Transformations and Viewing

What gives us Three.js?

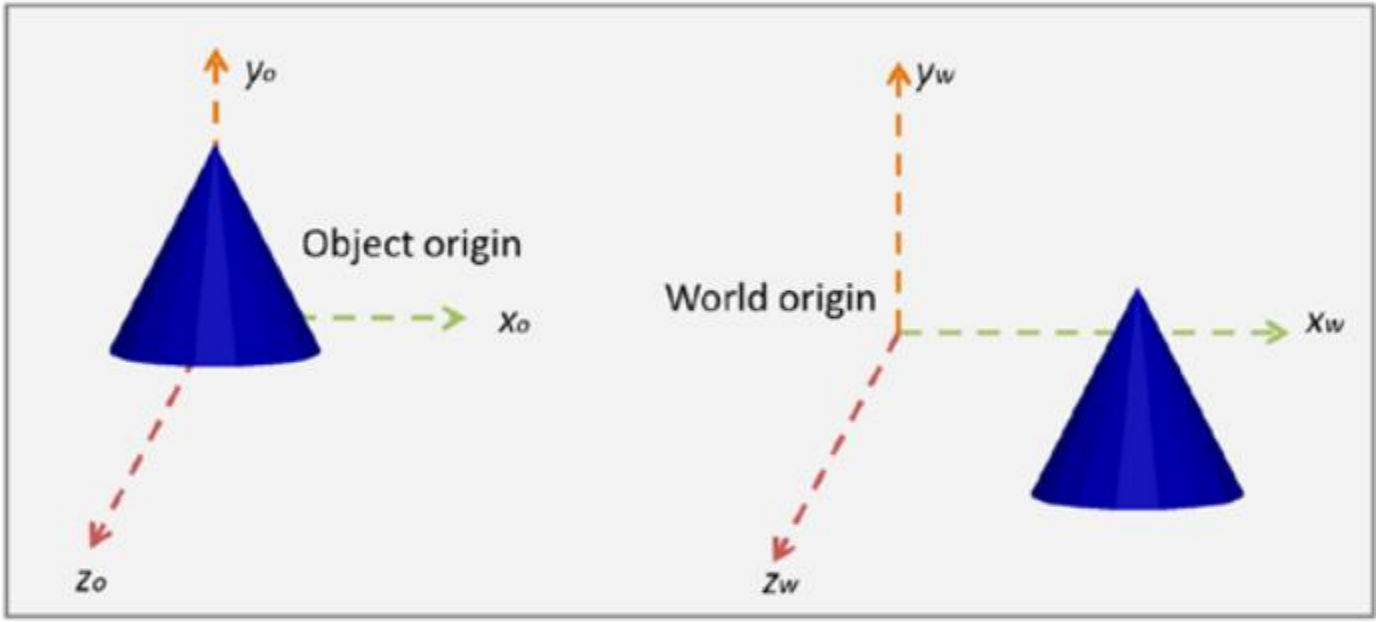
- … in transformations

- We meet examples at the very beginning
- Basic transformations on an object:
  - `position`
  - `rotation`
  - `scale`
  - `translateX()/translateY()/translateZ()`

properties

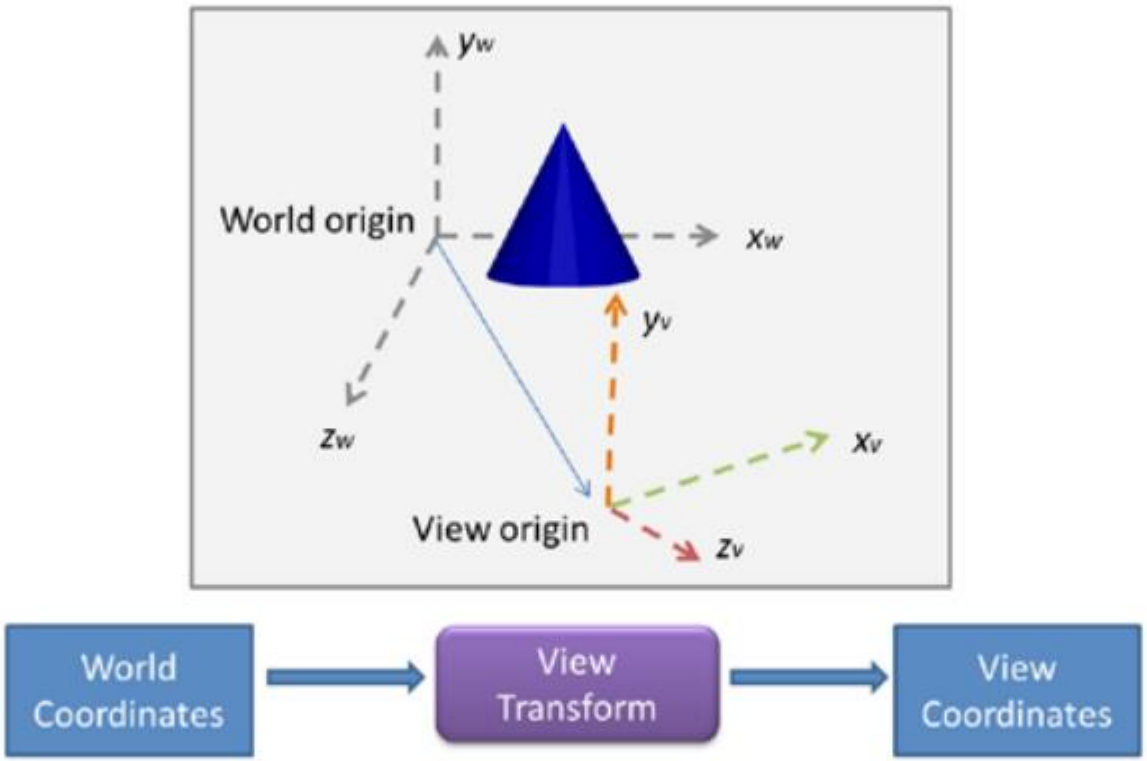
functions

Graphics Pipeline Elements



How do we do this?

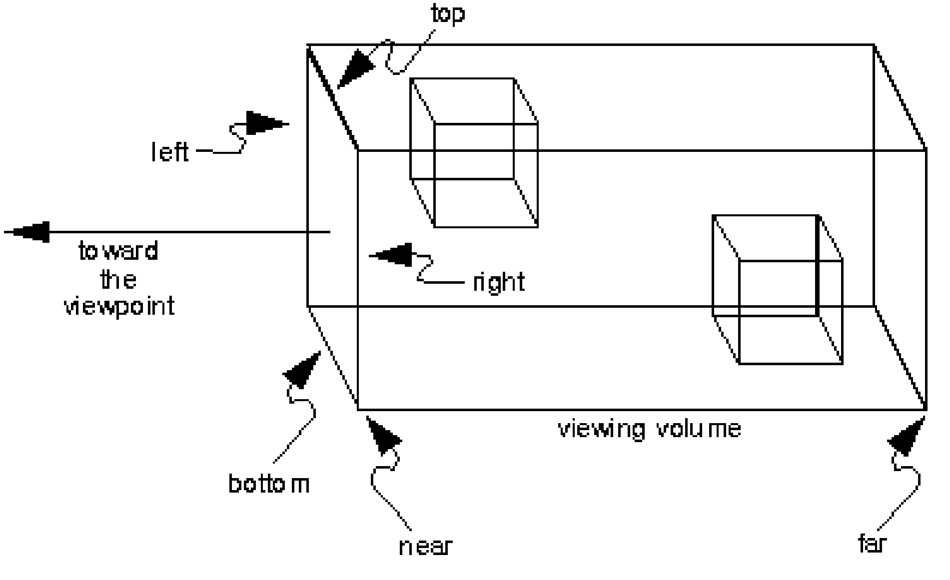
Model - View


y_w
World origin
x_w
y_v
z_w
x_v
View origin
z_v
World Coordinates
View Transform
View Coordinates

Ortographic Projection

OrthographicCamera(*left*, *right*, *bottom*, *top*, *near*, *far*)

```
var camera = new
THREE.OrthographicCamera(left, right,
bottom, top, near,  far);
scene.add( camera );
```

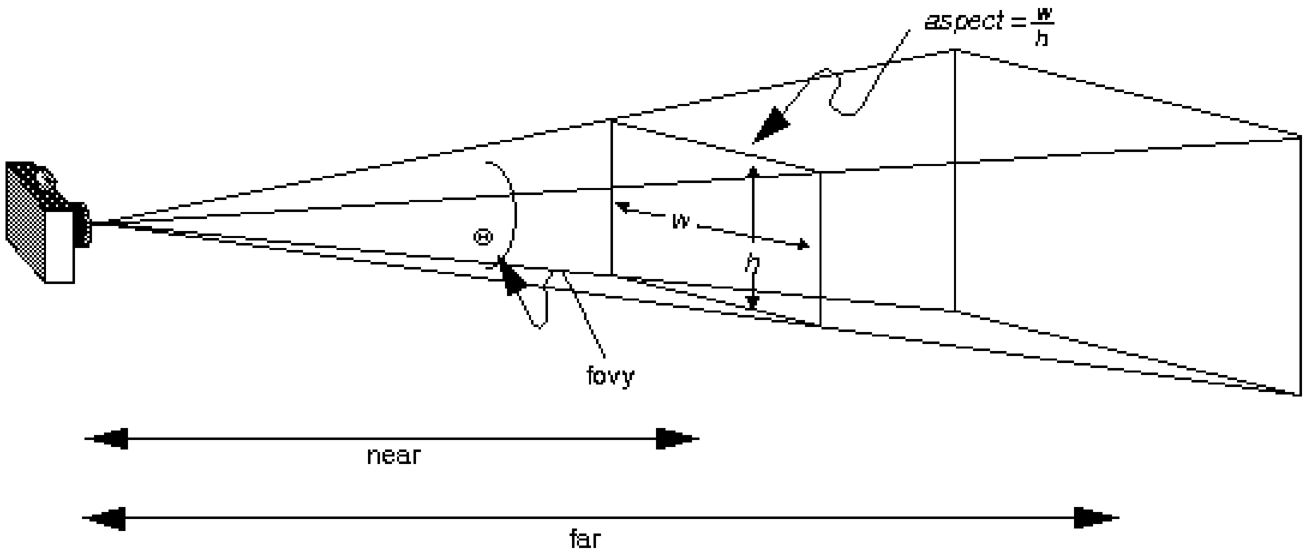


How do observer look?

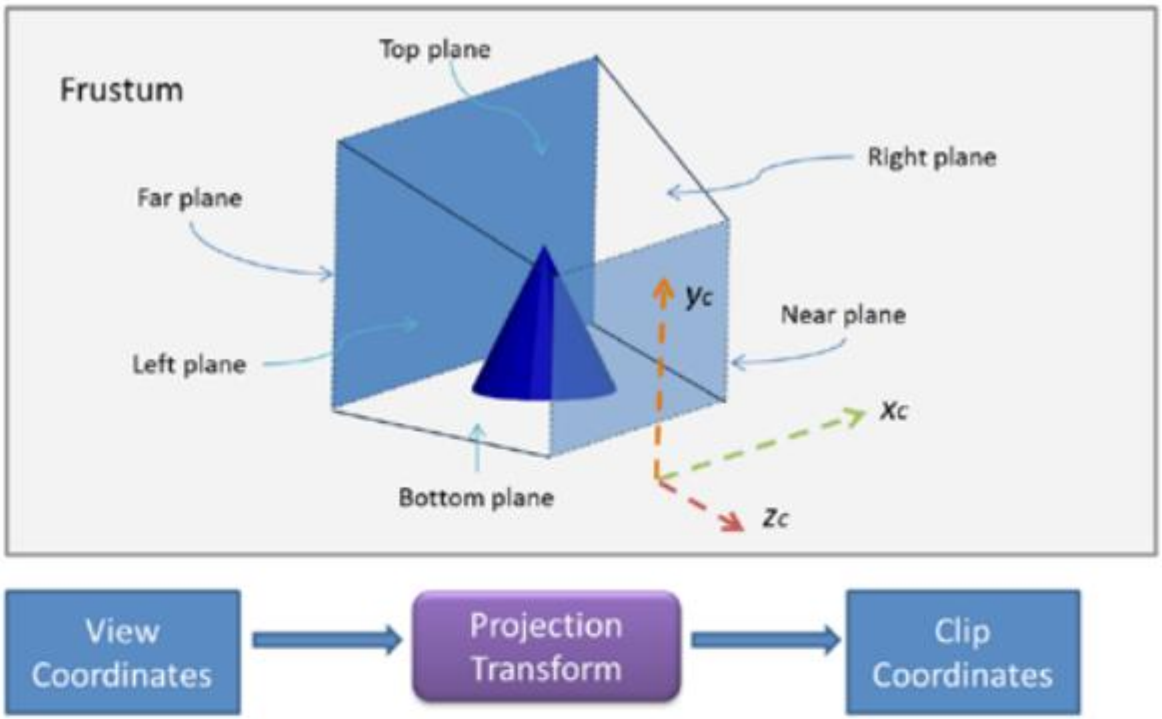
Perspective Projection

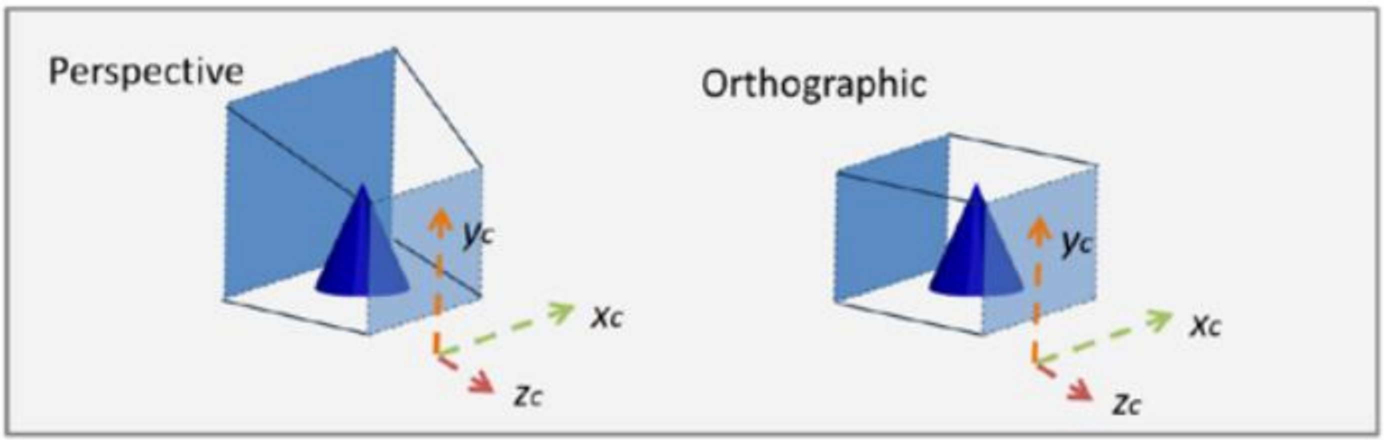
PerspectiveCamera(*fovy*, *aspect*, *near*, *far*)

```
var camera = new
THREE.PerspectiveCamera( 45, width /
height, 1, 1000 );
scene.add( camera );
```



View space – Clip space




Perspective
Orthographic
y_c
x_c
z_c
y_c
x_c
z_c

What next?

- *2D transformations: scale, shear, rotation*
- Composing transformations
- 3D rotations

(Nonuniform) Scale

$$x' = s_x \cdot x$$
$$y' = s_y \cdot y$$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \qquad S^{-1} = \begin{bmatrix} s_x^{-1} & 0 \\ 0 & s_y^{-1} \end{bmatrix}$$

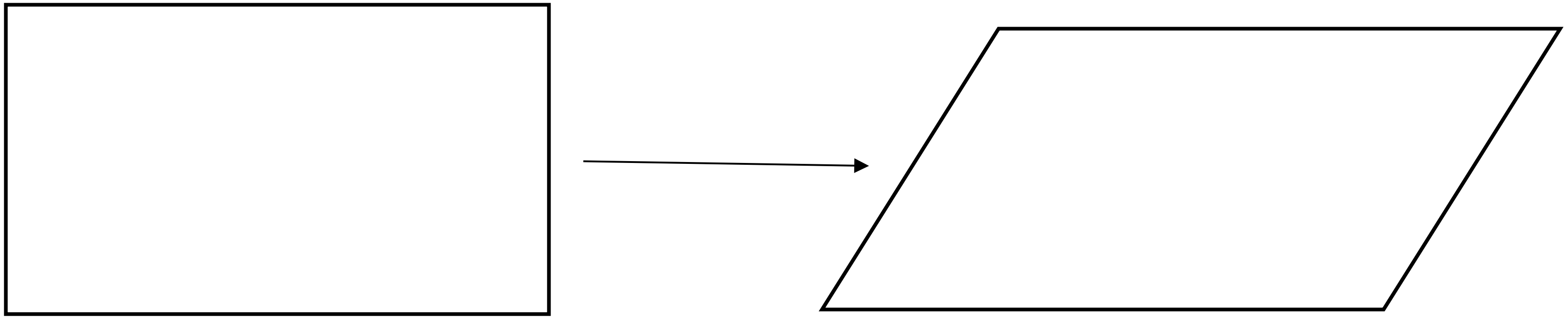
$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \qquad S^{-1} = \begin{bmatrix} s_x^{-1} & 0 & 0 \\ 0 & s_y^{-1} & 0 \\ 0 & 0 & s_z^{-1} \end{bmatrix}$$

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

Shear

$$SH = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \qquad SH^{-1} = \begin{bmatrix} 1 & -a \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$



Rotations

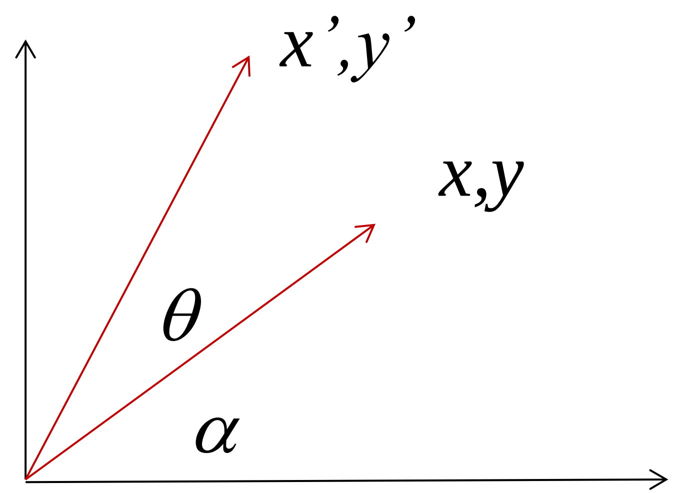
2D rotations are simple

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- Can we derive it?

$$x = \vec{r}\cos\alpha$$

$$x' = \vec{r}\cos(\alpha+\theta)\ldots$$



Translation

$$x' = x + T_x$$

$$y' = y + T_y$$

$$z' = z + T_z$$

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

- What to do with free terms?

More general transformation

- In general we have:

$$x' = a_x x + b_x y + c_x z + d_x$$
$$y' = a_y x + b_y y + c_y z + d_y$$
$$z' = a_z x + b_z y + c_z z + d_z$$

or

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

- We want to get rid off the free terms.

$$\vec{v}' = \mathbf{M}\vec{v}$$

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

(Modelview Matrix)

Rotations, contd.

Basic properties…

- Linear $\quad R(X+Y)=R(X)+R(Y)$
- Commutative

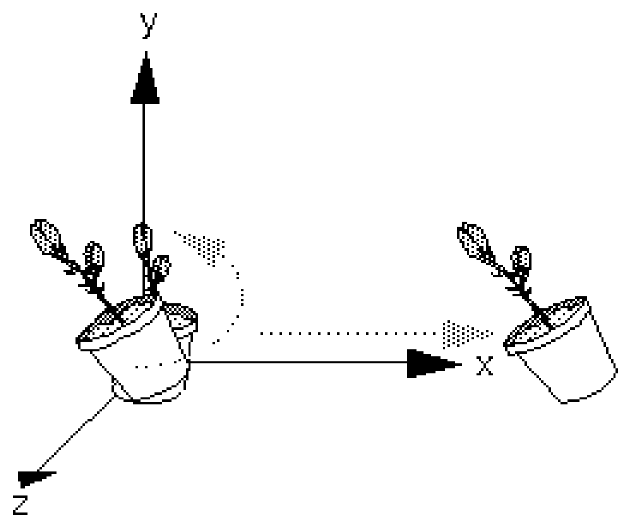
Outline

- 2D transformations: rotation, scale, shear
- *Composing transforms*
- 3D rotations
- Translation: Homogeneous Coordinates
- Transforming Normals

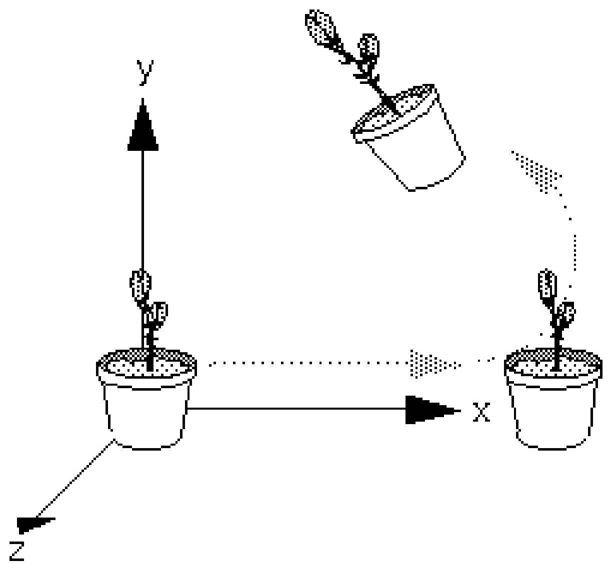
Composing Transforms

- Usually we build a complex transformation by composing simple ones (for which we have matrix patterns)
- E.g. first scale then rotate
- We see now advantage of matrix formulation:
  - Transformation is performed for many points
  - Complex transformation is in advanced put into a single matrix
- In general not commutative!! Order matters

Combining Translations, Rotations



First Rotation, then Translation



First Translation, then Rotation

E.g. Composing rotations, scales

- In terms of functions:

$$p' = S(p) \qquad\qquad p'' = R(p')$$

$$p''' = R(S(p)) = RS(p)$$

- In terms of matrices: RS is also a matrix
- $RS \neq SR$, not commutative

Composing example

Inverting Composite Transforms

- Say, we want to invert a combination of 3 transforms
- Option 1: Find composite matrix, invert
- Option 2: Invert each transform ***and swap order***
- Obvious from properties of matrices

$$M = M_3 M_2 M_1$$

$$M^{-1} = M_1^{-1} M_2^{-1} M_3^{-1}$$

$$M^{-1} M = M_1^{-1} \left(M_2^{-1} \left(M_3^{-1} M_3 \right) M_2 \right) M_1$$

Orthogonal matrix

Matrix A is orthogonal if and only if

$$A^T A = I$$

E.g. Rotation matrix is orthogonal. Check it.

$$R^T R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} =$$

$$\begin{bmatrix} \cos^2\theta + \sin^2\theta & -\cos\theta\sin\theta + \sin\theta\cos\theta \\ -\sin\theta\cos\theta + \cos\theta\sin\theta & \sin^2\theta + \cos^2\theta \end{bmatrix}$$

Orthogonal matrix

Important and useful property:

$$A^{-1} = A^T$$

It is much faster to transpose a matrix, than to invert it

Outline

- 2D transformations: rotation, scale, shear
- Composing transforms
- *3D rotations*
- Translation: Homogeneous Coordinates
- Transforming Normals

Rotations in 3D

- Rotations about coordinate axes is a simple extension of 2D rotations

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Again orthogonal matrices

Interpretation 3D Rotations Matrix

- Rows of matrix are the vectors of new coordinate system (the rotated one)
- Old coordinate frame: x, y, z versors
- After transformation:

$$R_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

- we have new uvw coordinate frame

Geometric Interpretation 3D Rotations

- Rows of matrix are 3 unit vectors of new coord frame
- Can construct rotation matrix from 3 orthonormal vectors
- Effectively, projections of point into new coord frame
- New coord frame uvw taken to cartesian components xyz
- Inverse or transpose takes xyz cartesian to uvw

Non-Commutativity

- Not Commutative (unlike in 2D)!!
- Rotate by x, then y is not same as y then x
- Order of applying rotations does matter
- Follows from matrix multiplication not commutative
  - R1 * R2 is not the same as R2 * R1

Translation

- Translation is probably the simplest transformation (is it?), but there's something tricky about it

- E.g. move x by +7 units, leave y, z unchanged:

$$x' = x + 7$$

$$y' = y$$

$$z' = z$$

- How to express it in a matrix form? I.e. $p' = Mp$

?

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} & & \\ & ? & \\ & & \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x+7 \\ y \\ z \end{bmatrix}$$

Homogeneous Coordinates

- We need to add a fourth coordinate (w=1). It's quite common way in linear algebra, to get rid off free terms.
- 4x4 matrices in 3D space - very common in computer graphics

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- w may look like a dummy variable – but may play it's role

Representation of Points (4-Vectors)

Homogeneous coordinates

$$P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$

- Divide by 4th coord (w) to get (inhomogeneous) point
- Multiplication by w > 0, no effect
- Assume w ≥ 0. For w > 0, normal finite point. For w = 0, point at infinity (used for vectors to stop translation)

What are the Advantages of Homogeneous Coords?

- Unified framework for translation and other transformations
- Only 4x4 matrix – easy to concatenate
- Less problems with perspective viewing
- No special cases – homogenous formulas

General Translation Matrix

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}p = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ w \end{bmatrix}$$

3 basic Rotation Matrices

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Combining Translations, Rotations

- Order matters!! TR is not the same as RT (demo)
- General form for rigid body transforms
- We show rotation first, then translation (commonly used to position objects) on next slide. Slide after that works it out the other way

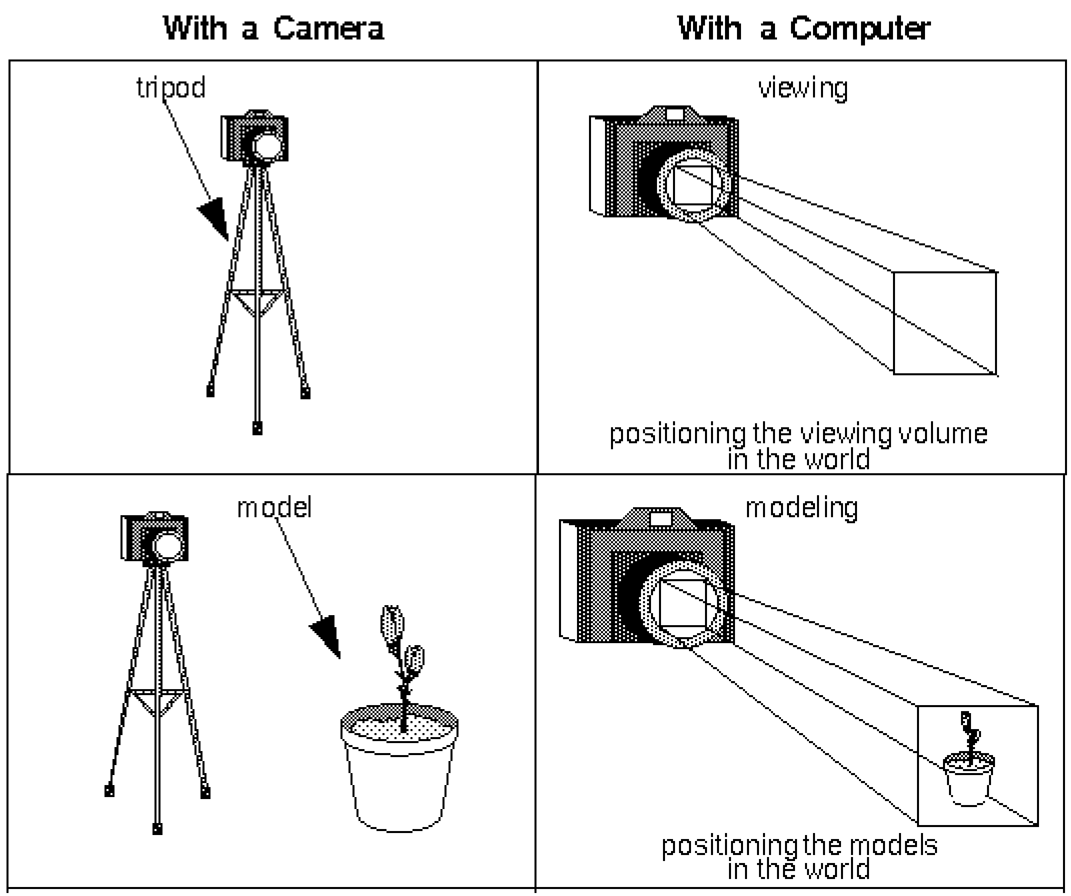
Geometry Transformation

Transformations are composed of four elements

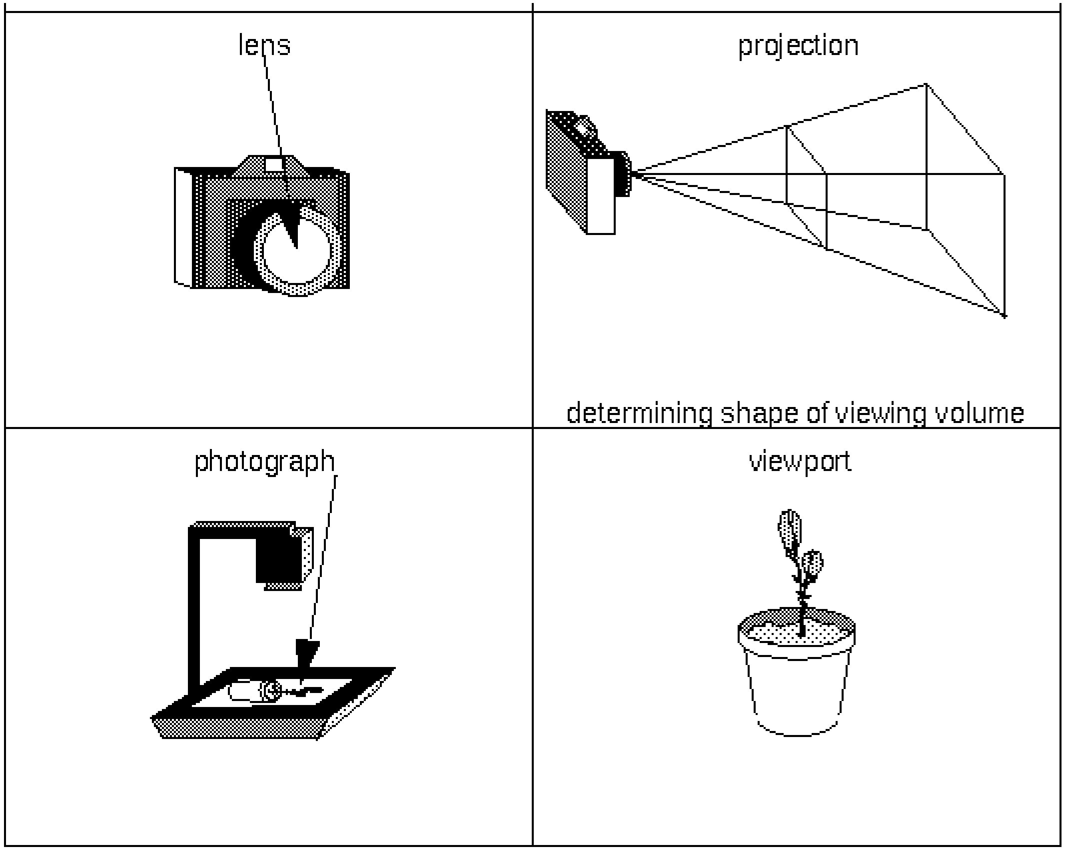
- **Viewing transformation –** positioning the camera
- **Modeling transformation -** moving objects

  (Viewing and **Modeling** are complementary**)**
- **Projection –** on the plane
- **Viewport –** mapping on the window

Camera analogy



Camera analogy contd.



Two ways of making transformation in OpenGL/WebGL

- First approach:
  we operate directly on matrices
- Second approach:
  we hide matrix transformations inside functions

- Which one is better?

Two ways of making transformation in OpenGL

- Functions typical to ”old” OpenGL.
- Cancelled in OpenGL 3.0
- Transformations and Projections moved to shader programming.
- Not everyone likes low-level shader programming.
- Search for new functions…

GLM – OpenGL Mathematics

- Can be easily found on glm.g-truc.net/ , along with other information
- Can be treated as enhancement of GLSL.
- Is included into unofficial OpenGL SDK

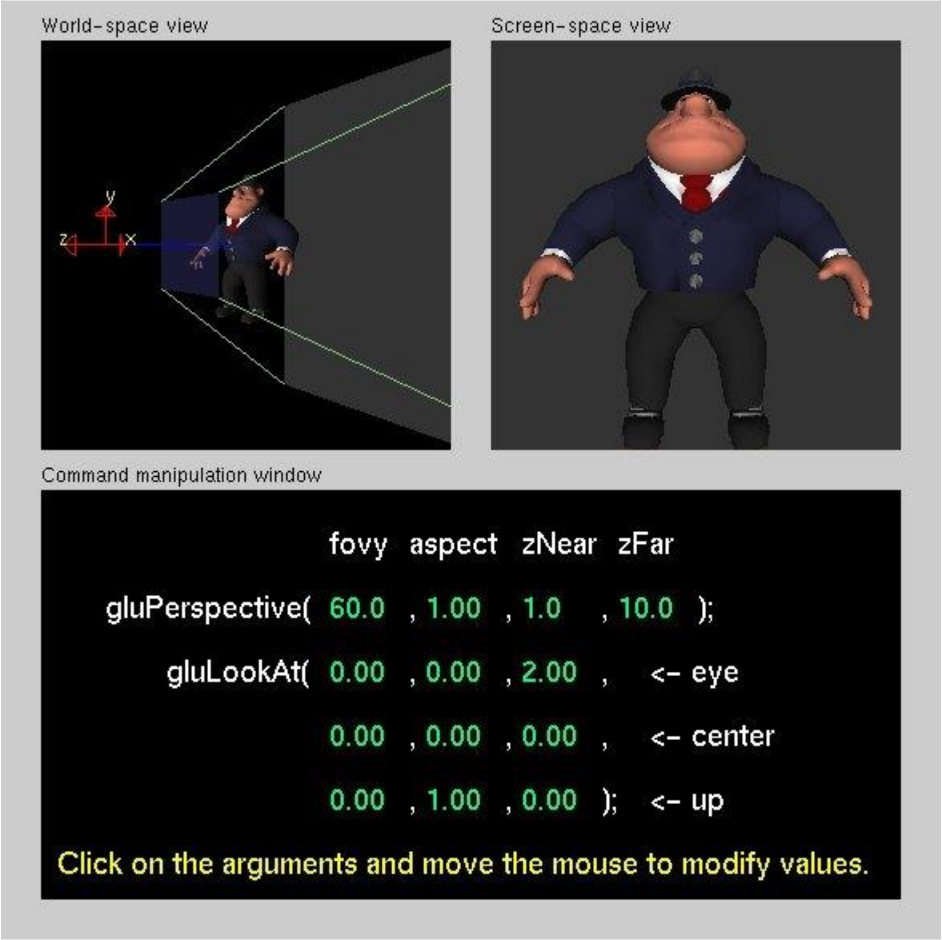
Viewing

Motivation

- We have seen transforms (between coordinate systems)
- But all that is in 3D
- We still need to make a 2D picture
- Project 3D to 2D. How do we do this?
- This lecture is about viewing transformations

Demo (Projection Tutorial)

- Nate Robbins OpenGL tutors
- http://user.xmission.com/~nate/
- Projection tutorial
- Old OpenGL Style
- [Demo]()



What we've seen so far

- Transforms (translation, rotation, scale) as 4x4 homogeneous matrices
- Last row always 0 0 0 1. Last w component always 1
- For viewing (perspective), we will use that last row and w component no longer 1 (must divide by it)

Outline

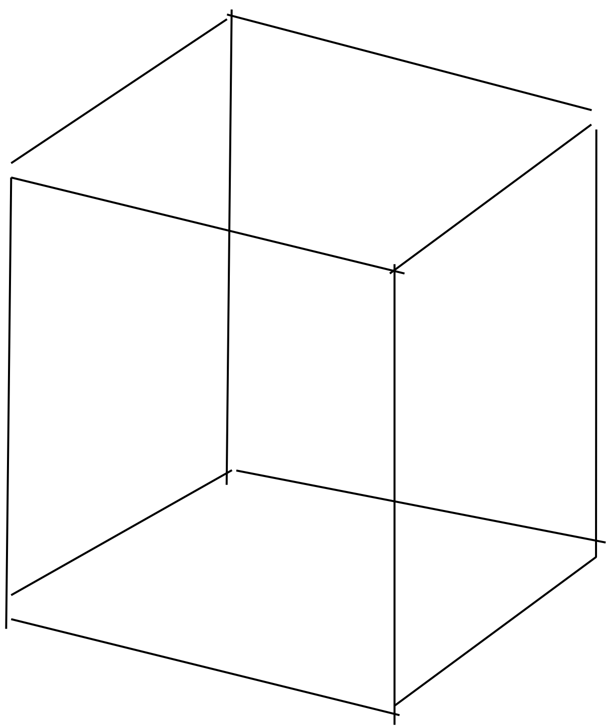
- *Orthographic projection (simpler)*
- Perspective projection, basic idea
- Derivation of gluPerspective (handout: glFrustum)
- Brief discussion of nonlinear mapping in *z*

Projections

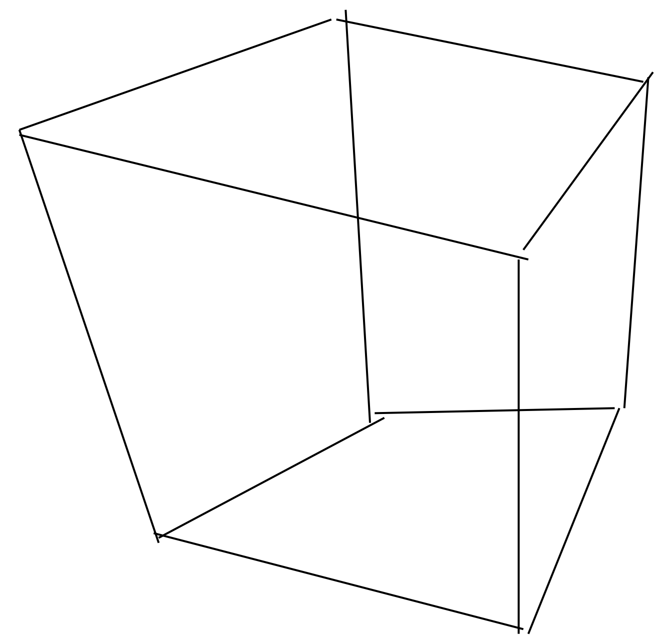
- To lower dimensional space (here 3D -> 2D)
- Preserve straight lines
- Trivial example: Drop one coordinate (Orthographic)

Orthographic Projection

- Characteristic: Parallel lines remain parallel
- Useful for technical drawings etc.



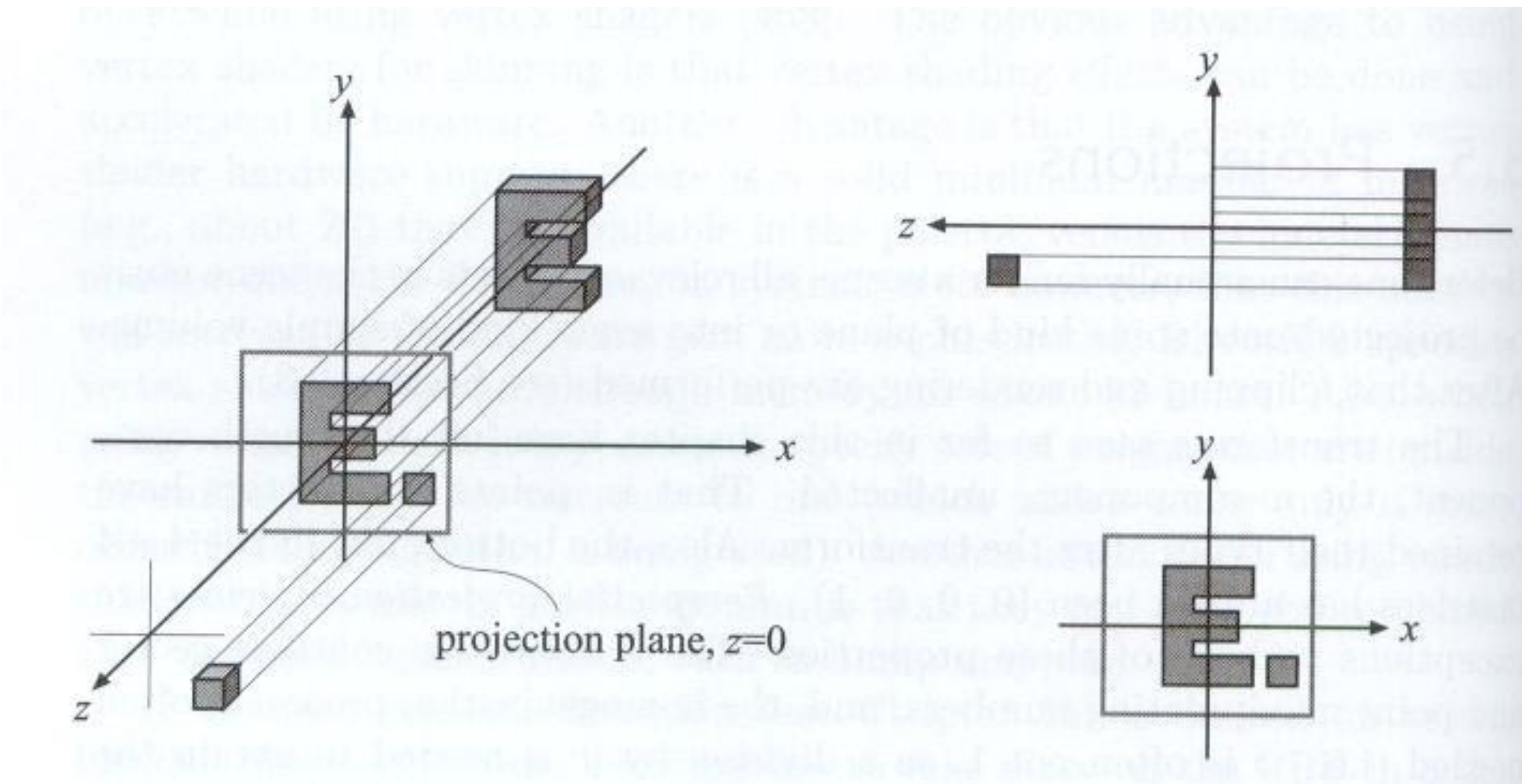
Orthographic



Perspective

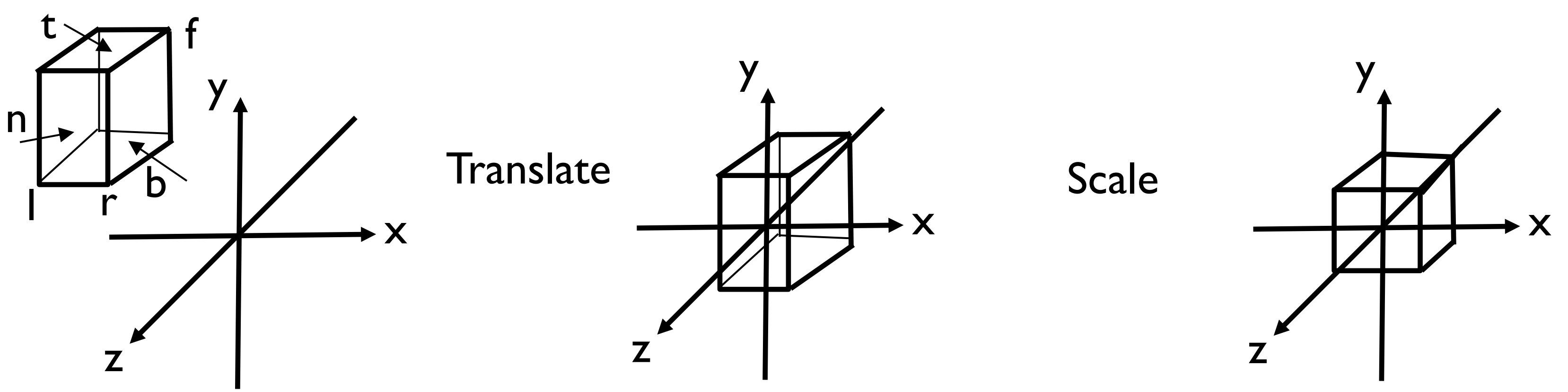
Example

- Simply project onto xy plane, drop z coordinate



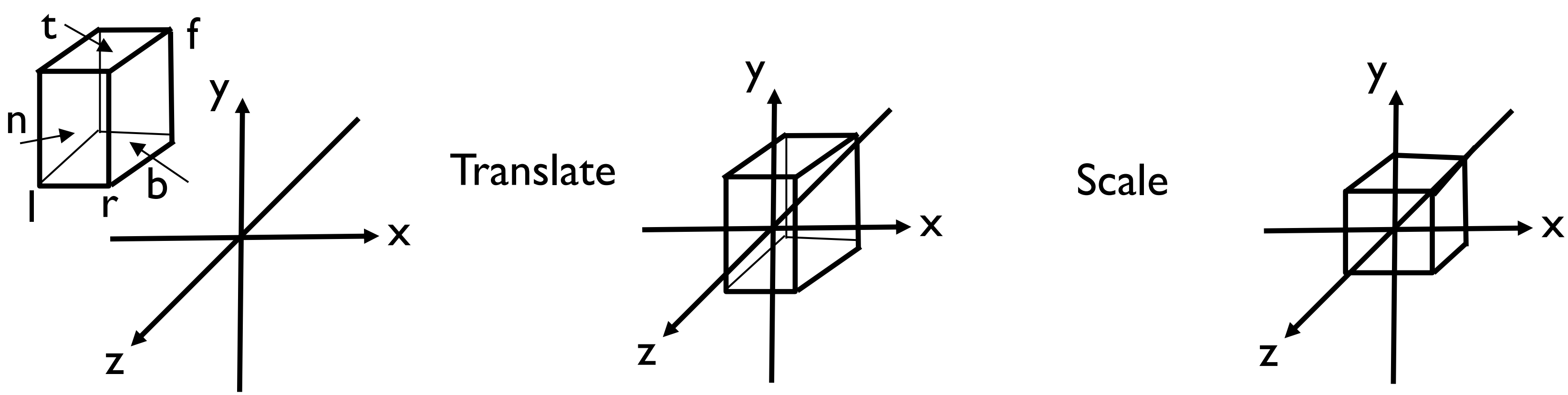
In general

- We have a cuboid that we want to map to the normalized or square cube from [-1, +1] in all axes
- We have parameters of cuboid (l,r ; t,b; n,f)



Orthographic Matrix

- First center cuboid by translating
- Then scale into unit cube



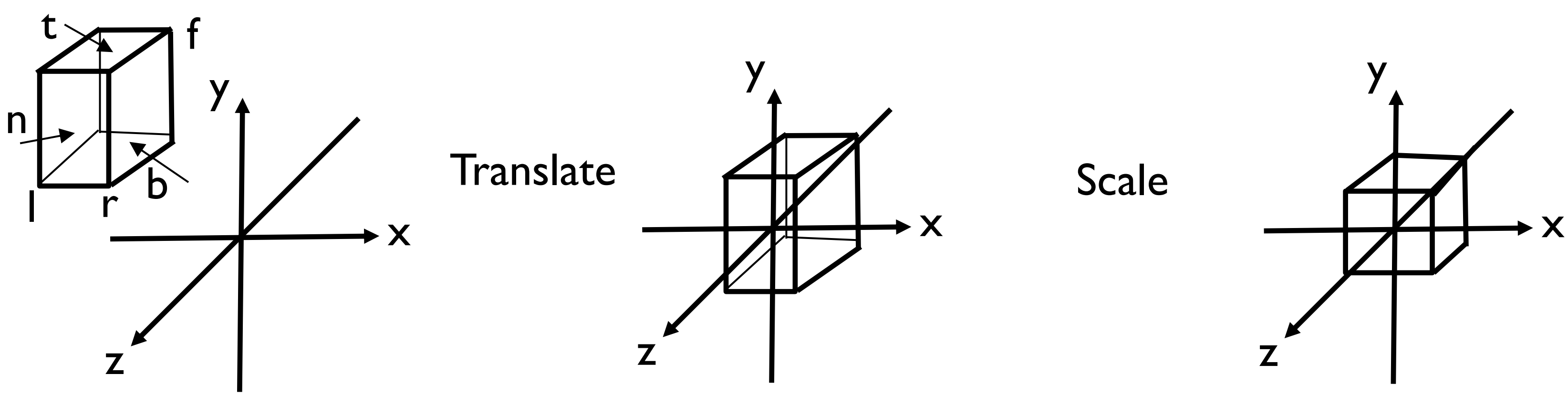
Transformation Matrix

Scale Translation

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

Attention!

- Looking down –z, f and n are negative (n > f)
- OpenGL convention: positive n, f, negate internally



Final Result

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

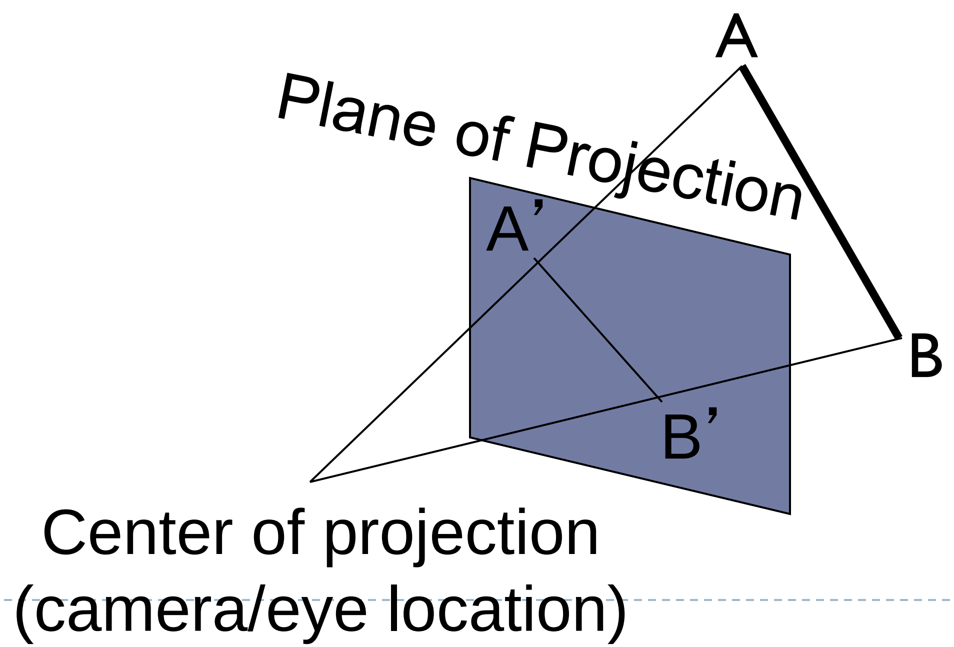
$$Ortho = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \boxed{\frac{-2}{n-f}} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Outline

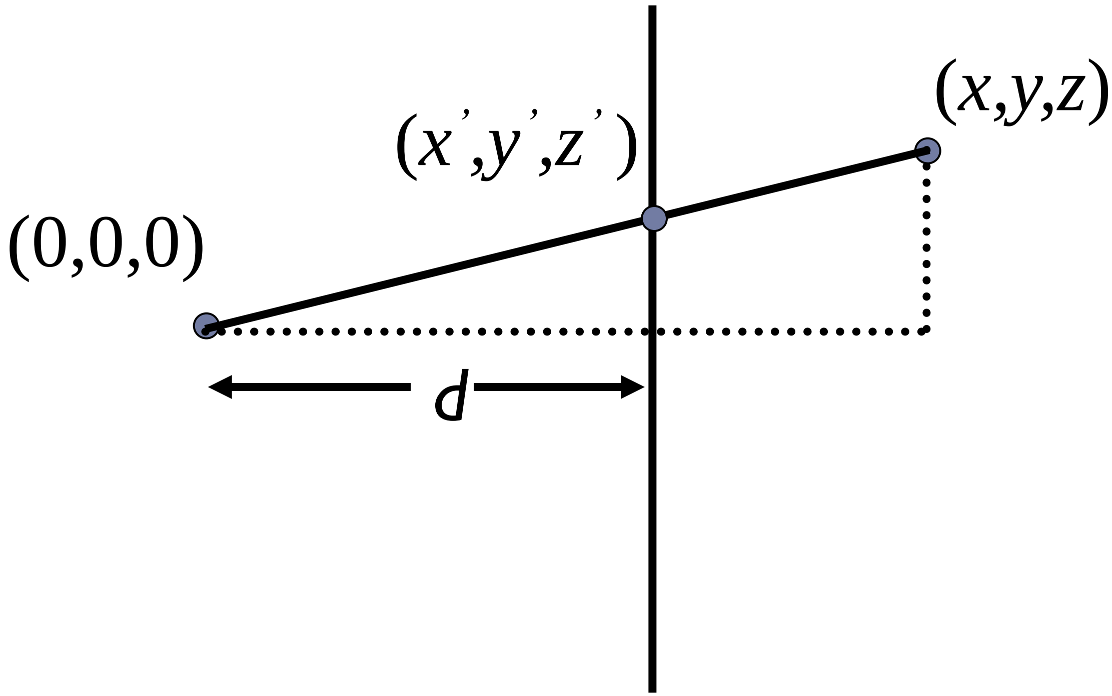
- Orthographic projection (simpler)
- *Perspective projection, basic idea*
- Derivation of gluPerspective (handout: glFrustum)
- Brief discussion of nonlinear mapping in z

Perspective Projection

- Most common computer graphics, art, visual system
- Further objects are smaller (size, inverse distance)
- Parallel lines not parallel; converge to single point



Overhead View of Our Screen



Looks like we've got some nice similar triangles here?

$$\frac{x}{z} = \frac{x'}{d} \qquad x' = \frac{dx}{z}$$

In Matrices

- Note negation of z coord (focal plane –d)
- (Only) last row affected (no longer 0 0 0 1)
- w coord will no longer = 1. Must divide at end

$$Ortho = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix}$$

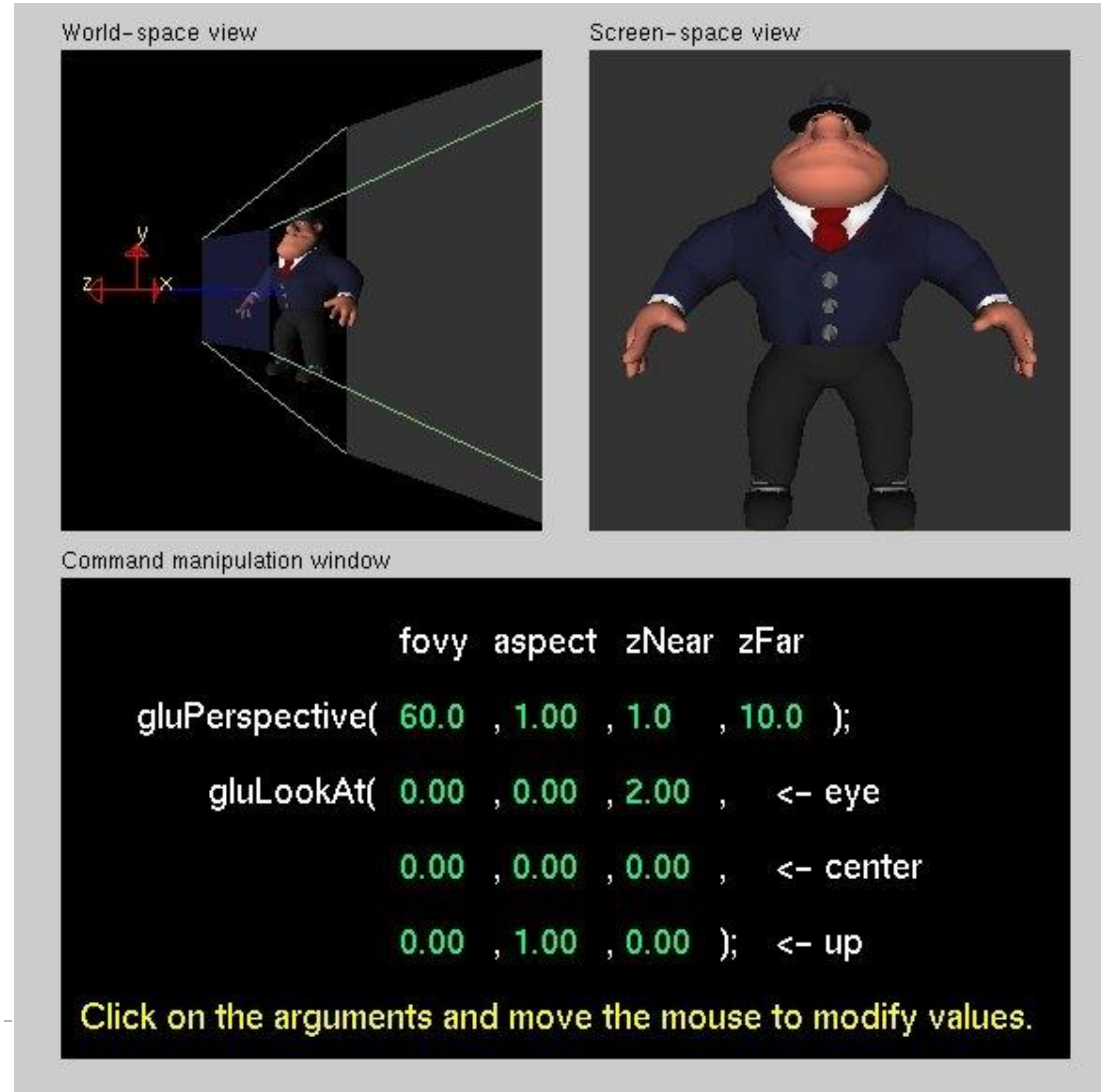
Verify

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\dfrac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -\dfrac{z}{d} \end{bmatrix}$$

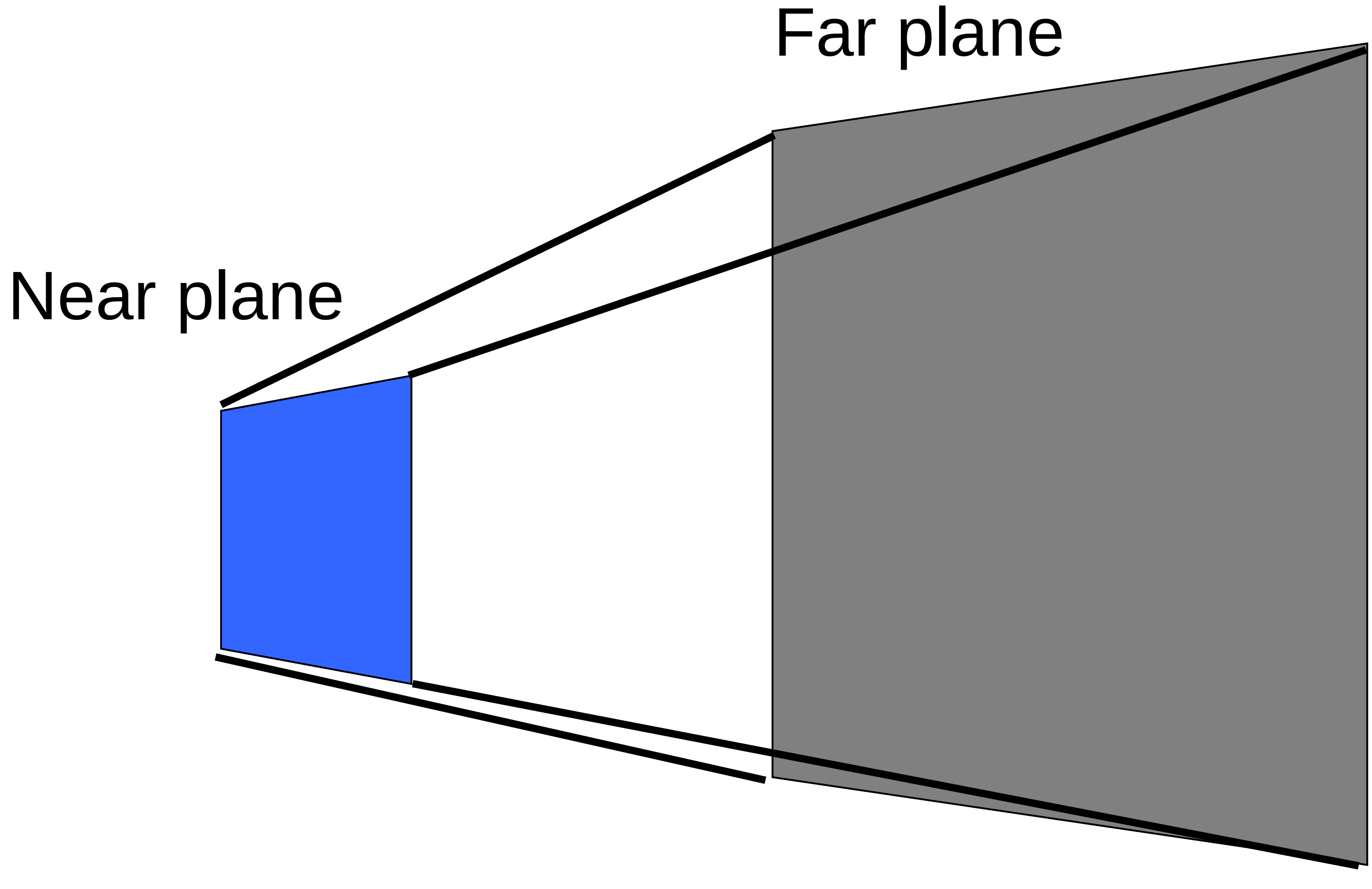
Outline

- Orthographic projection (simpler)
- Perspective projection, basic idea
- *Derivation of gluPerspective (handout: glFrustum)*
- Brief discussion of nonlinear mapping in z

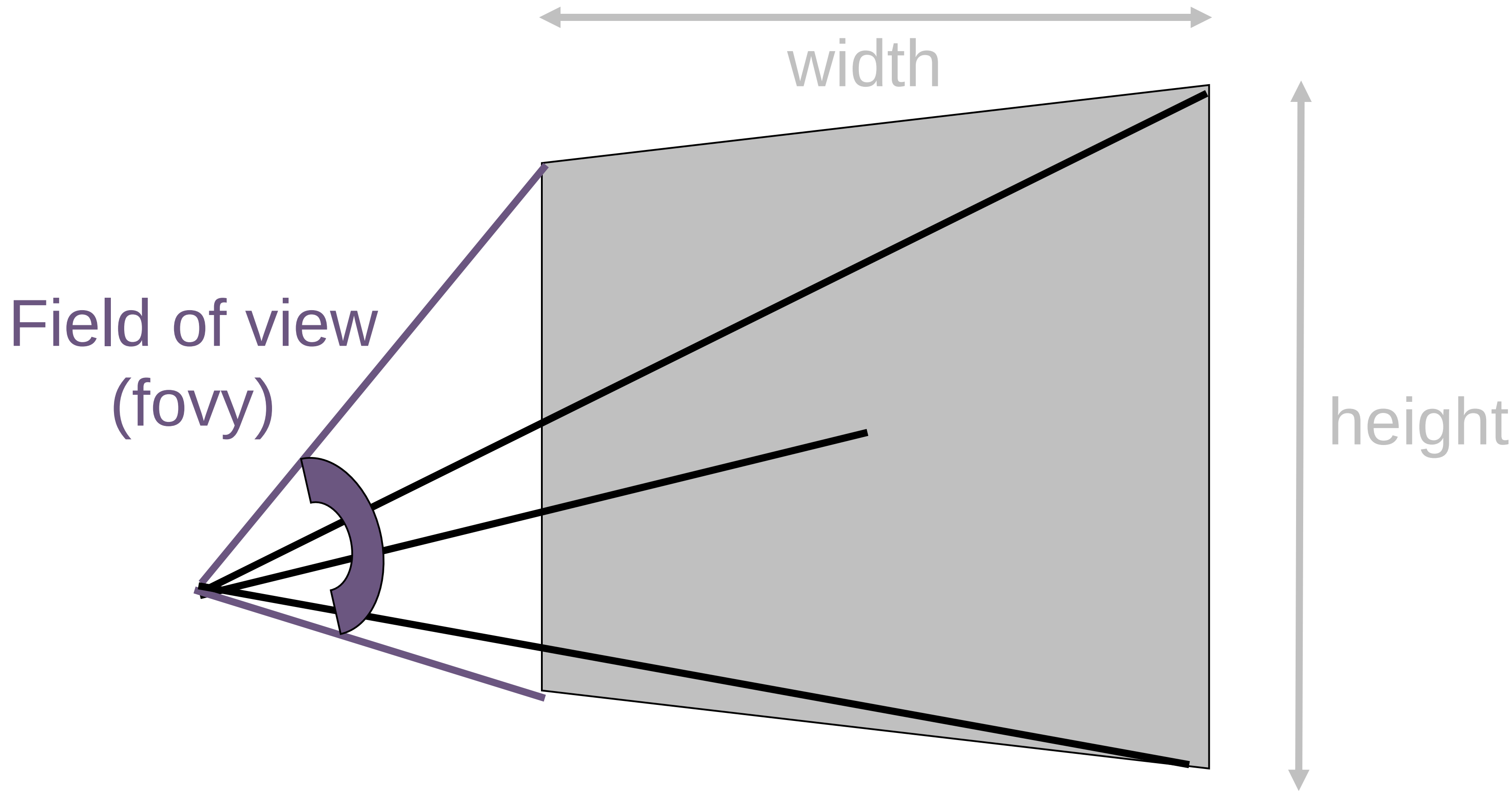
Remember projection tutorial



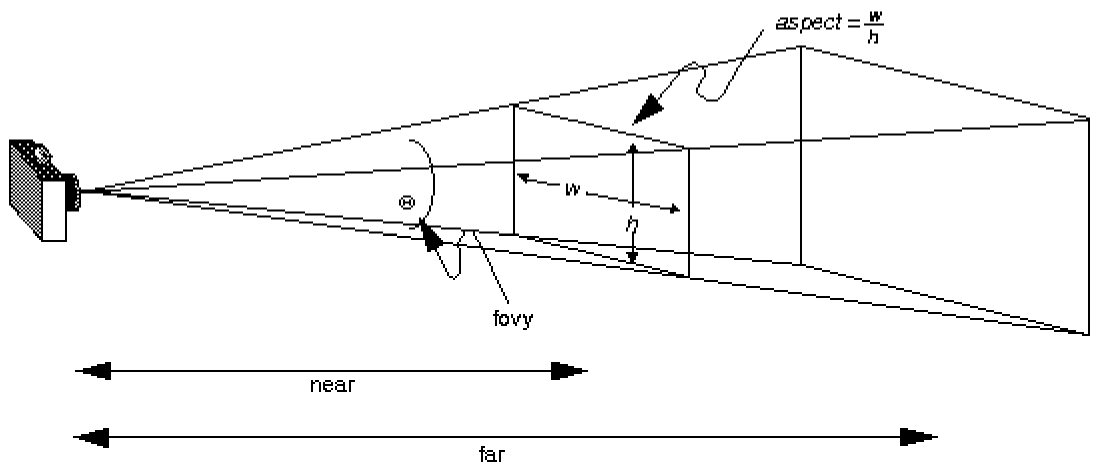
Viewing Frustum



Screen (Projection Plane)



Aspect ratio = width / height



$aspect = \frac{w}{h}$
Θ
w
h
fovy
near
far

gluPerspective

- gluPerspective(fovy, aspect, zNear > 0, zFar > 0)
- Fovy, aspect control fov in x, y directions
- zNear, zFar control viewing frustum

In Matrices

- Simplest form: $$Ortho = \begin{bmatrix} \frac{1}{aspect} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix}$$
- Aspect ratio taken into account
- Homogeneous, simpler to multiply through by d
- Must map z vals based on near, far planes (not yet)