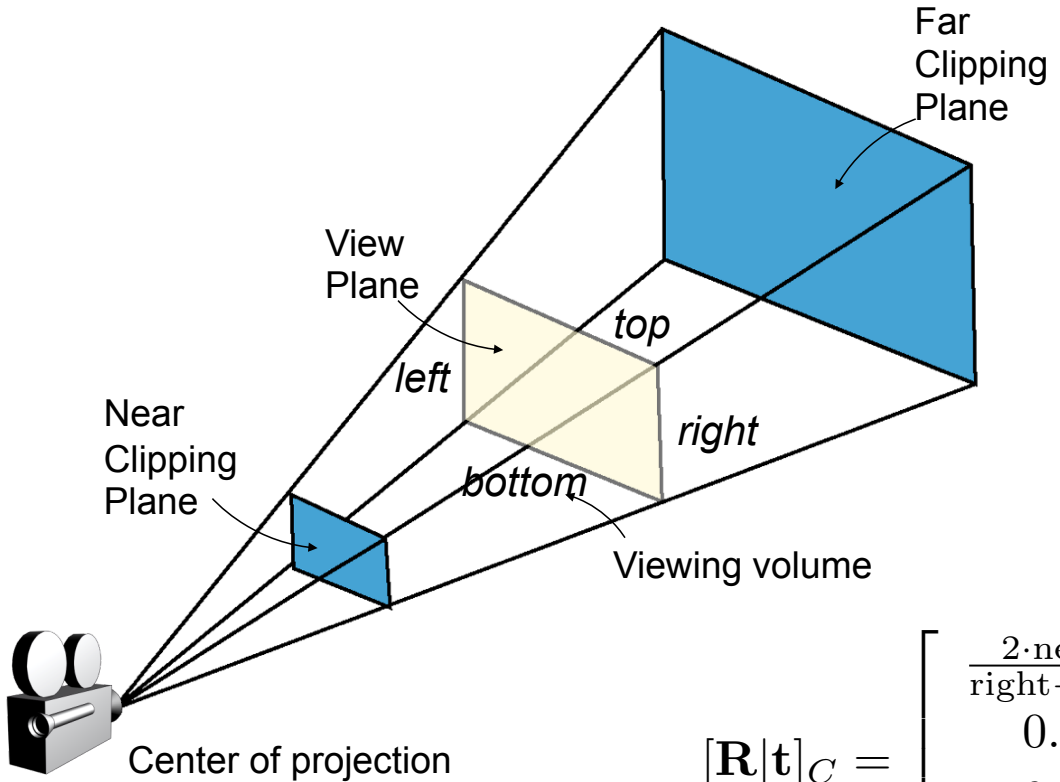
$$y' = \frac{d \cdot y}{z + d}$$

$$z' = 0$$

Concept of the perspective projection

Perspective Projection -
Viewing Volume

The perspective projection is represented as projection matrix which describes a truncated pyramid



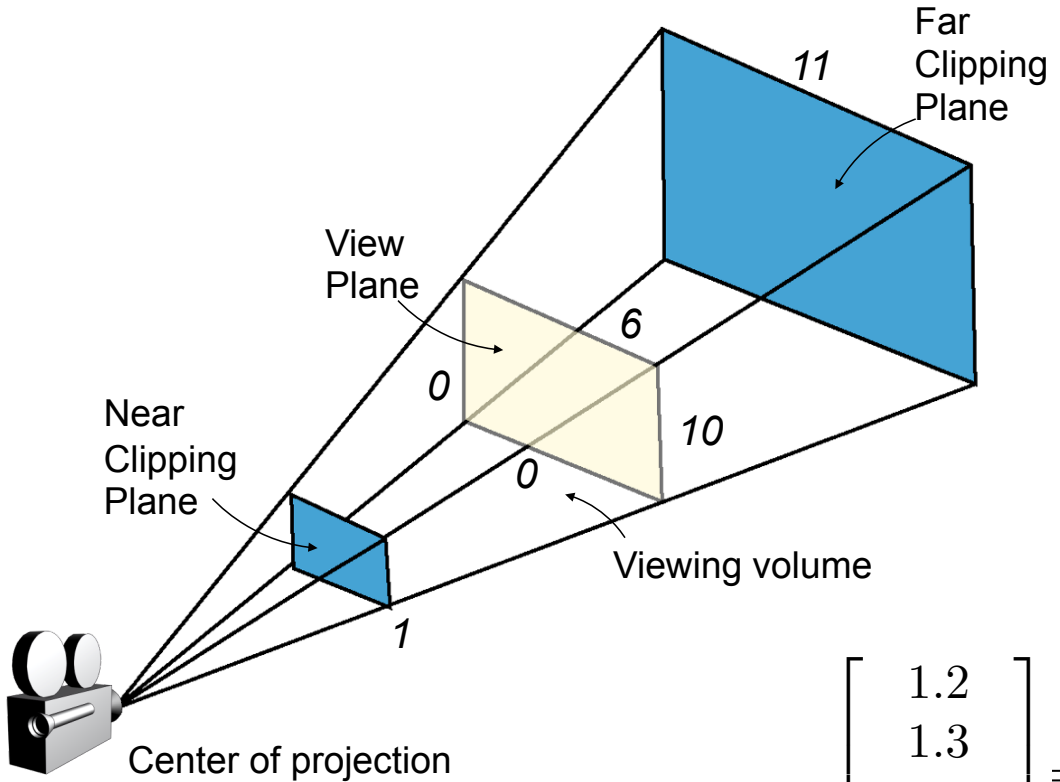
The viewing volume of a perspective.

$$[\mathbf{R}|\mathbf{t}]_C = \begin{bmatrix} \frac{2\cdot\text{near}}{\text{right}-\text{left}} & 0.0 & \frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0.0 \\ 0.0 & \frac{2\cdot\text{near}}{\text{top}-\text{bottom}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0.0 \\ 0.0 & 0.0 & \frac{\text{near}+\text{far}}{\text{near}-\text{far}} & \frac{2\cdot\text{near}\cdot\text{far}}{\text{near}-\text{far}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

Coordinates in world coordinate system

Perspective Projection -
Viewing Volume

The perspective projection is represented as projection matrix which describes a truncated pyramid



The viewing volume of a perspective.

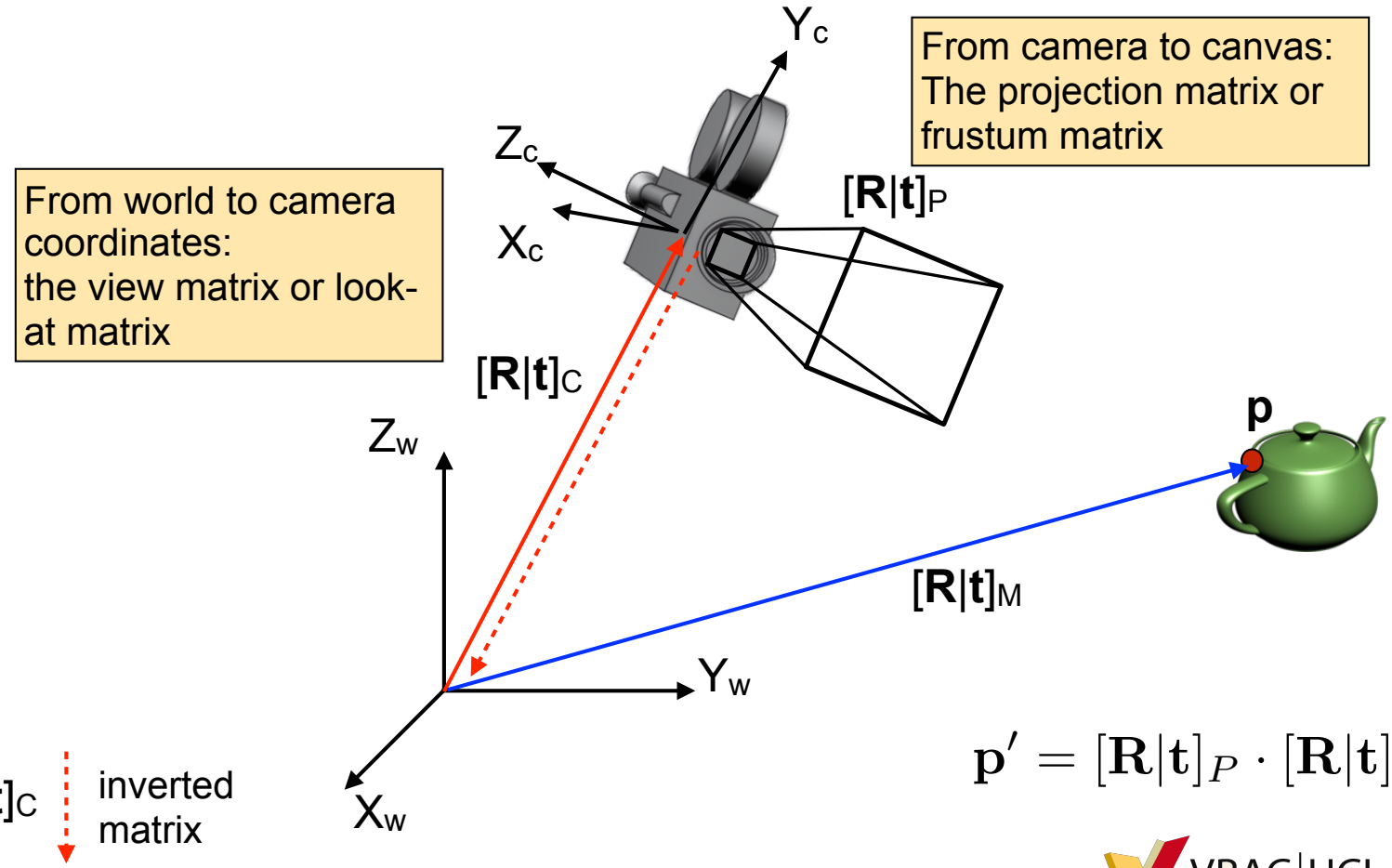
$$\begin{bmatrix} 1.2 \\ 1.3 \\ 1.1 \\ -1.0 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.3 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.1 & -2.2 \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix} \cdot \begin{bmatrix} 10.0 \\ 10.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Coordinates in world coordinate system

Camera Model

The camera model is used to transform every point in 3D point from 3D world coordinate system to the 2D image coordinate system. The camera model incorporates two matrices:

1. The view matrix or look-at matrix, which describes the position and alignment of the camera
2. The projection matrix, which defines the projection from 3D to 2D; it mimics a lens.



$$\mathbf{p}' = [\mathbf{R}|\mathbf{t}]_P \cdot [\mathbf{R}|\mathbf{t}]_C^{-1} \cdot [\mathbf{R}|\mathbf{t}]_M \cdot \mathbf{p}$$

[R|t]C inverted matrix

Camera Model -
Camera Location

The camera location is represented as a matrix, named the look-at matrix or view matrix.

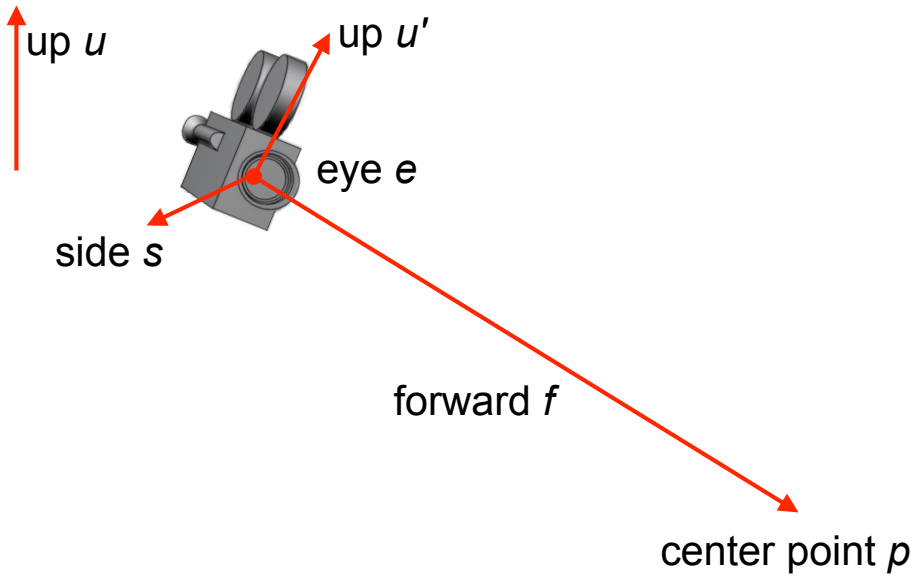
It places your camera into the 3D world and aligns it towards the object you want to look to.

We describe the position with three vectors:

eye: the origin

center: the point of interest we look to

up: the up-vector, required to describe a direction and to prevent spinning.



We have to construct an orthonormal coordinate system

First, compute a forward vector:

$$f = \frac{p - e}{||p - e||}$$

Second, construct a side vector using the cross product

$$s = f \times u$$

Construct the correct up vector *u'*

$$u' = s \times f$$

Camera Model -
Camera Location

The camera location is represented as a matrix, named the look-at matrix or view matrix.

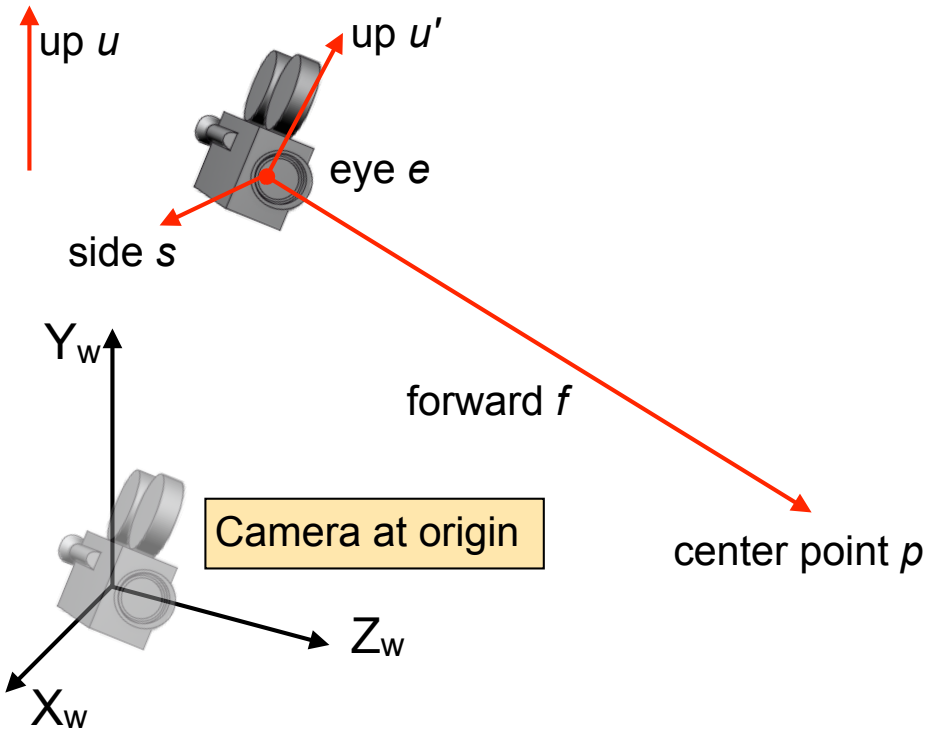
It places your camera into the 3D world and aligns it towards the object you want to look to.

We describe the position with three vectors:

eye: the origin

center: the point of interest we look to

up: the up-vector, required to describe a direction and to prevent spinning.



Put everything into a matrix

$$\mathbf{R} = \begin{bmatrix} s.x & u'.x & f.x & 0.0 \\ s.y & u'.y & f.y & 0.0 \\ s.z & u'.z & f.z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Camera at origin

Camera Model -
Camera Location

The camera location is represented as a matrix, named the look-at matrix or view matrix.

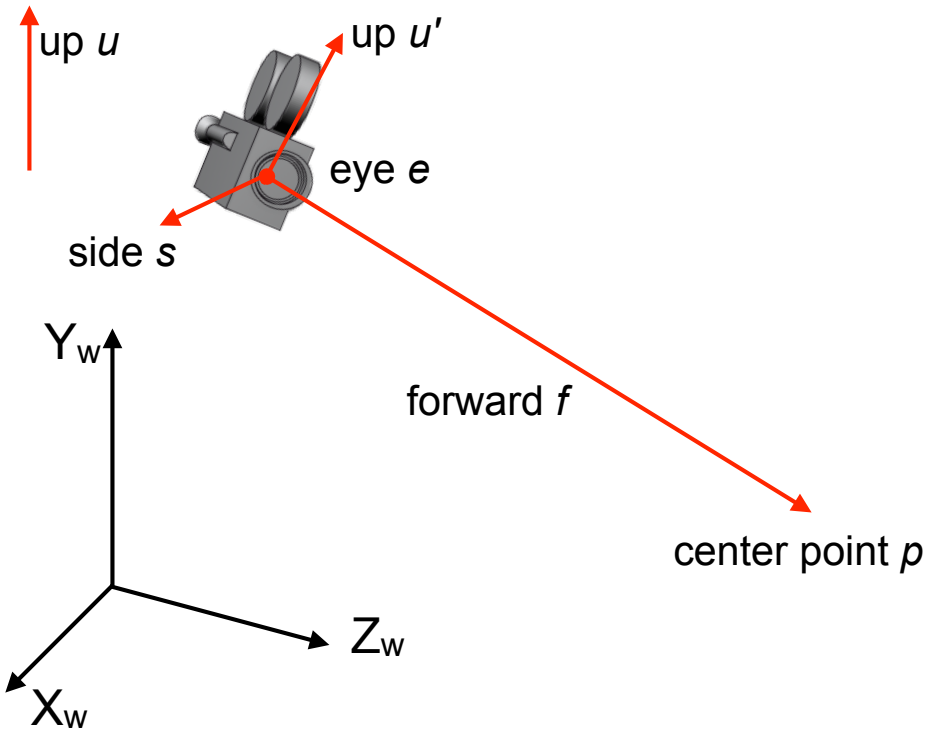
It places your camera into the 3D world and aligns it towards the object you want to look to.

We describe the position with three vectors:

eye: the origin

center: the point of interest we look to

up: the up-vector, required to describe a direction and to prevent spinning.



Put everything into a matrix

$$\mathbf{R} = \begin{bmatrix} s.x & u'.x & f.x & 0.0 \\ s.y & u'.y & f.y & 0.0 \\ s.z & u'.z & f.z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

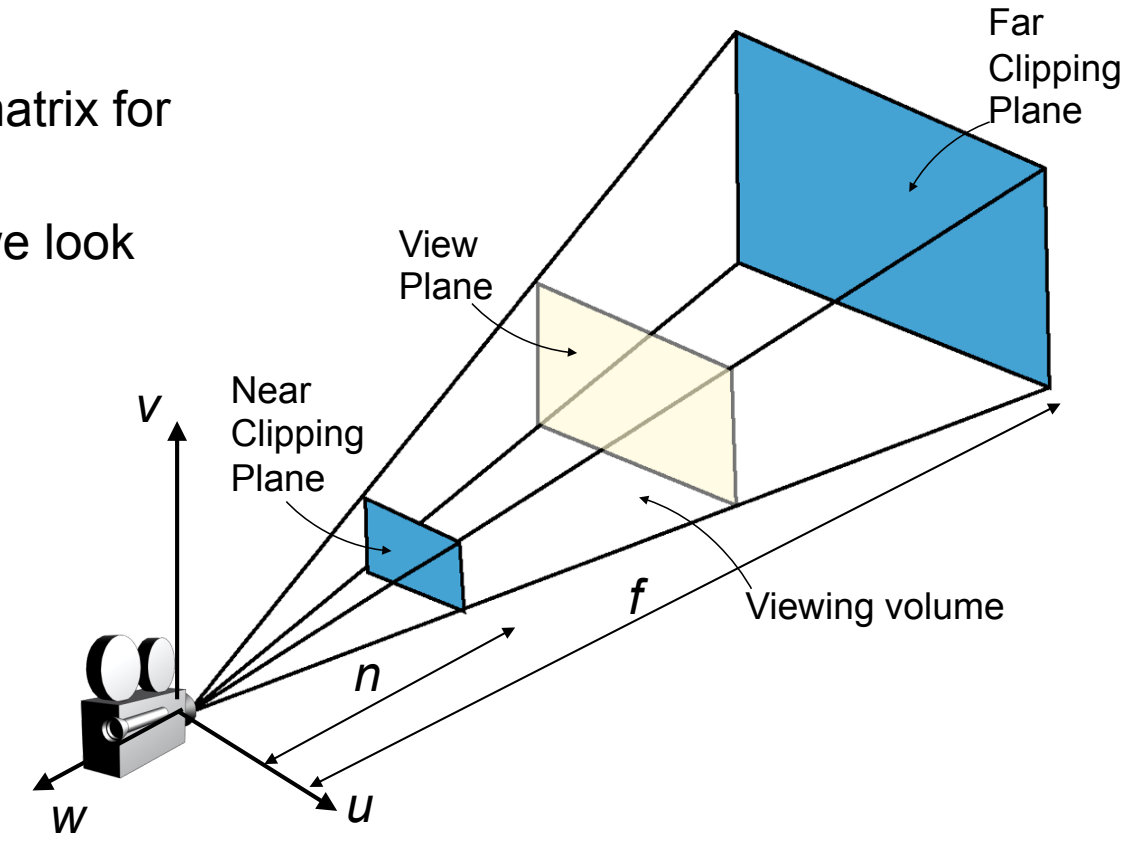
And we can move the origin using the eye vector:

$$\mathbf{R} = \begin{bmatrix} s.x & u'.x & f.x & -e.x \\ s.y & u'.y & f.y & -e.x \\ s.z & u'.z & f.z & -e.x \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Projective Transformation Summarized

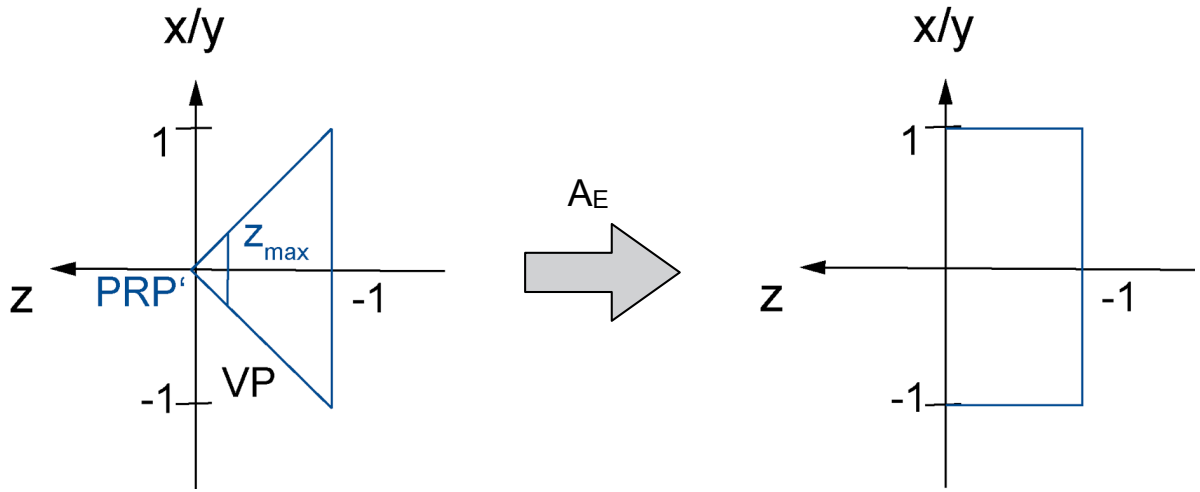
Summarized, the projection matrix for perspective projections.

Note, in computer graphics, we look into the negative direction.



$$\mathbf{M}_{per} = [\mathbf{R}|\mathbf{t}]_c = \begin{bmatrix} \frac{2\cdot\text{near}}{\text{right}-\text{left}} & 0.0 & \frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0.0 \\ 0.0 & \frac{2\cdot\text{near}}{\text{top}-\text{bottom}} & \frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0.0 \\ 0.0 & 0.0 & \frac{\text{near}+\text{far}}{\text{near}-\text{far}} & \frac{2\cdot\text{near}\cdot\text{far}}{\text{near}-\text{far}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & -P_x \\ v_x & v_y & v_z & -P_y \\ w_x & w_y & w_z & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Normalization



Open the frustum to a cuboid

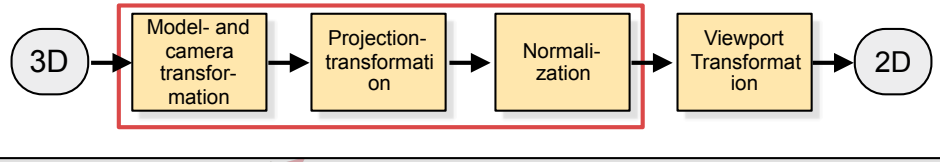
The normalization step transfers all data inside the frustum of a pyramid into a unit cuboid with edge dimensions
-1 ≤ x ≤ 1, -1 ≤ y ≤1, and 0 ≤ z ≤ -1.

Step 5: open the frustum of a pyramid to a cuboid

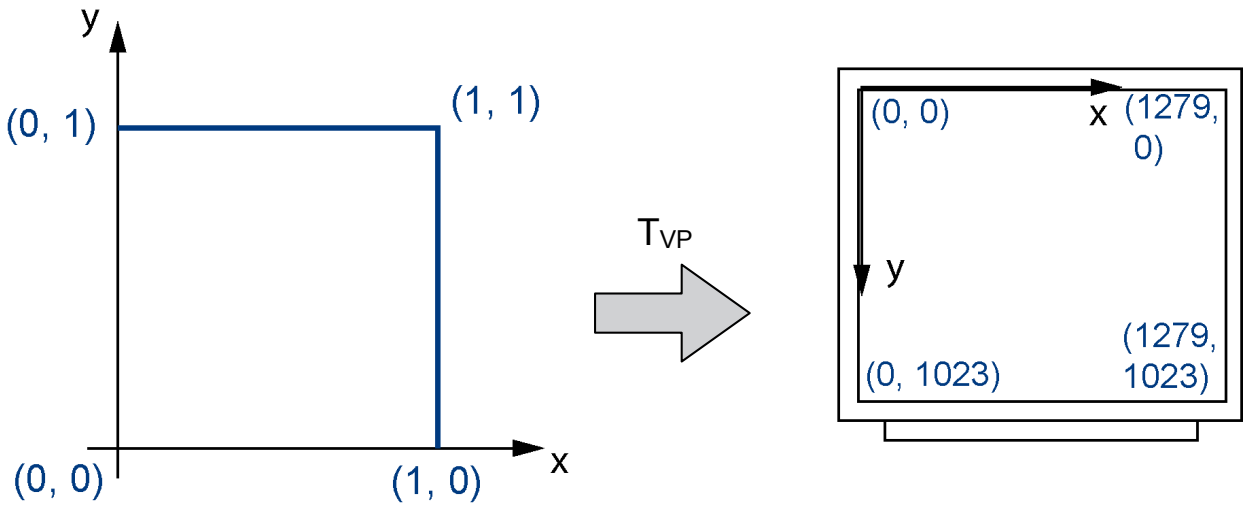
$$A = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & \frac{-1}{1+t_{max}} & \frac{z_{max}}{1+t_{max}} \\ 0 & 0 & 1 & 0 \end{bmatrix}, \text{ with } \quad z_{max} = \frac{d-F}{d+B} \quad \text{und} \quad z_{max} \neq -1$$

$$A_E = \begin{bmatrix} -\frac{1}{1} & 0 & \frac{1}{1} & 0 \\ 0 & -\frac{1}{2} & \frac{1}{1} & 0 \\ 0 & 0 & \frac{-1}{1+t_{max}} & \frac{z_{max}}{1+t_{max}} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$M_{Per} = A_E \cdot S_2 \cdot S_1 \cdot SH_{Per} \cdot R_z \cdot R_x \cdot R_y \cdot T$$



Viewport-Transformation



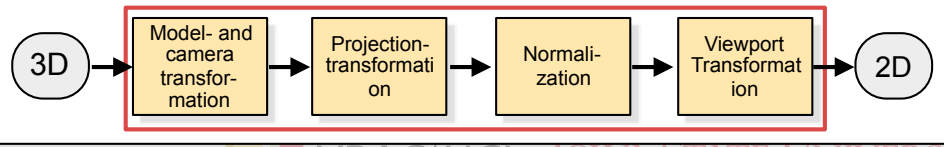
The viewport transformation maps the visual data in the unit cuboid to the pixel ranges in x and y of the screen.

Transformation of unit coordinates into screen coordinates. The example screen has a resolution of 1280 x 1024 pixels

Step 6: presentation of a 3-dimensional unit cuboid on a 2D screen

$$T_{VP} = \begin{bmatrix} x_{max} & 0 & 0 & 0 \\ 0 & -y_{max} & 0 & y_{max} \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{Per} = T_{VP} \cdot A_E \cdot S_2 \cdot S_1 \cdot SH_{Per} \cdot R_z \cdot R_x \cdot R_y \cdot T$$

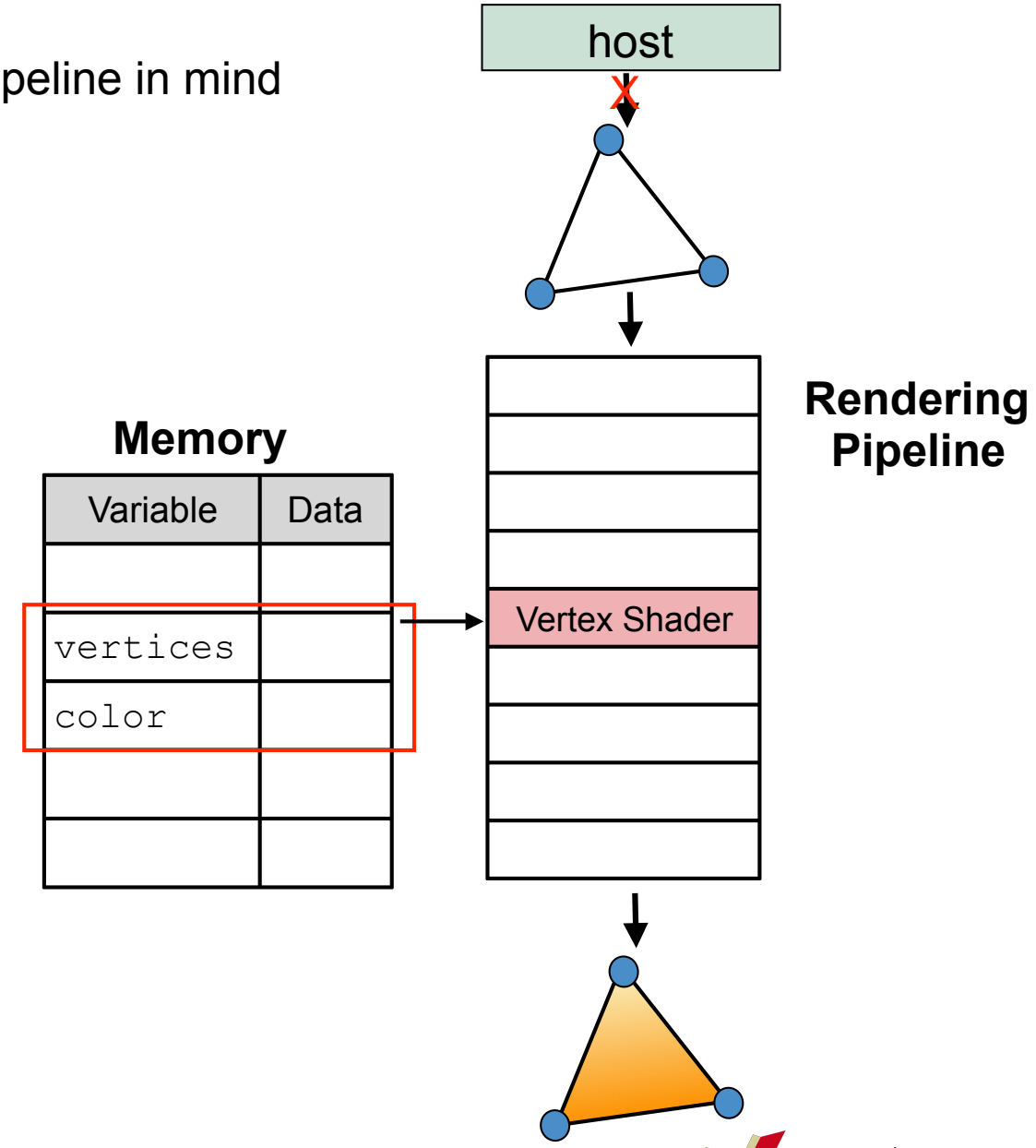


ARLAB

Viewing and Transformation in OpenGL

Pipeline model

Keep the rendering pipeline in mind



Example Shader Code

Shader source code that you can find in our example programs

```
static const string vs_string =
    "#version 410 core                                                 \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                    \n"
    "uniform mat4 viewMatrix;                                           \n"
    "uniform mat4 modelMatrix;                                          \n"
    "in vec3 in_Position;                                               \n"
    "                                                                   \n"
    "in vec3 in_Color;                                                  \n"
    "out vec3 pass_Color;                                               \n"
    "                                                                  \n"
    "void main(void)                                                   \n"
    "{                                                                 \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);  \n"
    "    pass_Color = in_Color;                                         \n"
    "}                                                                 \n";
```

Example Shader Code

The matrices are defined as uniform variables

```
static const string vs_string =
    "#version 410 core                                                 \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                    \n"
    "uniform mat4 viewMatrix;                                           \n"
    "uniform mat4 modelMatrix;                                          \n"
    "in vec3 in_Position;                                               \n"
    "                                                                   \n"
    "in vec3 in_Color;                                                  \n"
    "out vec3 pass_Color;                                               \n"
    "                                                                  \n"
    "void main(void)                                                   \n"
    "{                                                                 \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);  \n"
    "    pass_Color = in_Color;                                         \n"
    "}                                                                 \n";
```

- uniform: qualifier to declare the variable as a shared variable, shared between host computer and graphics card.
- mat4: a GLSL 4x4 matrix

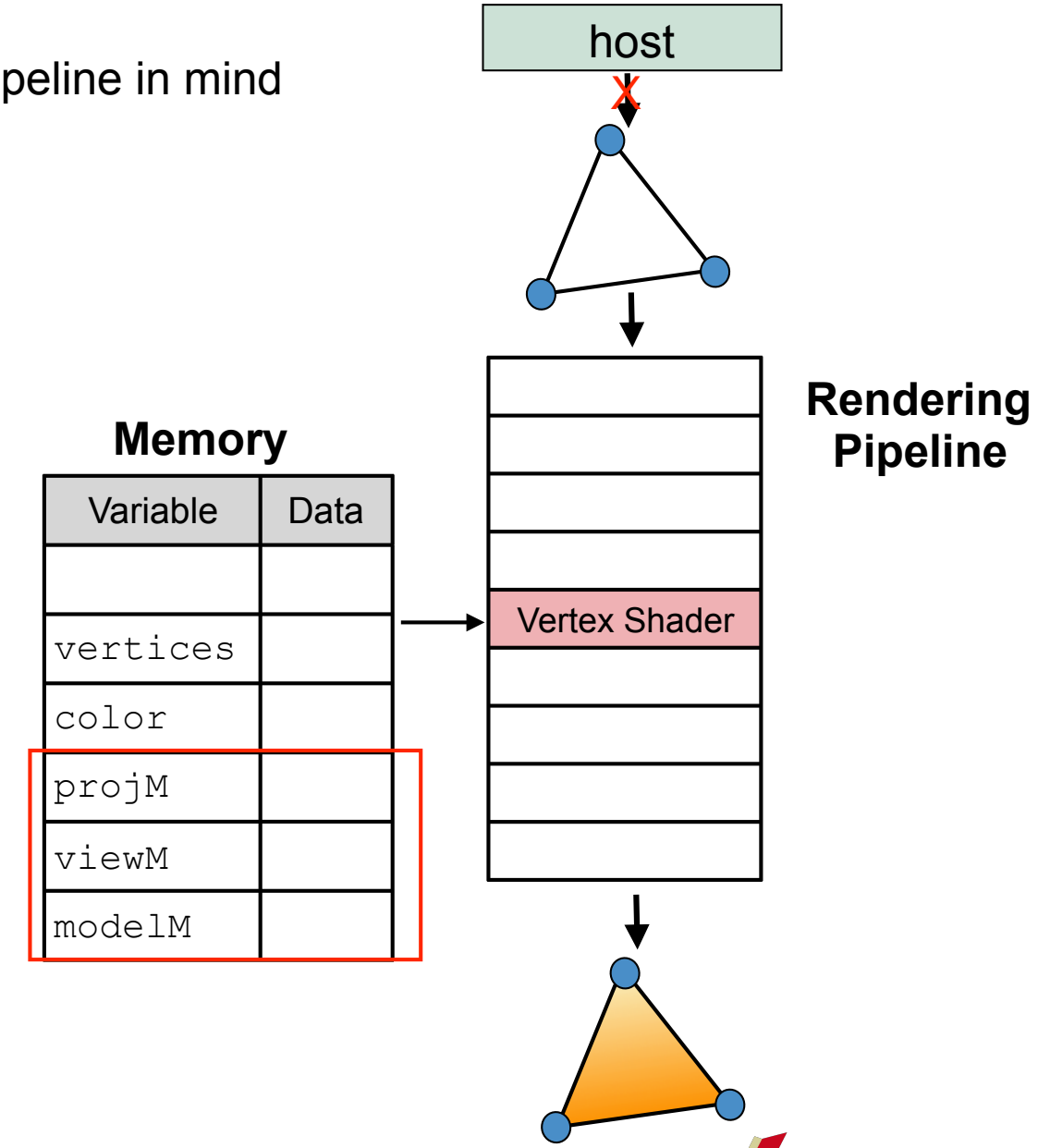
$$\mathbf{M} = \begin{bmatrix} r_{01} & r_{02} & r_{03} & t_x \\ r_{11} & r_{12} & r_{13} & t_y \\ r_{21} & r_{22} & r_{23} & t_z \\ r_{31} & r_{32} & r_{33} & 1 \end{bmatrix}$$

Terms

- **Model matrix:** the matrix that we use to transform the 3D model from local coordinates to global coordinates
- **View matrix:** the matrix we use to transform the camera
- **ModelView matrix:** the product of both, since in computer graphics, it does not matter whether one move the camera or the model.
- **Projection matrix:** the projection from 3D to a 2D point.

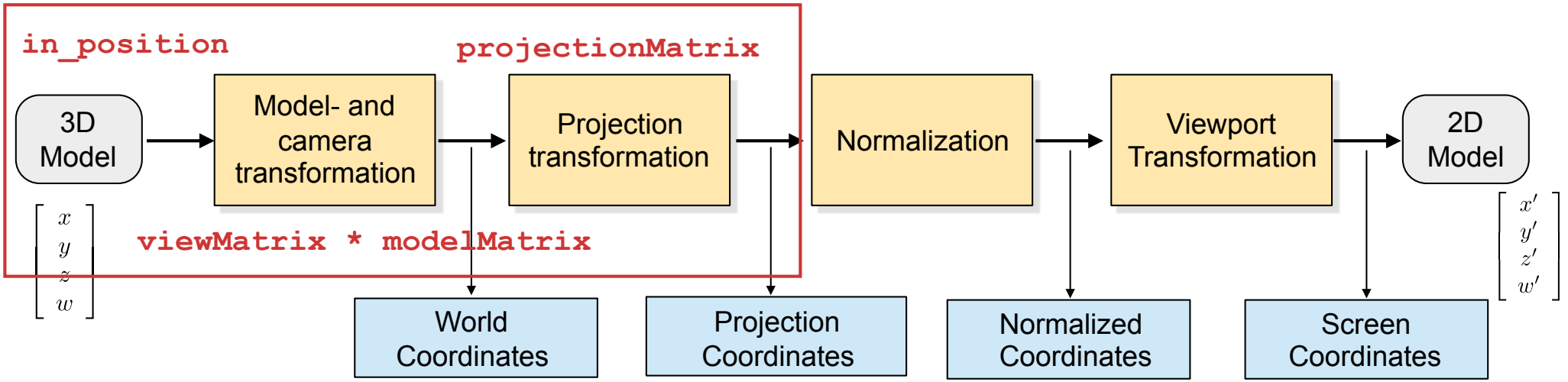
Pipeline model

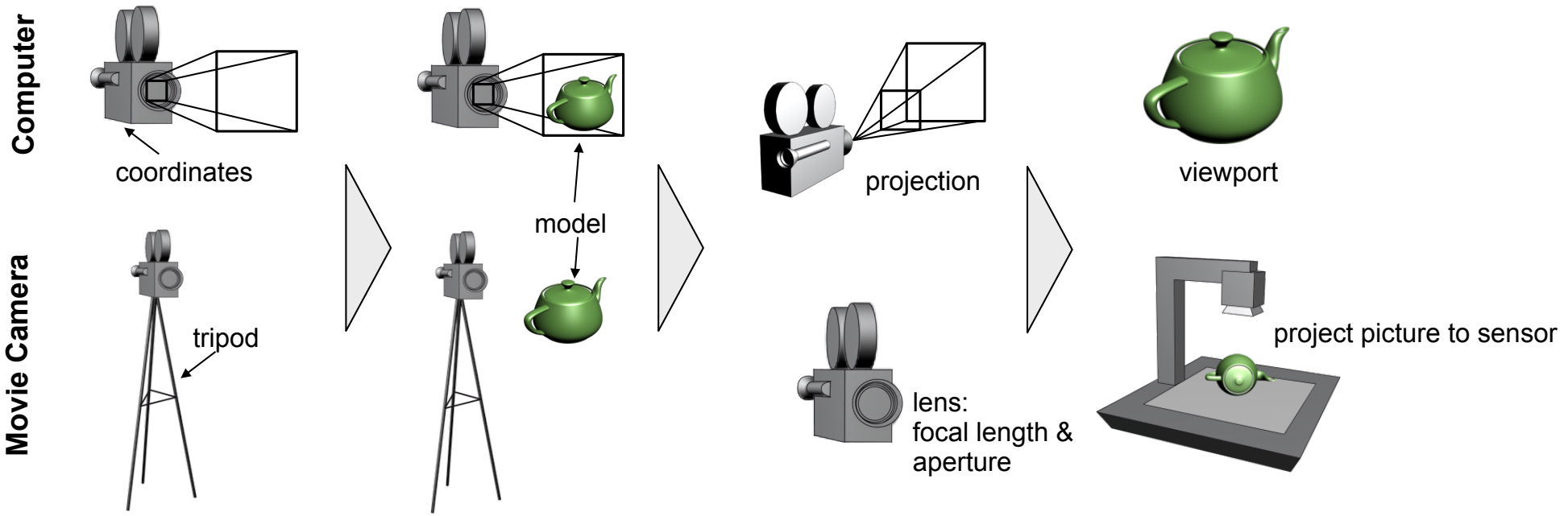
Keep the rendering pipeline in mind



Viewing in 3D

ARLAB





VRAC | HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Example Shader Code

```
static const string vs_string =
    "#version 410 core                                                 \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                    \n"
    "uniform mat4 viewMatrix;                                           \n"
    "uniform mat4 modelMatrix;                                          \n"
    "in vec3 in_Position;                                               \n"
    "                                                                   \n"
    "in vec3 in_Color;                                                  \n"
    "out vec3 pass_Color;                                               \n"
    "                                                                  \n"
    "void main(void)                                                   \n"
    "{                                                                 \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0); \n"
    "    pass_Color = in_Color;                                         \n"
    "}                                                                 \n";
```

This is the modelview projection calculation

- gl_Position: the built-in *varying* variable to pass the position information the the next step.
  Note, gl_Position is a vector with 4 elements [x, y, z, 1]

GLM Transformation Matrix

We use the glm transformation matrix
glm::mat4
to represent a 4x4 matrix.

$$\mathbf{M} = \begin{bmatrix} r_{01} & r_{02} & r_{03} & t_x \\ r_{11} & r_{12} & r_{13} & t_y \\ r_{21} & r_{22} & r_{23} & t_z \\ r_{31} & r_{32} & r_{33} & 1 \end{bmatrix}$$

Glm include files:

#include <glm/glm.hpp>

#include <glm/gtc/matrix_transform.hpp>

in our example code:

```
glm::mat4 projectionMatrix; // Store the projection matrix
glm::mat4 viewMatrix; // Store the view matrix
glm::mat4 modelMatrix; // Store the model matrix
```

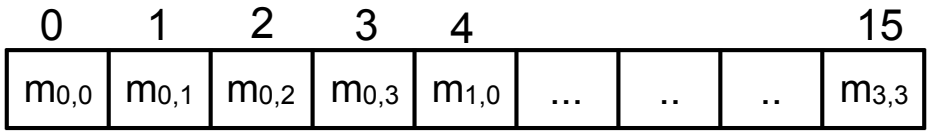
GLM Transformation Matrix
Row-major order

ARLAB

We distinguish between **row-major order** and **column-major order.**

row-major order

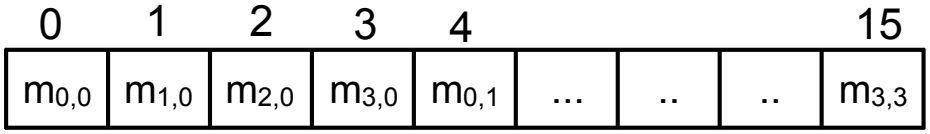
$$M = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$$



C/C++ uses this order

column-major order

$$M = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$$



OpenGL use this order

But, this is just a convention and the programmer can choose the right convention

```
{ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, transX, transY, transZ, 1 }.
```

GLM Transformation Matrix
Row-major order

ARLAB

An OpenGL matrix would look such as

$$M = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
{ 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, X, Y, Z, 1 }.
```

This is a matter of convention for how you interpret matrices in your program.

For OpenGL

GLfloat matrix[16]	OpenGL-friendly
GLfloat matrix[4][4]	May not be as convenient

GLM Transformation Matrix
Row-major order

ARLAB

OpenGL-friendly

```
GLfloat matrix[16] = { 1, 0, 0, 0,
                       0, 1, 0, 0,
                       0, 0, 1, 0,
                       X, Y, Z, 1 }.
```

x-column
y-column
z-column

Not OpenGL-friendly

```
GLfloat matrix[4][4] = { {1, 0, 0, 0},
                         {0, 1, 0, 0},
                         {0, 0, 1, 0},
                         {X, Y, Z, 1} }.
```

x-row
y-row
z-row

GLM Vector 3

We use the glm transformation matrix

glm::vec3

to represent a vector of size 3, and

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

glm::vec4

to represent a vector of size 4
(homogenous space)

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

Example:

```
glm::vec3 p = glm::vec3(0.0f, 1.0f, 0.0f);
```

Transformations

ARLAB

Translation

```
detail::tmat4x4< T > translate (T x, T y, T z)
```

Parameters:

- x, y, z: Specify the x, y, and z coordinates of a translation vector.

Rotation

```
detail::tmat4x4< T > rotate (T angle, T x, T y, T z)
```

Parameters:

- angle: Specifies the angle of rotation, in degrees.
- x, y, z: Specify the x, y, and z coordinates of a vector, respectively.

Scaling

```
detail::tmat4x4< T > scale (T x, T y, T z)
```

Parameters:

- x, y, z: Specify scale factors along the x, y, and z axes, respectively.

Datatype:

`detail::tmat4x4< T >` this is a **glm::mat4**

http://glm.g-truc.net/0.9.0/api/a00192.html

Transformation Example

ARLAB

Order is important!!!

When writing code, always multiply a matrix by a vector and read the sequence of transformations in reverse order (from right to left = reverse).

```
glm::mat4 translation_matrix = glm::translate(glm::vec3(2.0f, 0.0f, 0.0f));

glm::mat4 rotation_matrix = glm::rotate( 0.57f, glm::vec3(0.0f, 0.0f, 1.0f));

glm::vec4 input_vertex = glm::vec3(1.0f, 2.0f, 4.0f, 1.0f);


glm::vec4 out_vertex = translation_matrix * rotation_matrix * input_vertex;
```

rotation first, translation second

or

```
glm::mat4 composite_matrix = translation_matrix * rotation_matrix;
```

rotation first, translation second

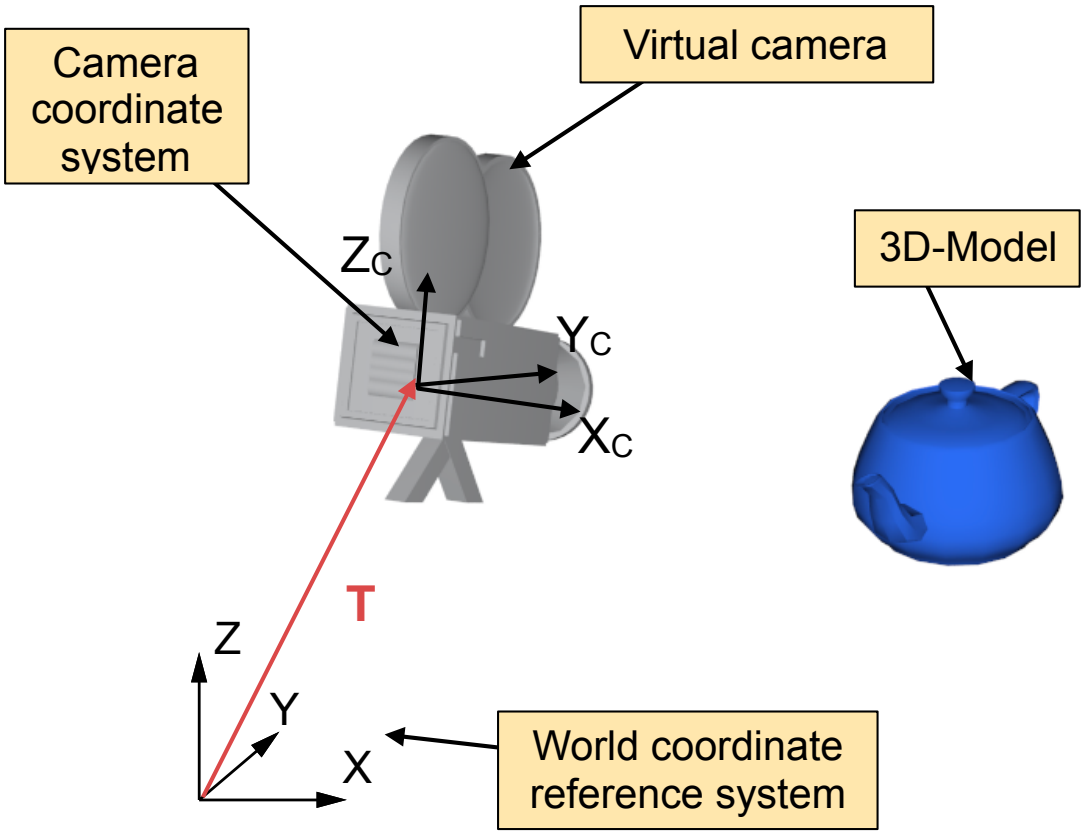
```
glm::vec4 out_vertex = composite_matrix * input_vertex;
```

Both will yield the same result!

Viewing in OpenGL

To Do:

1. Set a projection with glm::perspective
2. Locate and align the virtual camera with glm::lookAt



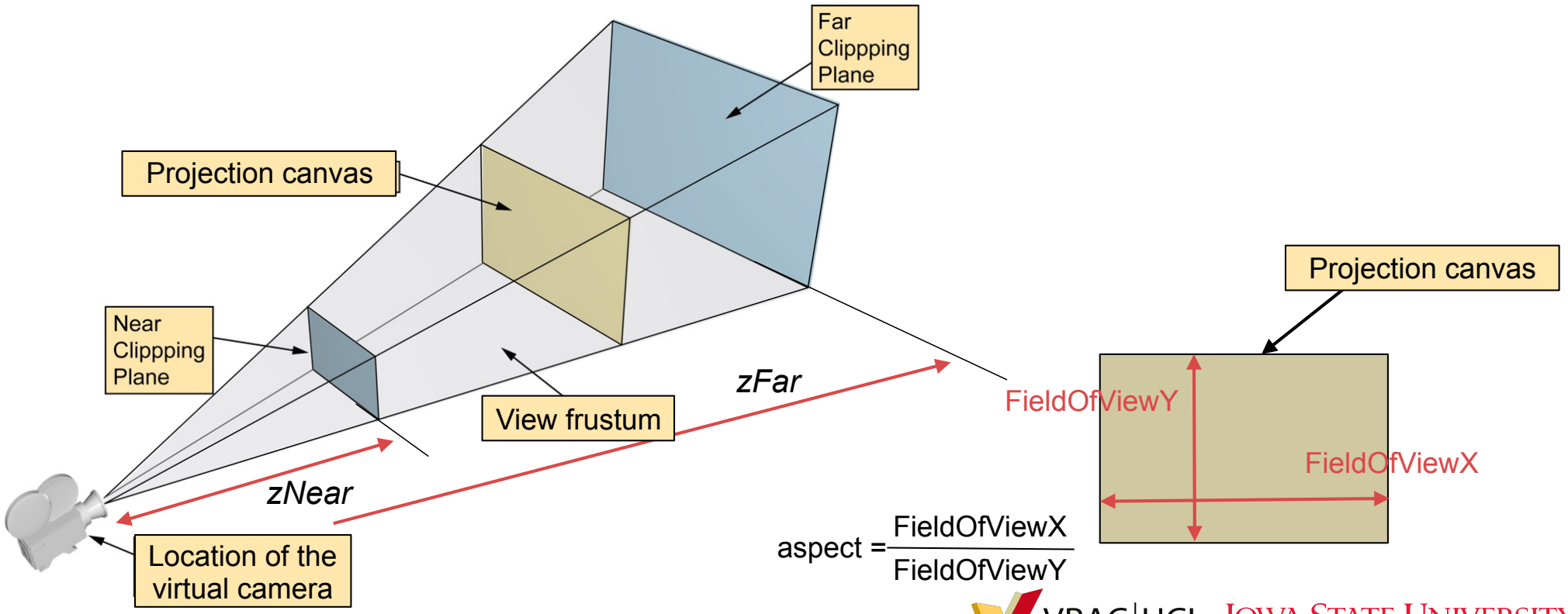
Set a Projection

Set a Projection

```
glm::mat4 perspective (T const &fovy, T const &aspect, T const &near, T const &far)
```

Parameters:

- fovy: Specifies the field of view angle, in degrees, in the y direction.
- aspect: Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
- zNear: Specifies the distance from the viewer to the near clipping plane (always positive).
- zFar: Specifies the distance from the viewer to the far clipping plane (always positive).

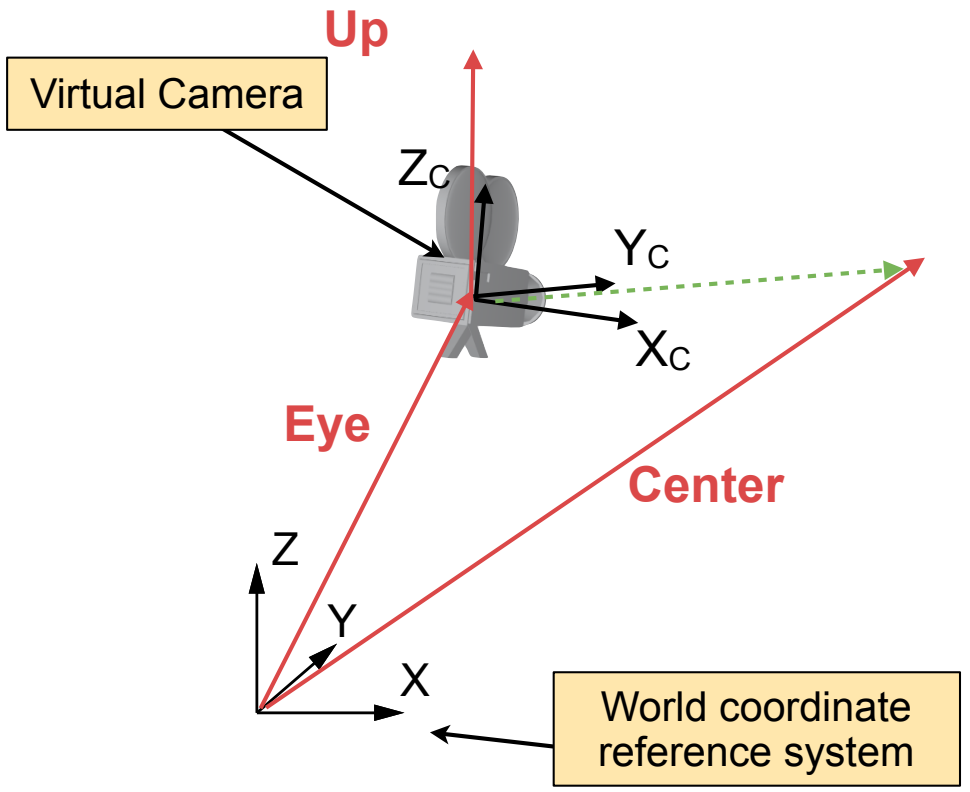


Camera Location and Orientation

```
glm::mat4  lookAt (
detail::tvec3< T > const &eye,
detail::tvec3< T > const &center,
detail::tvec3< T > const &up)
```

Parameters:

- eye: Specifies the position of the eye.
- center
  Specifies the position of the point to look to.
- up: Specifies the direction of the up vector.



Vectors to setup a virtual camera

Example

ARLAB

Global defined coordinates

```
glm::mat4 projectionMatrix; // Store the projection matrix
glm::mat4 viewMatrix; // Store the view matrix
glm::mat4 modelMatrix; // Store the model matrix
```

Init part of the program

Example and Structure

ARLAB

Global

```
glm::mat4 projectionMatrix; // Store the projectio
glm::mat4 viewMatrix; // Store the view matrix
glm::mat4 modelMatrix; // Store the model matrix
```

Global variables can be accessed at any location in your program. (qualifier **extern** is required in different files)

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

Program init

```
projectionMatrix = glm::perspective(1.57f, (float)800 / (float)600, 0.1f,
100.f);
modelMatrix = glm::translate(0.0f, 0.0f, 0.0f);

viewMatrix = glm::lookAt(  glm::vec3(0.0f, 0.0f, 1.5f),
                           glm::vec3(0.0f, 0.0f, 0.0f),
                           glm::vec3(0.0f, 1.0f, 0.0f));
```

main loop

```
}
```

Question

What do I have to do if I have two or more models and all models should be placed at a different location?

Solutions

- Multiple model variables
- Change the content before you change the content.

Example and Structure

ARLAB

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

Program init

```
while
```

Clear the window

main loop

```
// Enable the shader program
glUseProgram(program);

glBindVertexArray(vaoID[0]);
modelMatrix = glm::translate(0.0f
[ something more ]
glDrawArrays(GL_TRIANGLE_STRIP, 0, 16);

glBindVertexArray(vaoID[1]);
modelMatrix = glm::translate(5,9f
[ something more ]
glDrawArrays(GL_TRIANGLE, 0, 16);

glUseProgram(0);
```

Swap buffers

```
}
```

We change the values before we render the object.
To save memory, keep one variable if you can life with one.

[something more] :
copy the data to your GPU

Uniform Variables

ARLAB

```
GLint glGetUniformLocation( GLuint program, const GLchar *name);
```

Returns the location of a uniform variable

Parameters:

- program: Specifies the program object to be queried.
- name: string containing the name of the uniform variable

Example:

```
int projectionMatrixLocation = glGetUniformLocation(program, "projectionMatrix");
int viewMatrixLocation = glGetUniformLocation(program, "viewMatrix");
int modelMatrixLocation = glGetUniformLocation(program, "modelMatrix");
```

```
static const string vs_string =
    "#version 410 core
    "
    "uniform mat4 projectionMatrix;
    "uniform mat4 viewMatrix;
    "uniform mat4 modelMatrix;
    "in vec3 in_Position;
    "
    "in vec3 in_Color;
    "out vec3 pass_Color;
    "
```

The names **must** be equal

```
\n"
 \n"
\n"
 \n"
 \n"
 \n"
 \n"
 \n"
 \n"
\n"
```

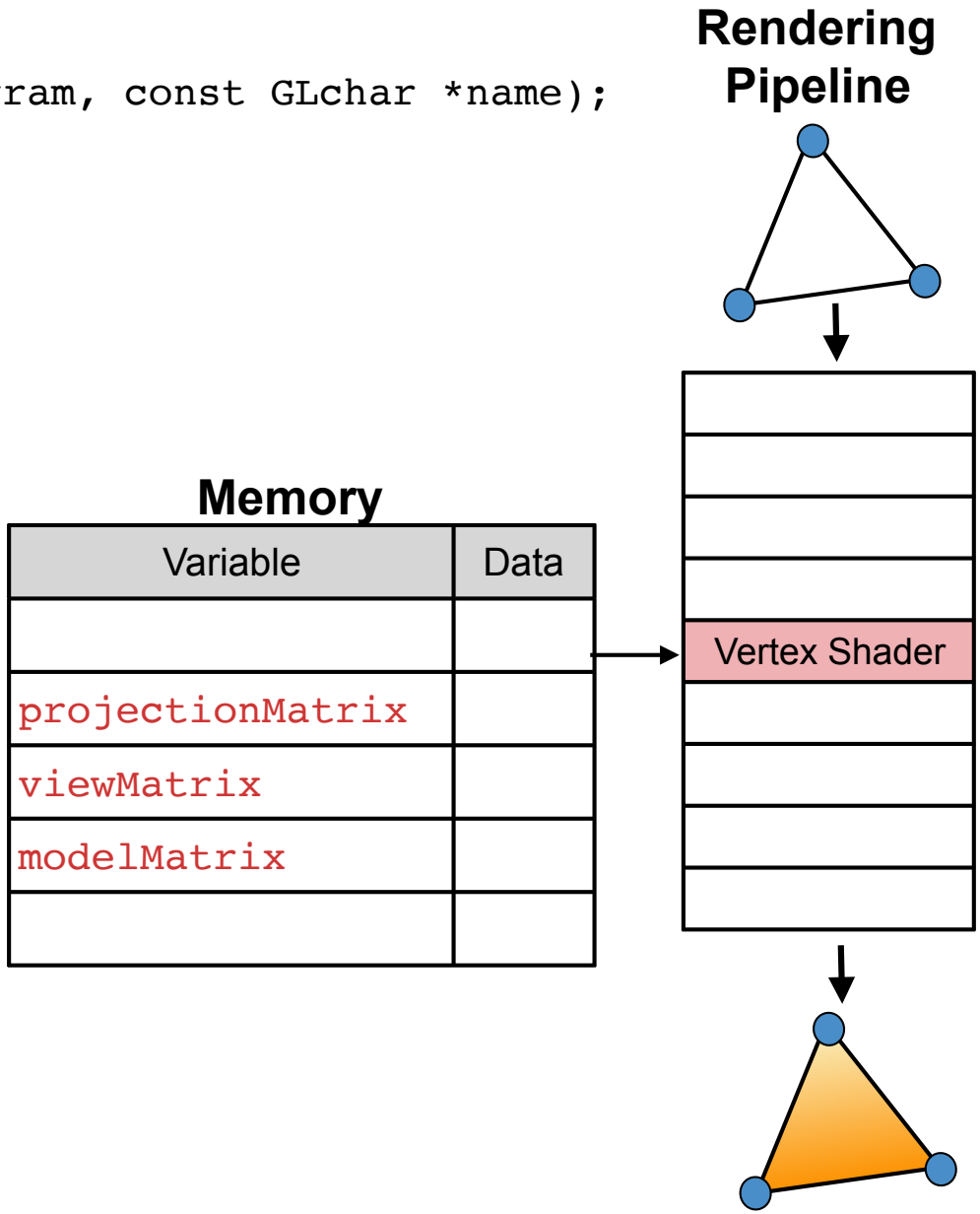
program: the shader program variable

Uniform Variables

```
GLint glGetUniformLocation( GLuint program, const GLchar *name);
```

Returns the location of a uniform variable

- This function only creates the memory
- The data field remains empty



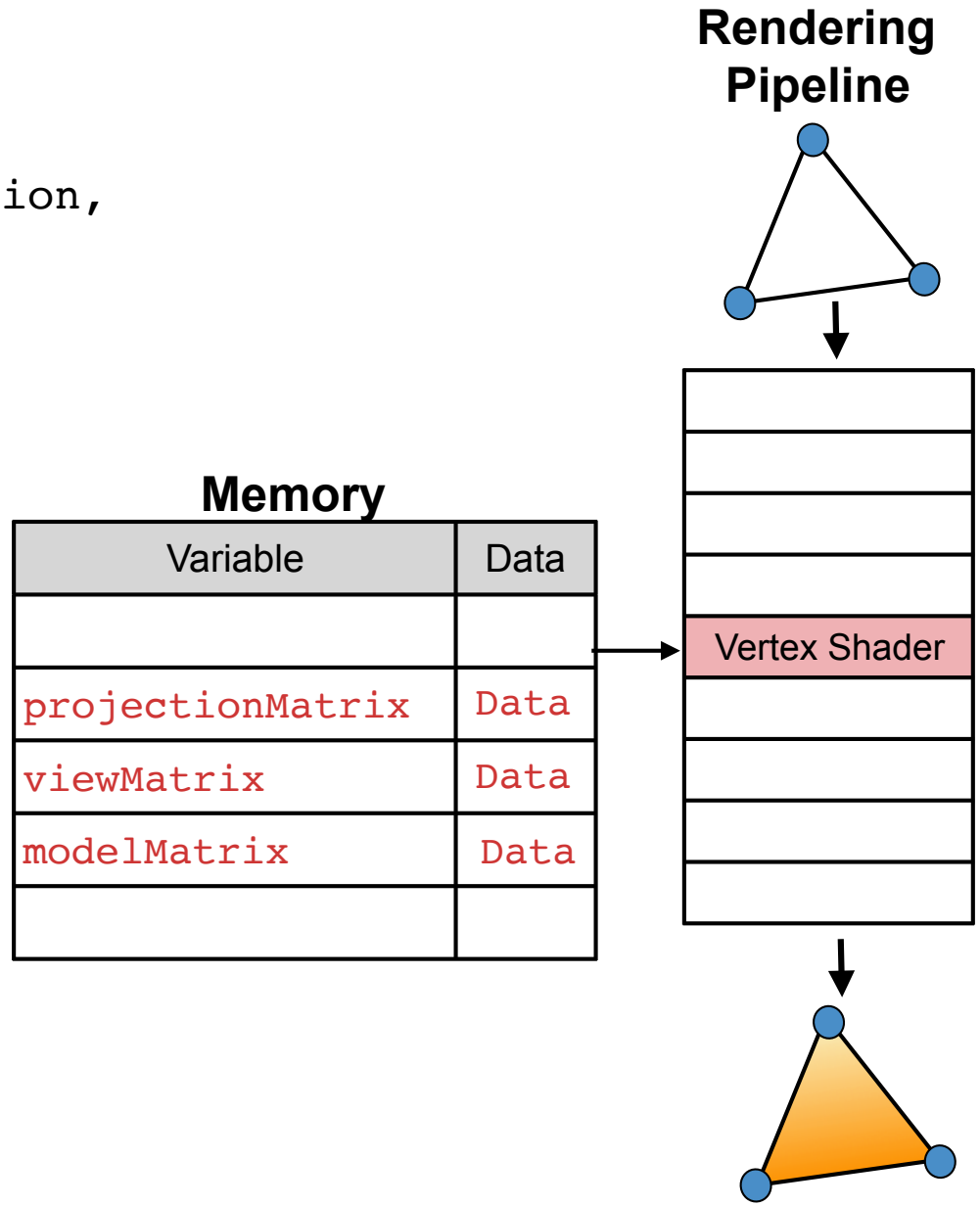
glUniformMatrix4fv

Copies the data into the graphics memory

```
void glUniformMatrix4fv( GLint location,
GLsizei count,
GLboolean transpose,
const GLfloat *value);
```

Parameters:

- location: Specifies the location of the uniform value to be modified.
- count: Specifies the number of matrices that are to be modified.
- transpose: Must be GL_FALSE.
- value: Specifies a pointer to an array of count values that will be used to update the specified uniform variable.



Example

```
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE,
                   &projectionMatrix[0][0]);
glUniformMatrix4fv(viewMatrixLocation, 1, GL_FALSE,
                   &viewMatrix[0][0]);
glUniformMatrix4fv(modelMatrixLocation, 1, GL_FALSE,
                   &modelMatrix[0][0]);
```

Example and Structure

ARLAB

```
int main(int argc, const char * argv[])
{
    glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE,
                       &projectionMatrix[0][0]);
    [..also the others.]
```

To initialize the memory. The value should not stay undefined.

Program init

```
    while
```

Clear the window

main loop

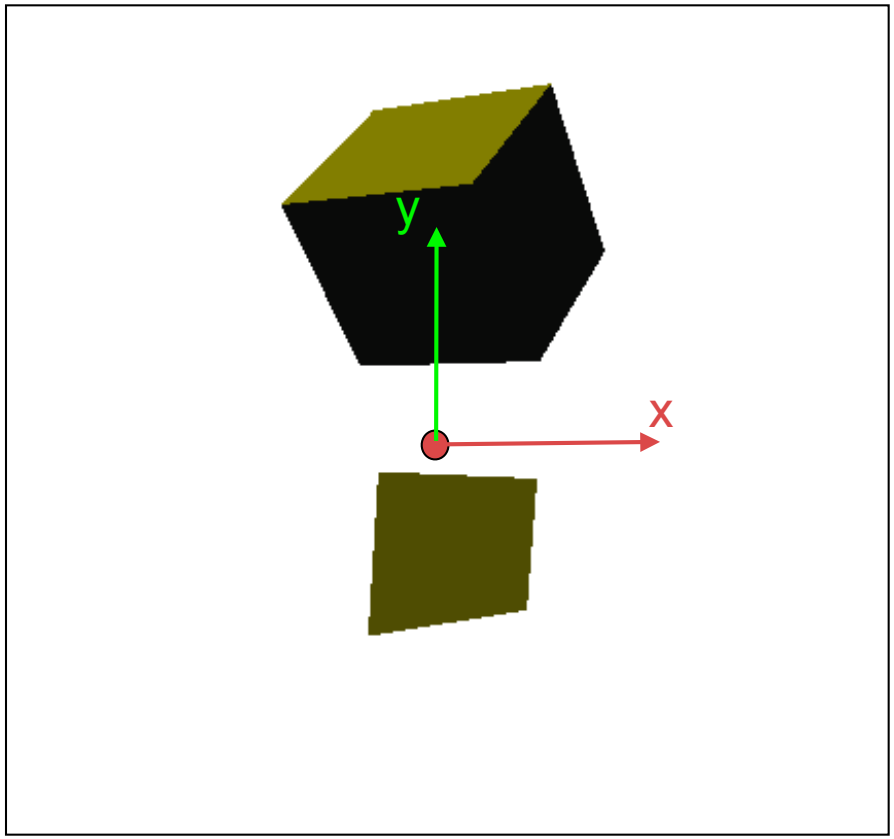
```
    // Enable the shader program
    glUseProgram(program);

        glBindVertexArray(vaoID[0]);
        modelMatrix = glm::translate(0.0f
        glUniformMatrix4fv(.....
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 16);

        glBindVertexArray(vaoID[1]);
        modelMatrix = glm::translate(5,9f
        glUniformMatrix4fv(.....
        glDrawArrays(GL_TRIANGLE, 0, 16);

    glUseProgram(0);
```

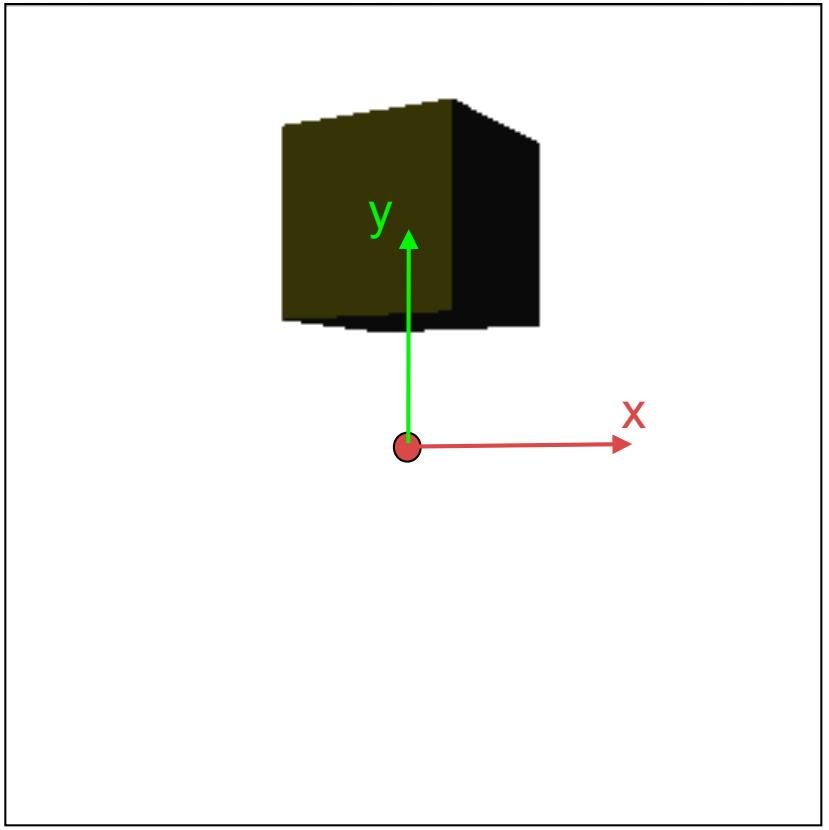
To copy data during runtime

Swap buffers

```
}
```

A lot of things to remember, but we will practice on Thursday





ARLAB

Questions ?

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY