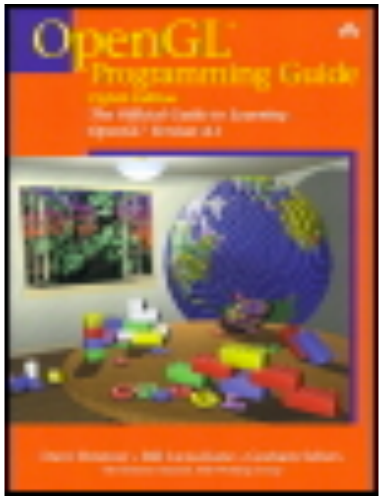
CSE 5542 - Real Time Rendering
Week 5

THE OHIO STATE UNIVERSITY

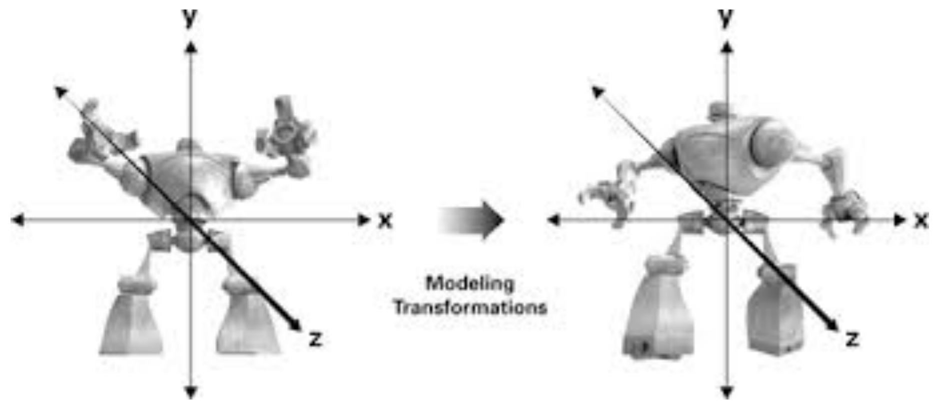
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Slides(Some) Courtesy – E. Angel and D. Shreiner

Much Content …

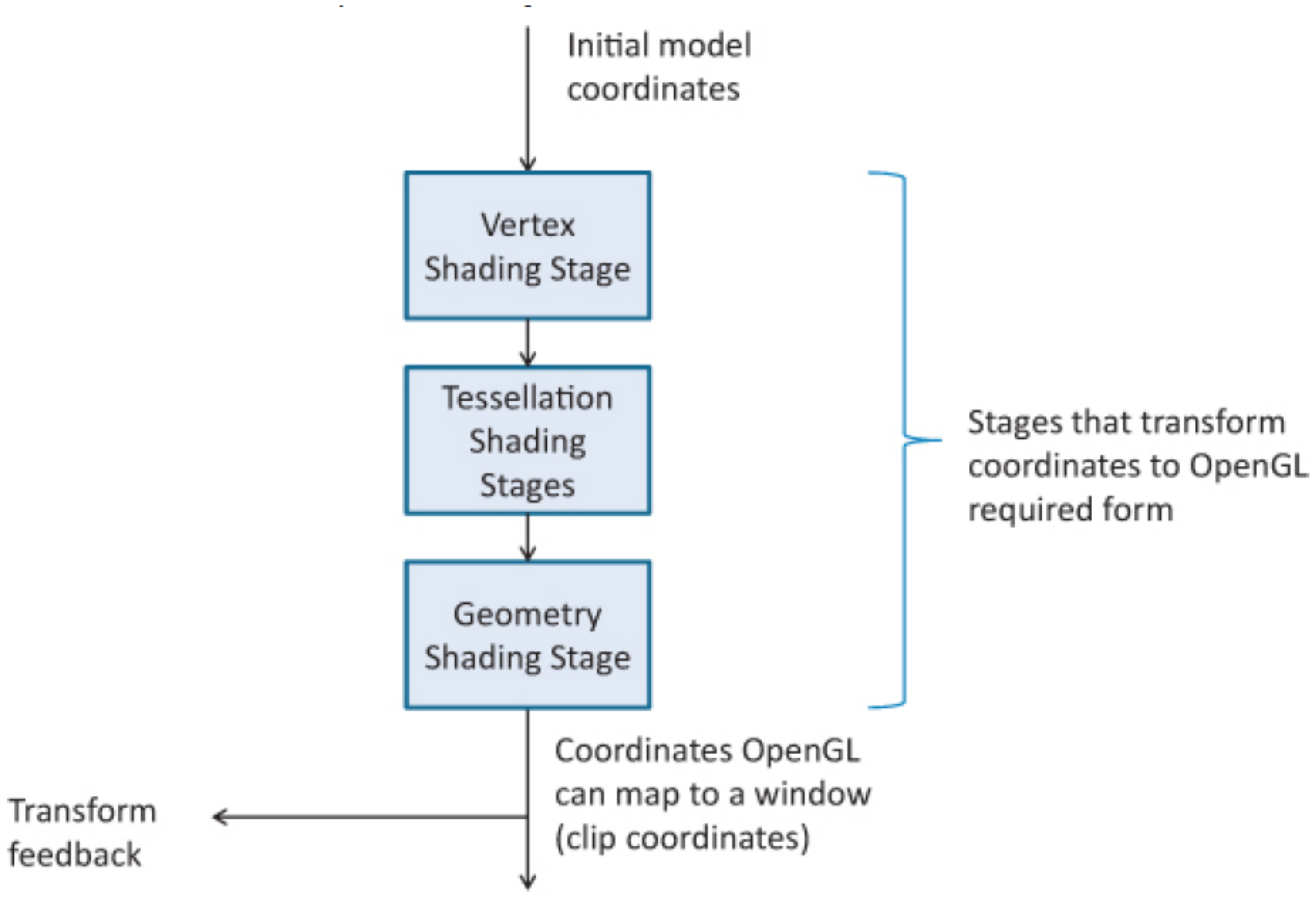


What ?



Transformations

Where in OpenGL ?



Initial model coordinates
Vertex Shading Stage
Tessellation Shading Stages
Geometry Shading Stage
Stages that transform coordinates to OpenGL required form
Coordinates OpenGL can map to a window (clip coordinates)
Transform feedback

In Vertex Shader

```
#version 330 core

uniform mat4 Transform; // stays the same for many vertices
                      // (primitive granularity)
in vec4 Vertex;         // per-vertex data sent each time this
                      // shader is run

void main()
{
    gl_Position = Transform * Vertex;
}
```

Typical Transform

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} ax & + & by & + & cz & + & dw \\ ex & + & fy & + & gz & + & hw \\ ix & + & jy & + & kz & + & lw \\ mx & + & ny & + & oz & + & pw \end{pmatrix}$$

Scaling

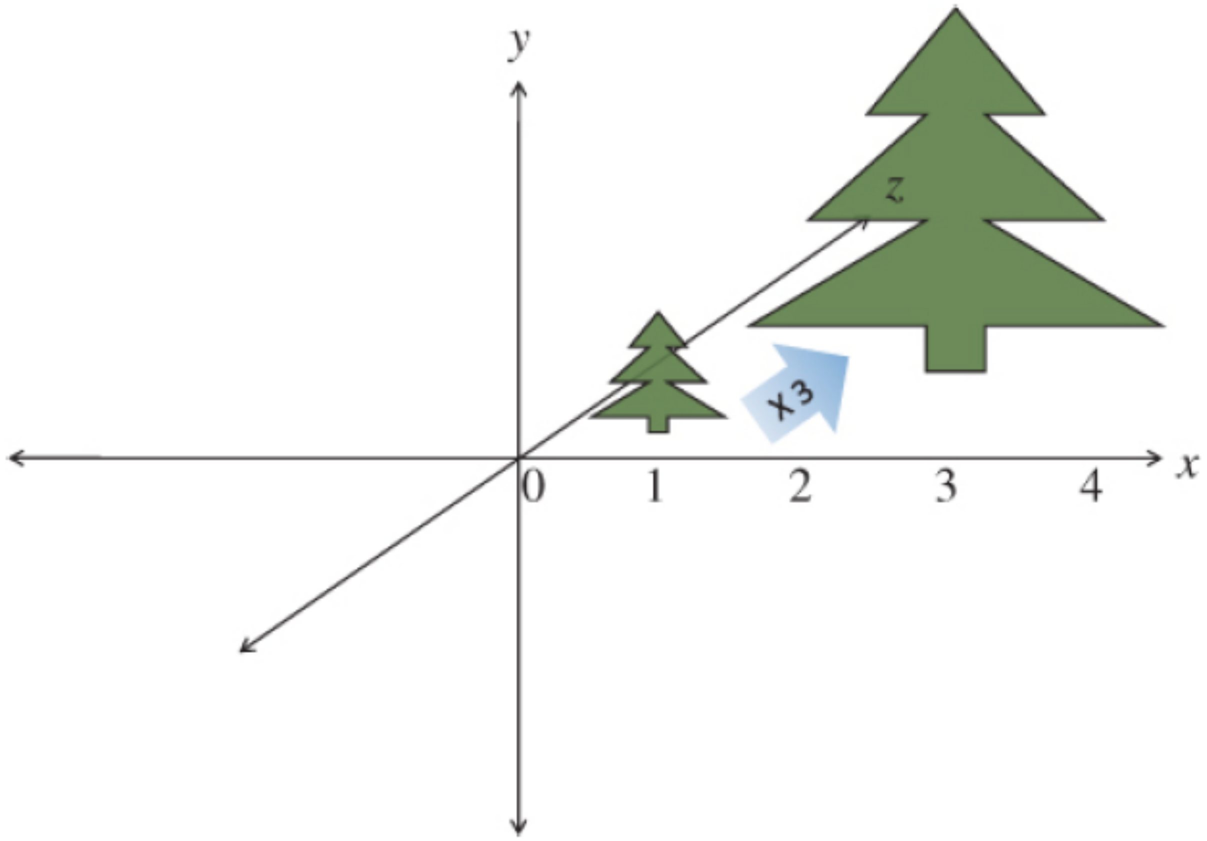


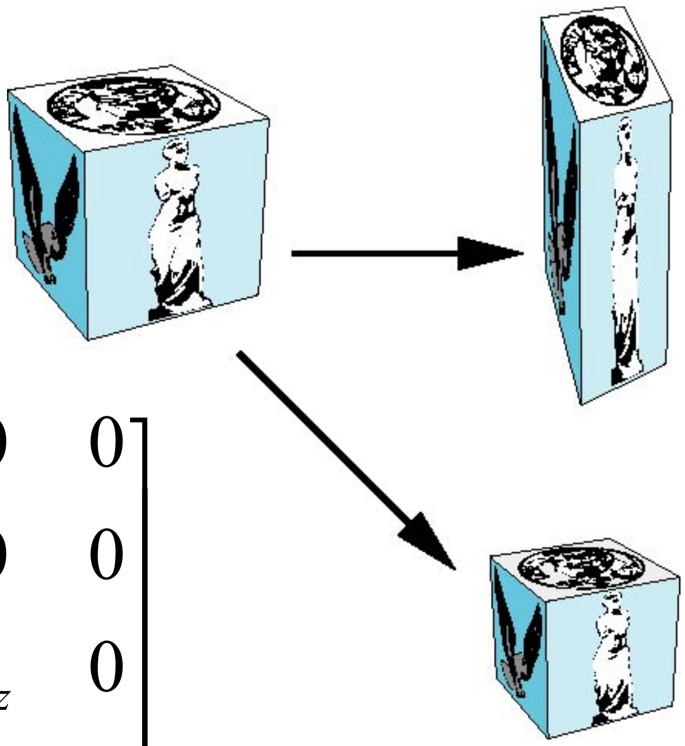
Figure 5.9. Scaling an object to three times its size (Note that if the object is off center, this also moves its center three times further from (0, 0, 0).)

Scaling

Expand or contract along each axis (fixed point of origin)

$x' = s_x x$
$y' = s_y x$
$z' = s_z x$

<u>$p' = Sp$</u>



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

Linear Transform - Scaling

Scaling

$$S = \begin{bmatrix} x & 0 & 0 & 1 \\ 0 & y & 0 & 1 \\ 0 & 0 & z & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and, } S^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 & 1 \\ 0 & \frac{1}{y} & 0 & 1 \\ 0 & 0 & \frac{1}{z} & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling – This will do !

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

3x3

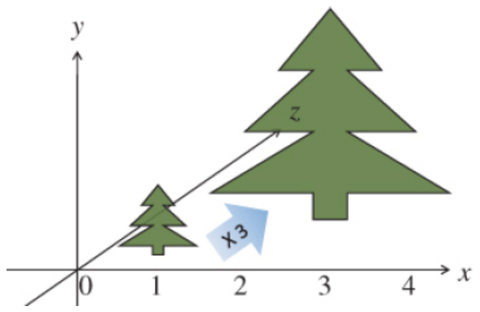
$$S = \begin{bmatrix} x & 0 \\ 0 & y \end{bmatrix}$$

2x2

As Matrices

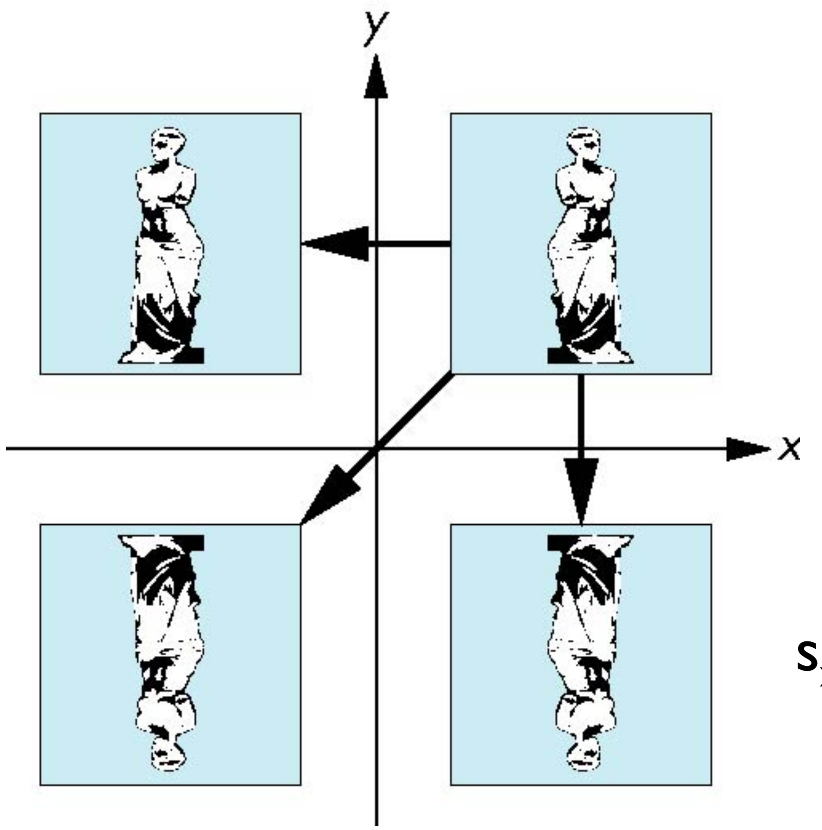
$$S = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\begin{pmatrix} 3x \\ 3y \\ 3z \\ 1 \end{pmatrix} = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$



Reflection

corresponds to negative scale factors



$s_x = -1 \; s_y = 1$

original

$s_x = -1 \; s_y = -1$

$s_x = 1 \; s_y = -1$

Rotation

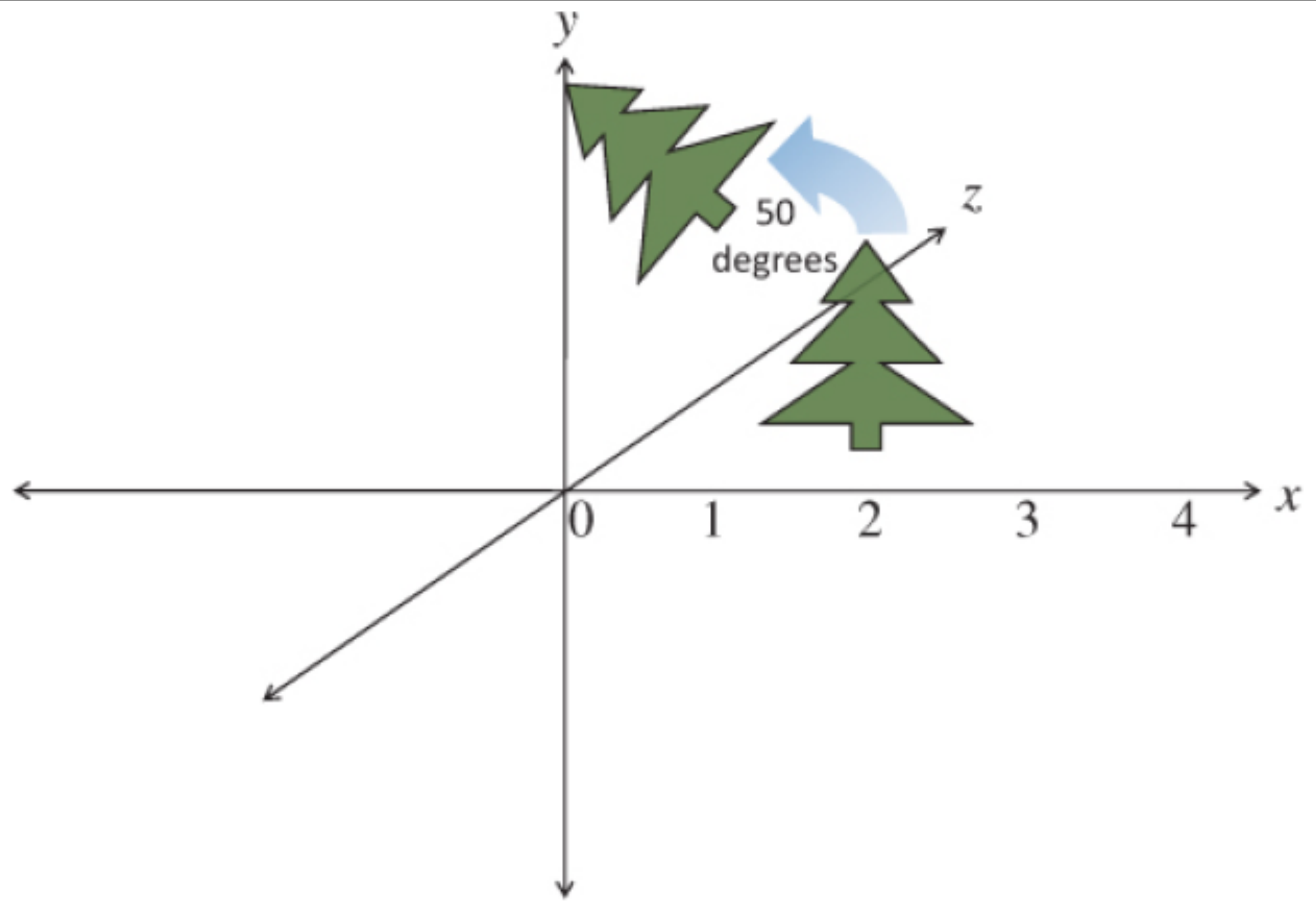


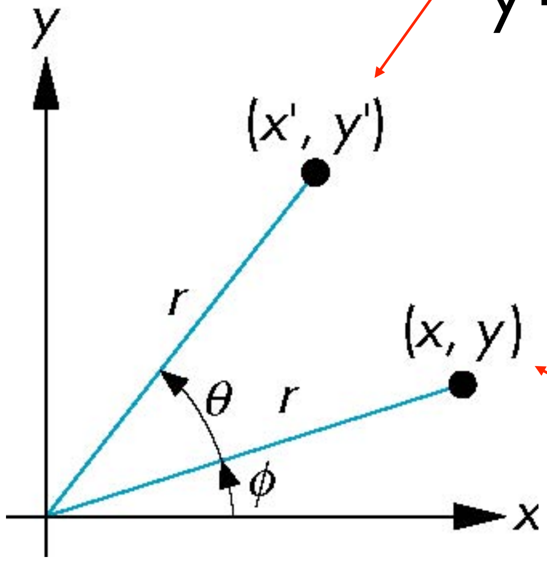
Figure 5.11. Rotation (Rotating an object 50 degrees in the *xy* plane, around the *z* axis. Note if the object is off center, it also revolves the object around the point (0, 0, 0).)

Rotation (2D)

Consider rotation about the origin by q degrees

– radius stays the same, angle increases by *q*

$x = r \cos(f + q)$
$y = r \sin(f + q)$



$$\boxed{\begin{aligned} x' &= x \cos q - y \sin q \\ y' &= x \sin q + y \cos q \end{aligned}}$$

$x = r \cos f$
$y = r \sin f$

Rotation about z axis

- Rotation about z axis in three dimensions leaves all points with the same z
  - Equivalent to rotation in two dimensions in planes of constant z

$$x' = x \cos q - y \sin q$$
$$y' = x \sin q + y \cos q$$
$$z' = z$$

  - or in homogeneous coordinates

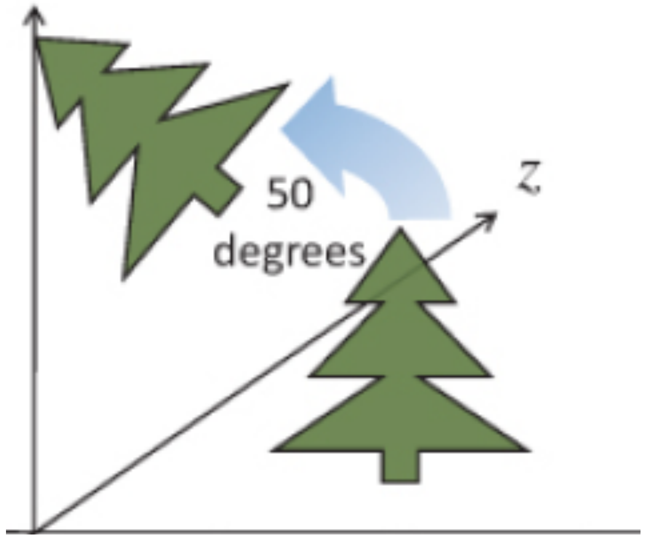
$$\mathbf{p}' = \mathbf{R}_{\mathbf{z}}(q)\mathbf{p}$$

Rotation Matrix

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

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Example



$$\begin{pmatrix} \cos 50 \cdot x - \sin 50 \cdot y \\ \sin 50 \cdot x + \cos 50 \cdot y \\ z \\ 1.0 \end{pmatrix} = \begin{bmatrix} \cos 50 & -\sin 50 & 0.0 & 0.0 \\ \sin 50 & \cos 50 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

X- & Y-axes

Same argument as for rotation about *z* axis

- For rotation about *x* axis, *x* is unchanged
- For rotation about *y* axis, *y* is unchanged

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

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

Linear Spaces

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} ax & + & by & + & cz & + & dw \\ ex & + & fy & + & gz & + & hw \\ ix & + & jy & + & kz & + & lw \\ mx & + & ny & + & oz & + & pw \end{pmatrix}$$

Now, some observations.

- Each component of the new vector is a linear function of all the components of the old vector, hence the need for 16 values in the matrix.
- The multiplication always takes the vector (0, 0, 0, 0) to (0, 0, 0, 0). This is characteristic of linear transformations and shows that if this was a 3 × 3 matrix times a three-component vector, why translation (moving) can't be done with a matrix multiply. We'll see how translating a three-component vector becomes possible with a 4 × 4 matrix and homogeneous coordinates.

Recap: Linear Independence

A set of vectors v_1, v_2, …, v_n is *linearly independent* if

$$\alpha_1 v_1+\alpha_2 v_2+..\ \alpha_n v_n=0 \text{ iff } \alpha_1=\alpha_2=\ldots=0$$

Recap: Coordinate Systems

- A basis $v_1, v_2, \ldots, v_n$
- A vector is written $v=\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$
- Scalars $\{\alpha_1, \alpha_2, \ldots A_n\}$- *representation* of v given basis
- Representation as a row or column array of scalars

$$\mathbf{a}=[\alpha_1 \ \ \alpha_2 \ \ \ldots \ \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdot \\ \alpha_n \end{bmatrix}$$

Change of Coordinate Systems

- Consider two representations of same vector with respect to two different bases:

$$\mathbf{a}=[a_1\ a_2\ a_3]$$
$$\mathbf{b}=[b_1\ b_2\ b_3]$$

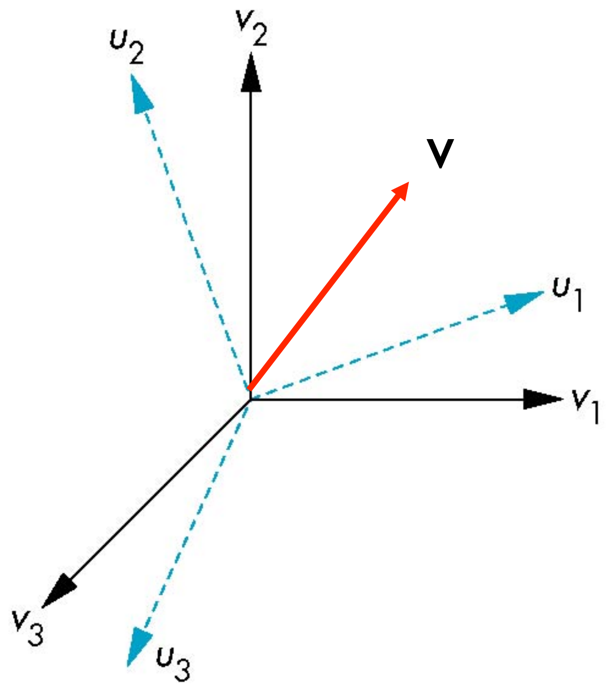
where

$$v=a_1 v_1+a_2 v_2+a_3 v_3=[a_1\ a_2\ a_3]\,[v_1\ v_2\ v_3]^T$$
$$=b_1 u_1+b_2 u_2+b_3 u_3=[b_1\ b_2\ b_3]\,[u_1\ u_2\ u_3]^T$$

Second in Terms of First

Each of the basis vectors, u1,u2, u3, are vectors that can be represented in terms of the first basis

$$u_1 = g_{11}v_1+g_{12}v_2+g_{13}v_3$$
$$u_2 = g_{21}v_1+g_{22}v_2+g_{23}v_3$$
$$u_3 = g_{31}v_1+g_{32}v_2+g_{33}v_3$$



Recap: Matrix Form

The coefficients define a 3 x 3 matrix

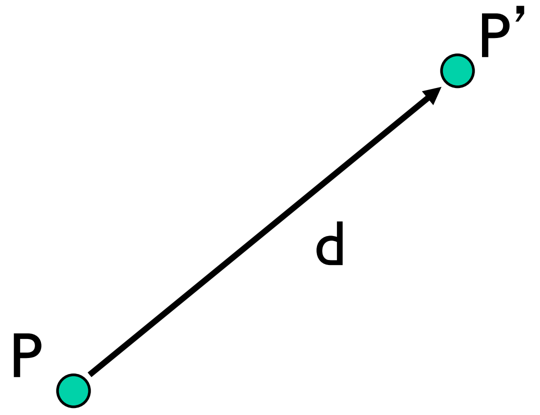
$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix}$$

and the bases can be related by

$$\mathbf{a}=\mathbf{M}^T\mathbf{b}$$

Translation

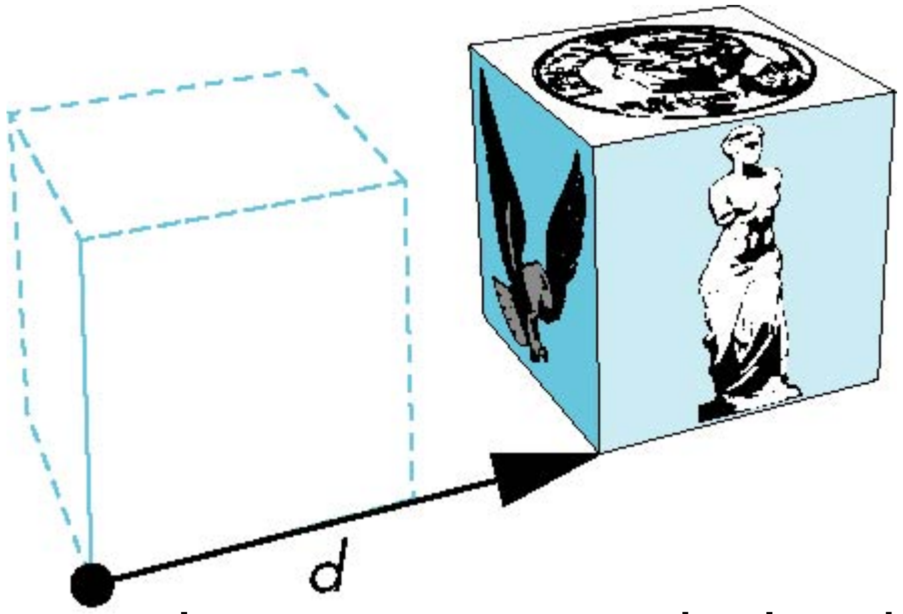
- Move (translate, displace) a point to a new location



- Displacement determined by a vector d
  - Three degrees of freedom
  - P' =P+d

Translating in 3D

object



translation: every point displaced by same vector

Translation

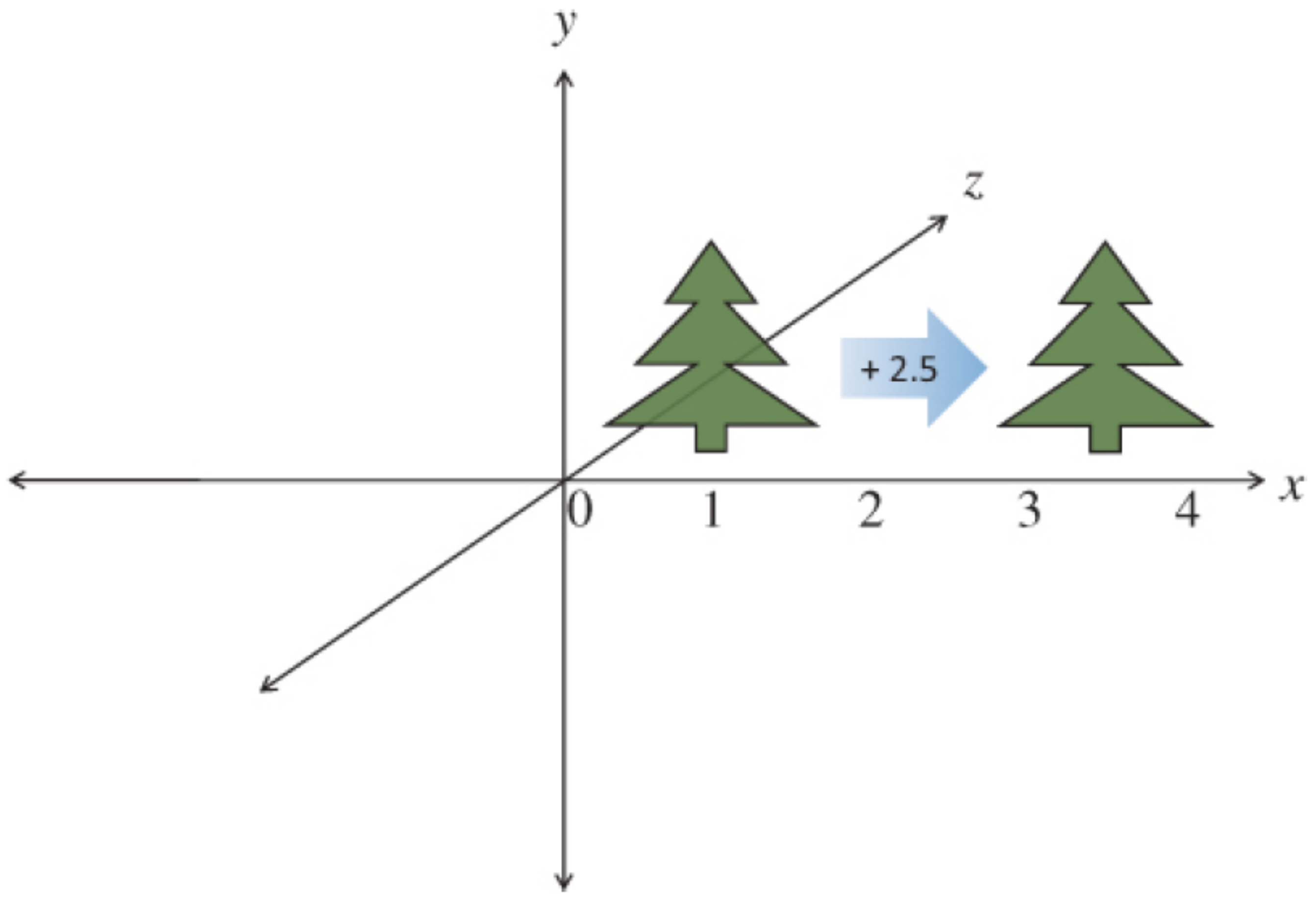


Figure 5.8. Translating an object 2.5 in the *x* direction

With 2D/3D Vectors

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \\ \chi \end{bmatrix}$$

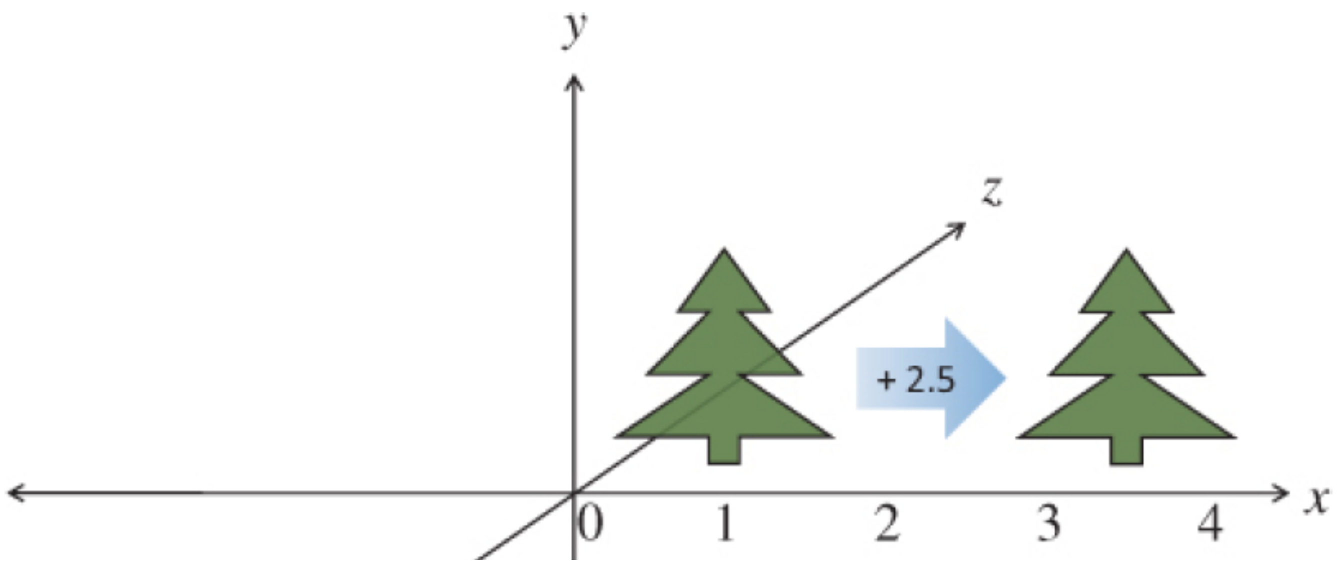
In Matrix Form

Translation

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and, } T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As Matrices

$$\begin{pmatrix} x+2.5 \\ y \\ z \\ 1.0 \end{pmatrix} = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.5 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix}$$

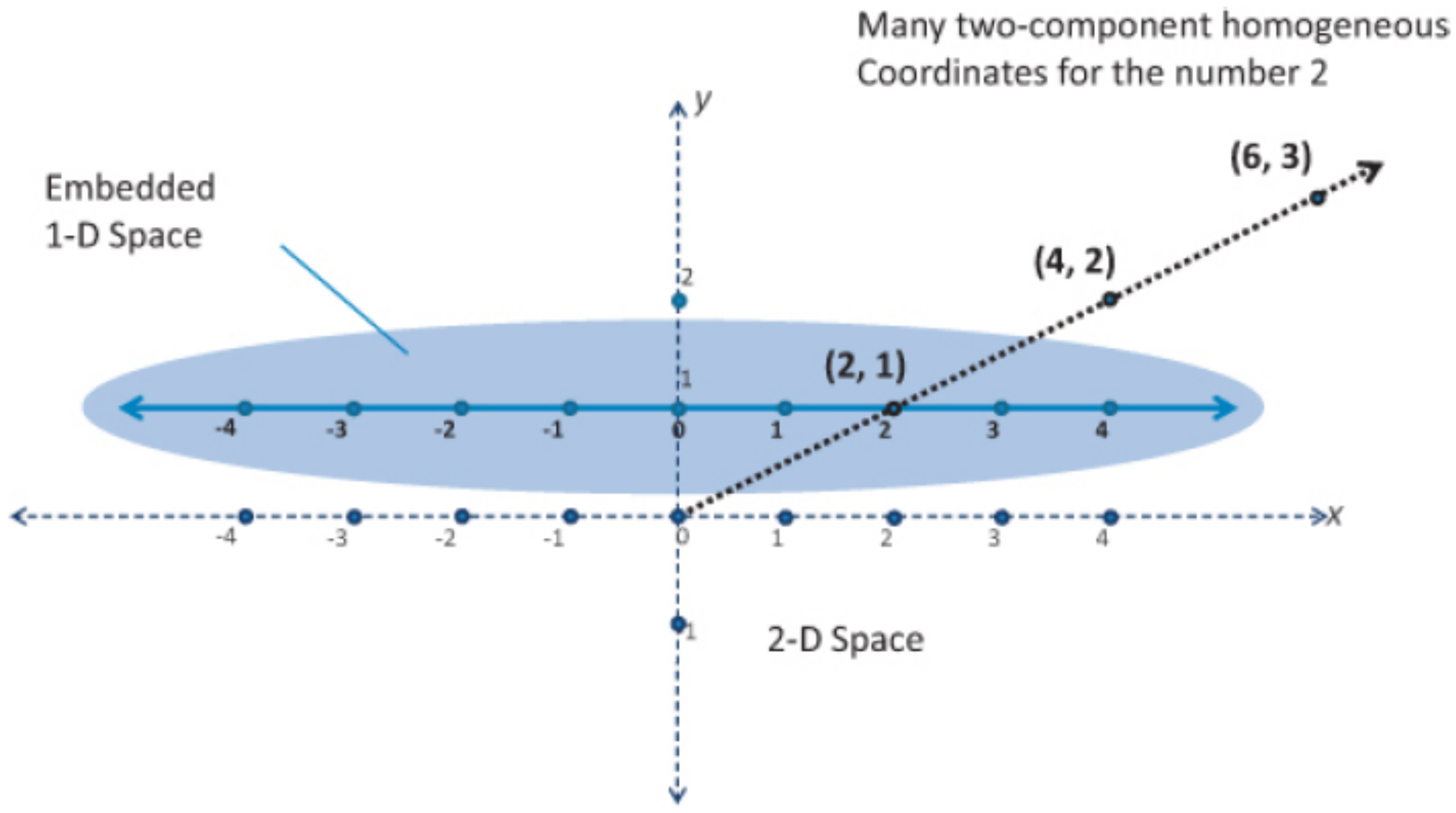


As Code

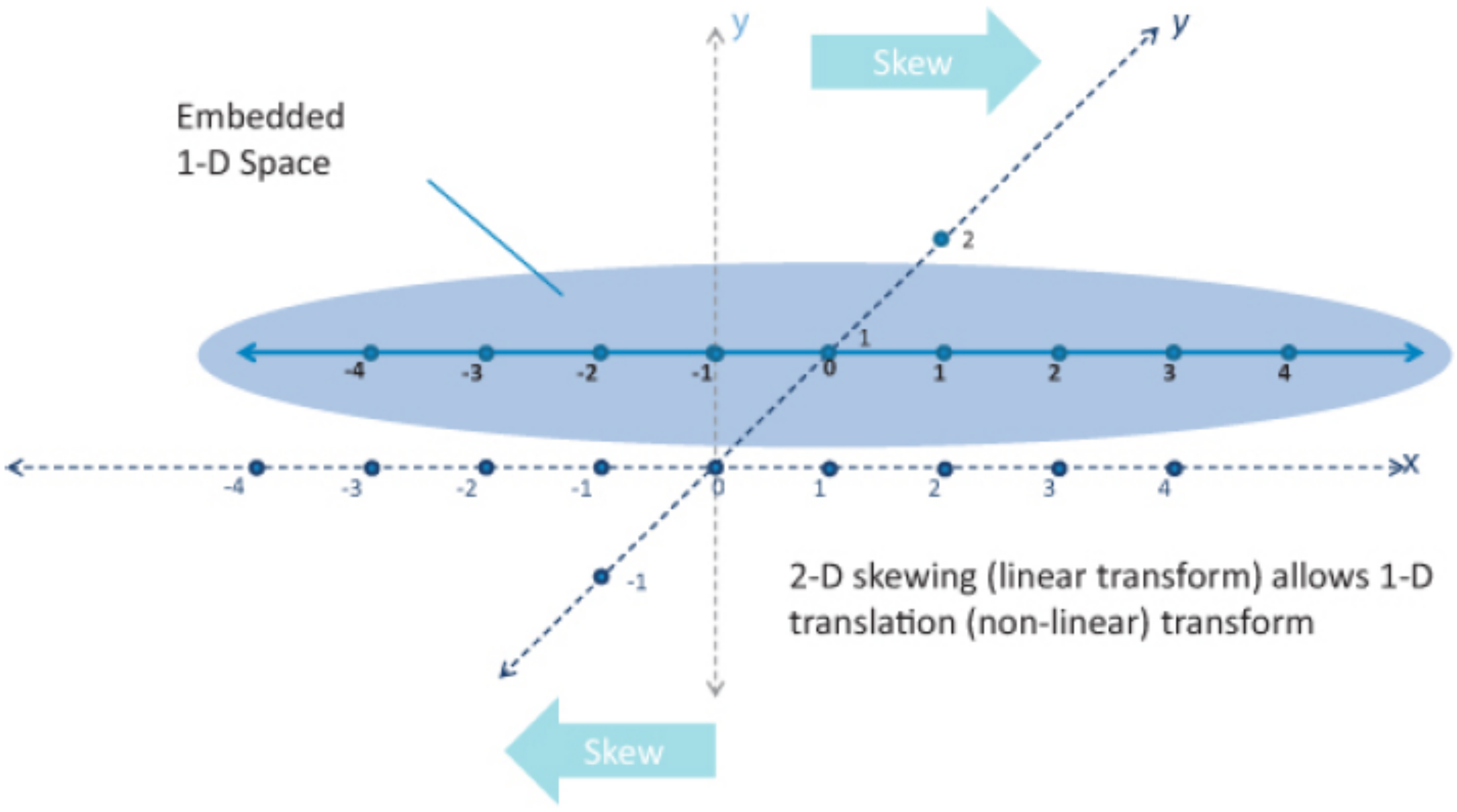
```
// Application (C++) code
#include "vmath.h"
     .
     .
     .
// Make a transformation matrix that translates coordinates by (1, 2, 3)
vmath::mat4 translationMatrix = vmath::translate(1.0, 2.0, 3,0);

// Set this matrix into the current program.
glUniformMatrix4fv(matrix_loc, 1, GL_FALSE, translationMatrix);
     .
     .
```

Homogenous Coordinates



Homogenous Coordinates



2-D skewing (linear transform) allows 1-D translation (non-linear) transform

Allows Linearity

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \to \begin{pmatrix} ax & + & by & + & cz & + & dw \\ ex & + & fy & + & gz & + & hw \\ ix & + & jy & + & kz & + & lw \\ mx & + & ny & + & oz & + & pw \end{pmatrix}$$

Now, some observations.

- Each component of the new vector is a linear function of all the components of the old vector, hence the need for 16 values in the matrix.
- The multiplication always takes the vector (0, 0, 0, 0) to (0, 0, 0, 0). This is characteristic of linear transformations and shows that if this was a 3 × 3 matrix times a three-component vector, why translation (moving) can't be done with a matrix multiply. We'll see how translating a three-component vector becomes possible with a 4 × 4 matrix and homogeneous coordinates.

Homogeneous Coordinates

Homogeneous coordinates for [x y z]:

$\mathbf{p} = [x' \; y' \; z' \; w]^T = [wx \; wy \; wz \; w]^T$

Three dimensional point (for $w \neq 0$) by

$x \leftarrow x'/w$, $y \leftarrow y'/w$, $z \leftarrow z'/w$

If w=0, the representation is that of a vector

Replaces points in 3D by lines through origin in 4D dimensions

For w=1, the representation of a point is [x y z 1]

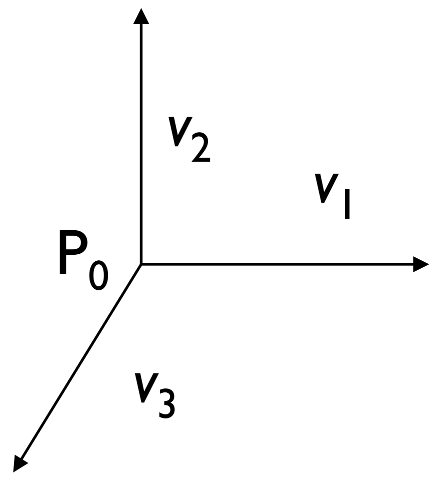
In Computer Graphics

Homogeneous coordinates key

- All standard transformations: rotation, translation, scaling
- Implemented matrix multiplications with 4 x 4 matrices
- Hardware pipeline works with 4D representations
- Orthographic viewing: w=0 – vectors & w=1 – points
- For perspective we need a *perspective division*

Recap: Frames

- A coordinate system is insufficient for points
- *Origin* & basis vectors form a *frame*



Recap : Unified Representation

- Frame determined by (P_0, v_1, v_2, v_3)

- For this frame, every vector can be written as

  $v = a_1 v_1 + a_2 v_2 + \ldots. + a_n v_n$

- Every point can be written as

  $P = P_0 + b_1 v_1 + b_2 v_2 + \ldots. + b_n v_n$

Recap : Unified Representation

Since 0•P = **0** and 1•P =P

$v=a_1 v_1 + a_2 v_2 + a_3 v_3 = [a_1\ a_2\ a_3\ 0]\ [v_1\ v_2\ v_3\ P_0]^T$

$P = P_0 + b_1 v_1 + b_2 v_2 + b_3 v_3 = [b_1\ b_2\ b_3\ 1]\ [v_1\ v_2\ v_3\ P_0]^T$

Thus we obtain the four-dimensional *homogeneous coordinate* representation

$\mathbf{v} = [a_1\ a_2\ a_3\ 0]^T$

$\mathbf{p} = [b_1\ b_2\ b_3\ 1]^T$

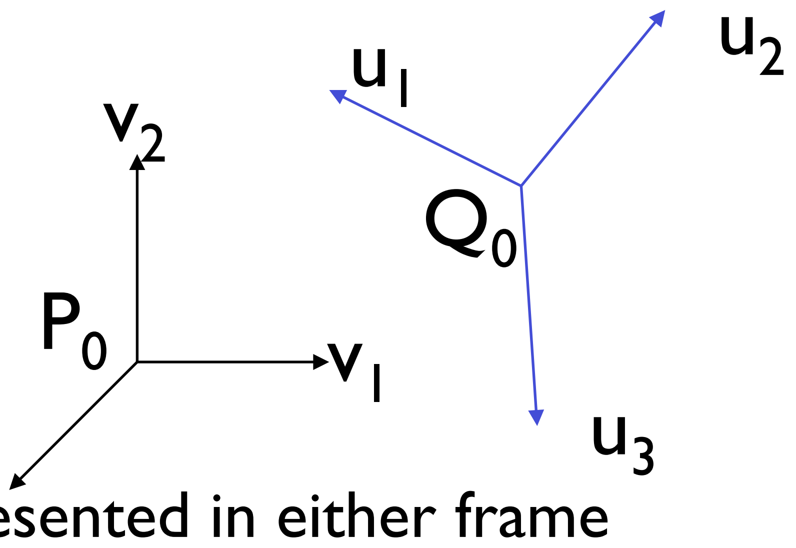
Change of Frames

- We can apply a similar process in homogeneous coordinates to the representations of both points and vectors

Consider two frames:

(P_0, v_1, v_2, v_3)

(Q_0, u_1, u_2, u_3)



- Any point or vector can be represented in either frame
- We can represent Q_0, u_1, u_2, u_3 in terms of P_0, v_1, v_2, v_3

One Frame in Terms of Other

Extending what we did with change of bases

$u_1 = \gamma_{11}v_1+\gamma_{12}v_2+\gamma_{13}v_3$
$u_2 = \gamma_{21}v_1+\gamma_{22}v_2+\gamma_{23}v_3$
$u_3 = \gamma_{31}v_1+\gamma_{32}v_2+\gamma_{33}v_3$
$Q_0 = \gamma_{41}v_1+\gamma_{42}v_2+\gamma_{43}v_3 +\gamma_{44}P_0$

defining a 4 x 4 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

Working with Representations

Within the two frames any point or vector has a representation of the same form

$\mathbf{a}=[a_1\ a_2\ \ a_3\ a_4\ ]$ in the first frame
$\mathbf{b}=[b_1\ b_2\ \ b_3\ b_4\ ]$ in the second frame

where $a_4 = b_4 = 1$ for points and $a_4 = b_4 = 0$ for vectors and

$$\mathbf{a}=\mathbf{M}^T\mathbf{b}$$

The matrix **M** is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine Transformations

- Every linear transformation is equivalent to a change in frames
- Every affine transformation preserves lines
- However, an affine transformation has only 12 *degrees of freedom* because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations

General Structure

$$\begin{bmatrix} & SR & & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Inverses

- Can compute inverses use simple geometric observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Rotation: $\mathbf{R}^{-1}(q) = \mathbf{R}(-q)$
    - Holds for any rotation matrix
    - Note that since cos(-q) = cos(q) and sin(-q)=-sin(q)

    $\mathbf{R}^{-1}(q) = \mathbf{R}^{T}(q)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$

Concatenation

- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices

- Since same transformation is applied to many vertices, the cost of forming a matrix **M=ABCD** is not significant compared to the cost of computing **Mp** for many vertices **p**

In-situ Transformations

In Place Scaling

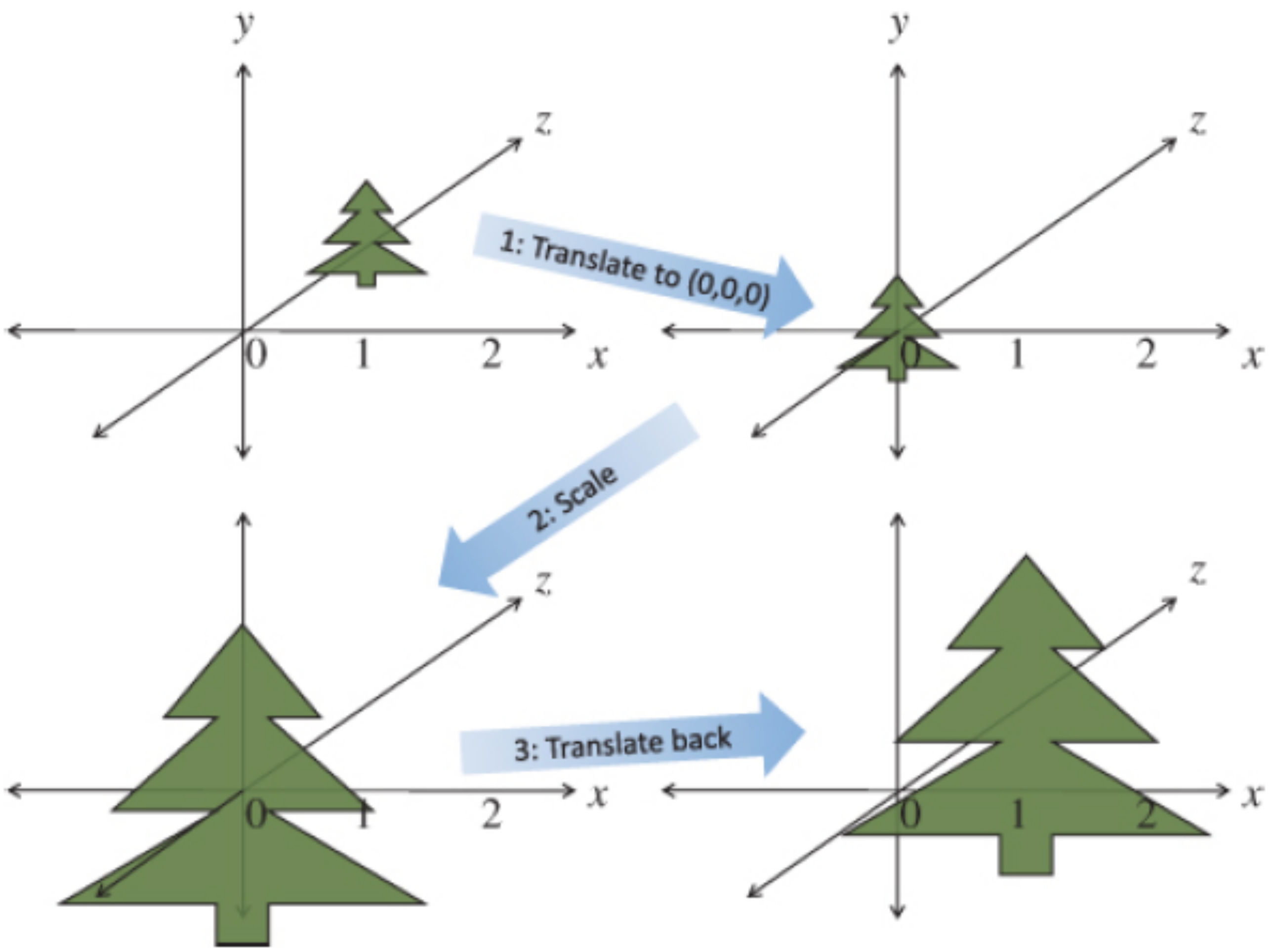
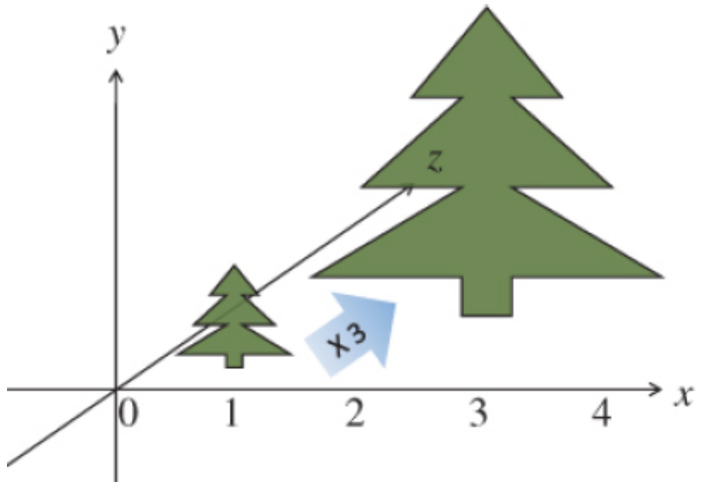
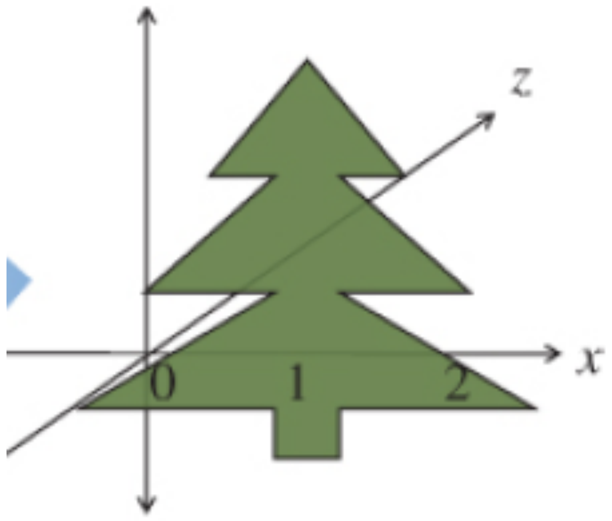


Figure 5.10. Scaling an object in place (Scale in place by moving to (0, 0, 0), scaling, and then moving it back.)

As Opposed To





As Matrices and code

$$v' = T^{-1}(S(Tv))$$

or

$$v' = (T^{-1}ST)v$$

which allows for pre-multiplication of the three matrices into a single matrix.

$$M = T^{-1}ST$$

$$v' = Mv$$

M now does the complete job of scaling an off-center object.

```
vmath::mat4 translateMatrix = vmath::translate(1.0, 2.0, 3,0);
vmath::mat4 scaleMatrix = vmath::scale(5.0);
vmath::mat4 scaleTranslateMatrix = scaleMatrix * translateMatrix;
```

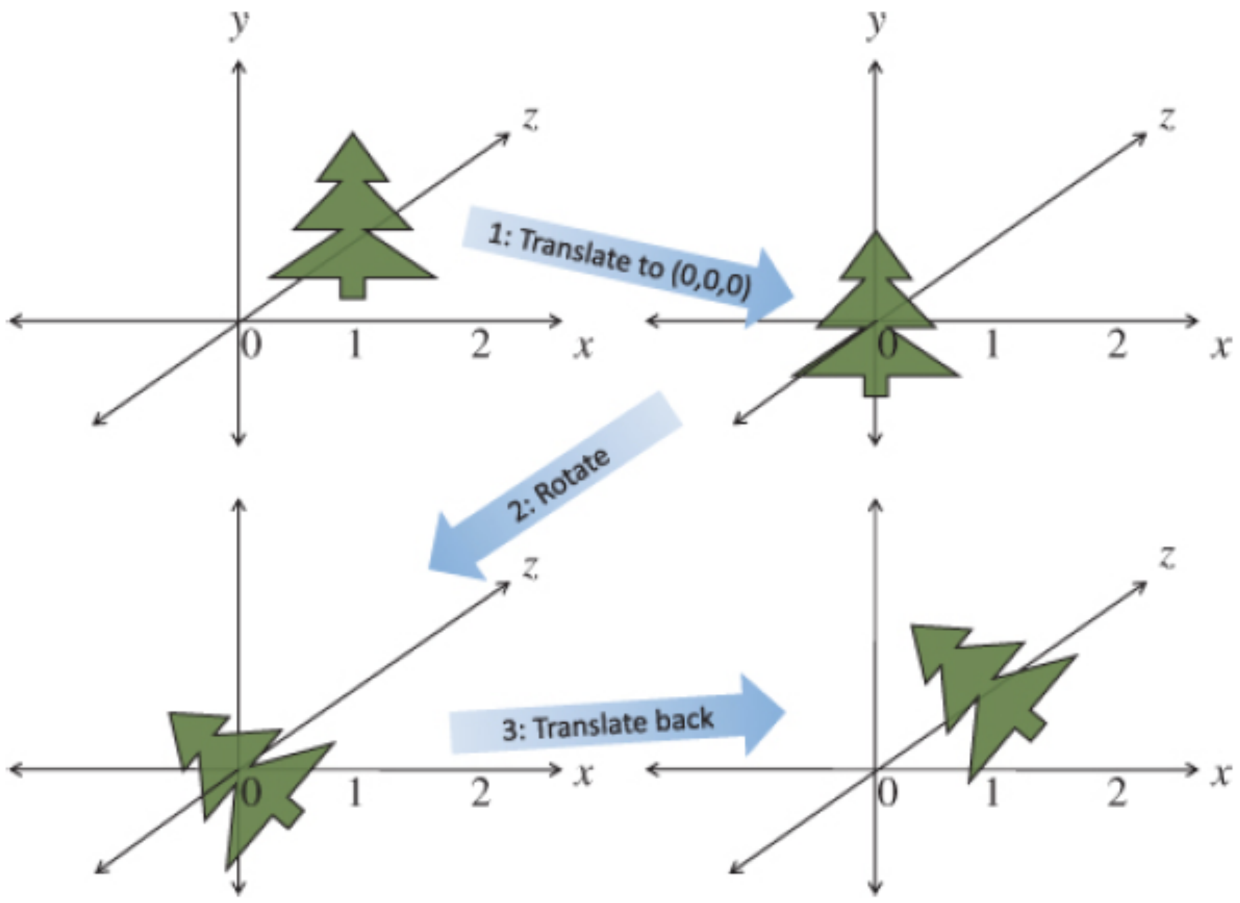


Figure 5.12. Rotating in place (Rotating an object in place by moving it to (0, 0, 0), rotating, and then moving it back.)

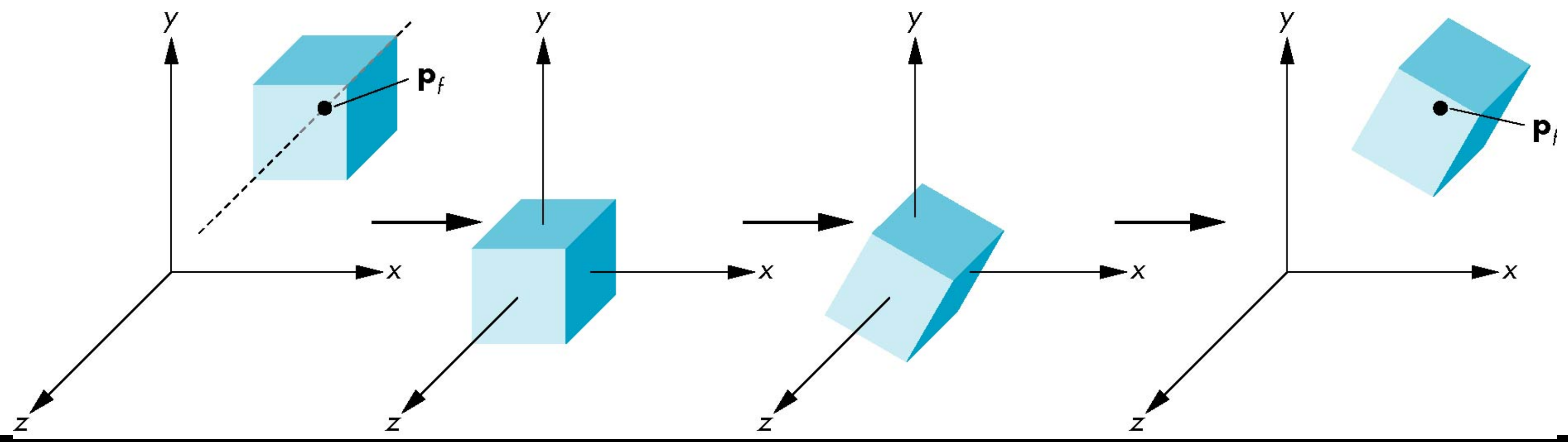
Fixed Point Other Than Origin

Move fixed point to origin

Rotate

Move fixed point back

$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$



Order of Transformations

- Note that matrix on the right is the first applied
- Mathematically, the following are equivalent

$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

- In terms of column matrices

$$\mathbf{p}'^{T} = \mathbf{p}^{T}\mathbf{C}^{T}\mathbf{B}^{T}\mathbf{A}^{T}$$

In Shaders

```
uniform mat4 ViewT, ViewR, ModelT, ModelR,
ModelS, Project;
in vec4 Vertex;

void main()
{
   gl_Position = Project
                 * ModelS * ModelR * ModelT
                 * ViewR * ViewT
                 * Vertex;
}
```

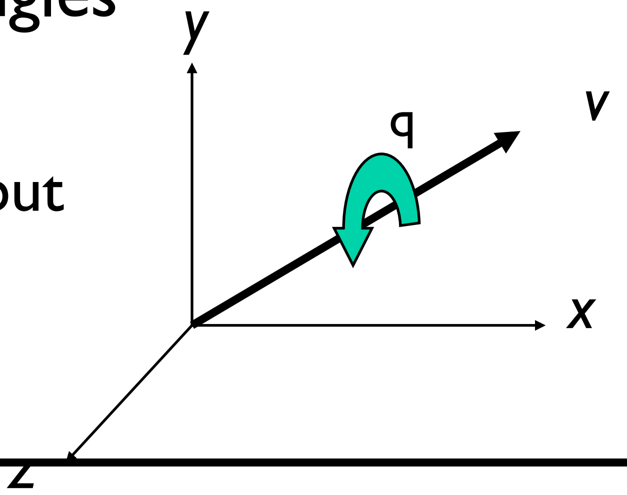
General Rotation About Origin

A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R}(\theta) = \mathbf{R}_z(\theta_z)\ \mathbf{R}_y(\theta_y)\ \mathbf{R}_x(\theta_x)$$

$\theta_x\ \theta_y\ \theta_z$ are called the Euler angles

Note that rotations do not commute
We can use rotations in another order but with different angles



Let $v = (x, y, z)^T$, and $u = v/\|v\| = (x', y', z')^T$. Also let

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix}$$

and,

$$M = uu^T + \cos\theta(I - uu^T) + \sin\theta\, S$$

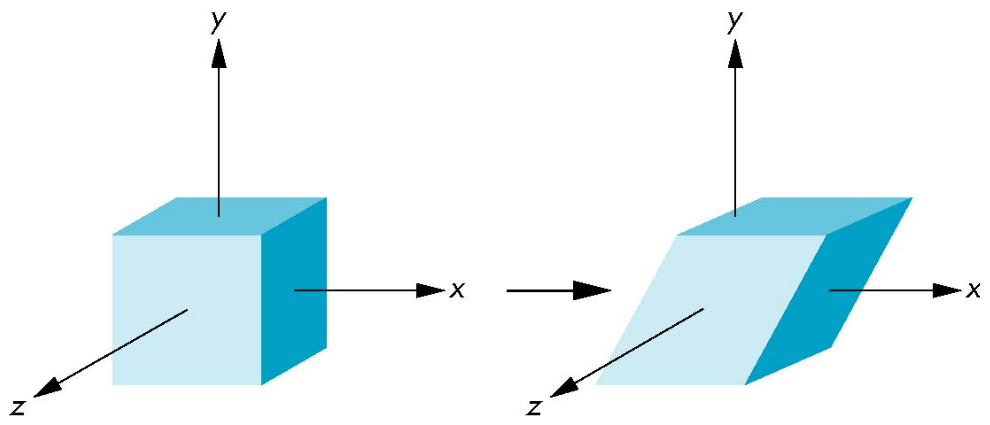
$$R = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$u\,u^T = \begin{bmatrix} x^2 & x\,y & x\,z \\ x\,y & y^2 & y\,z \\ x\,z & y\,z & z^2 \end{bmatrix}$$

Shear

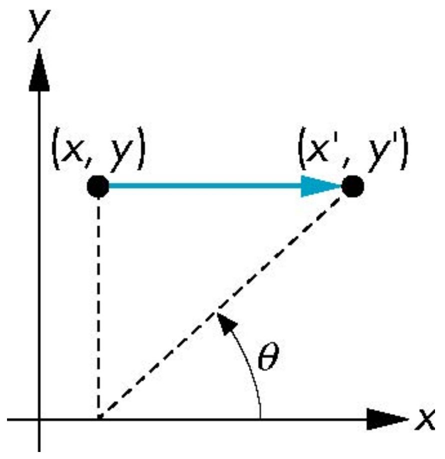
- Helpful to add one more basic transformation
- Equivalent to pulling faces in opposite directions



Shear Matrix

Consider simple shear along *x* axis

$$x' = x + y \cot\theta$$
$$y' = y$$
$$z' = z$$



$$\mathbf{H}(q) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewing Transform

World & Camera Frames

- Changes in frame are then defined by 4 x 4 matrices
- In OpenGL, base frame is the world frame
- Represent entities in the camera frame by changing world representation using model-view matrix
- Initially these frames are the same (**M**=**I**)

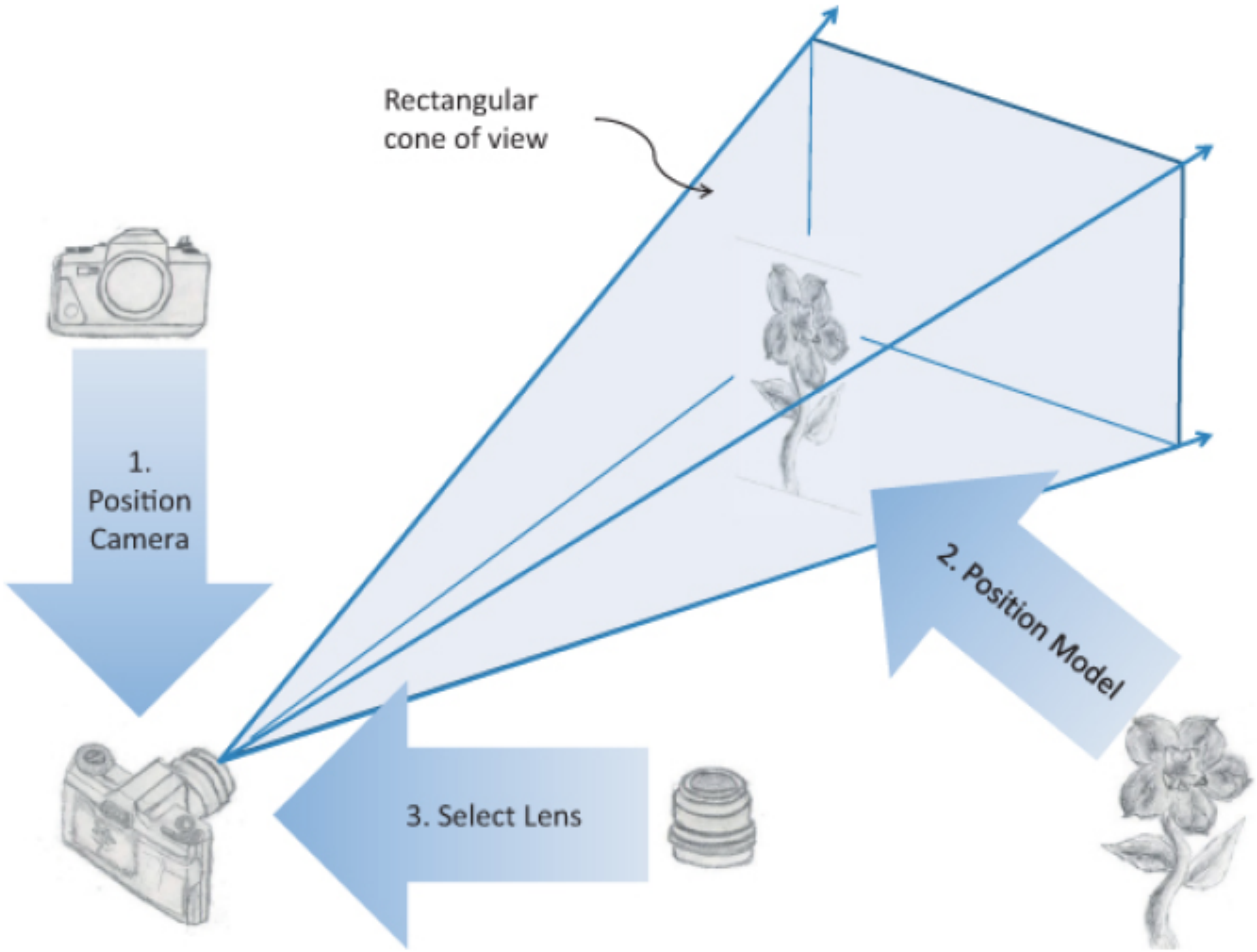


Figure 5.1. Steps in configuring and positioning the viewing frustum

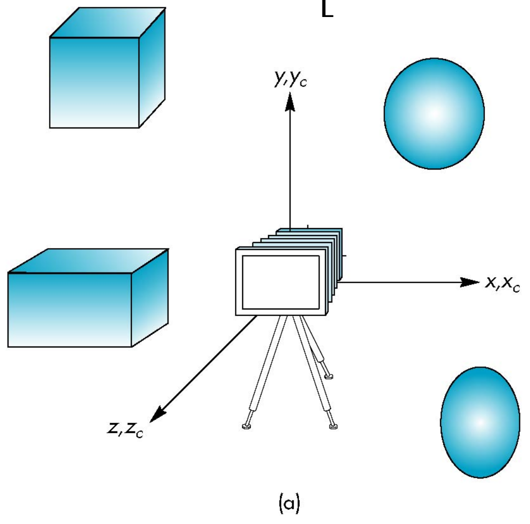
Steps

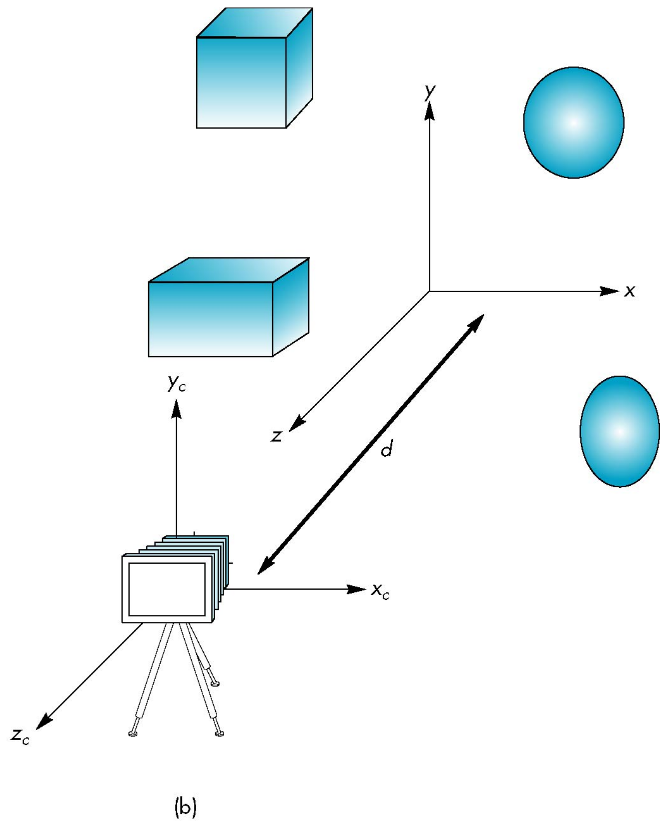
1. Move your camera to the location you want to shoot from and point the camera the desired direction (viewing transformation).
2. Move the subject to be photographed into the desired location in the scene (modeling transformation).
3. Choose a camera lens or adjust the zoom (projection transformation).
4. Take the picture (apply the transformations).
5. Stretch or shrink the resulting image to the desired picture size (viewport transformation). For 3D graphics, this also includes stretching or shrinking the depth (depth-range scaling). This is not to be confused with Step 3, which selected *how much of the scene* to capture, not *how much to stretch* the result.

Moving the Camera

If objects are on both sides of z=0, move camera frame

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$





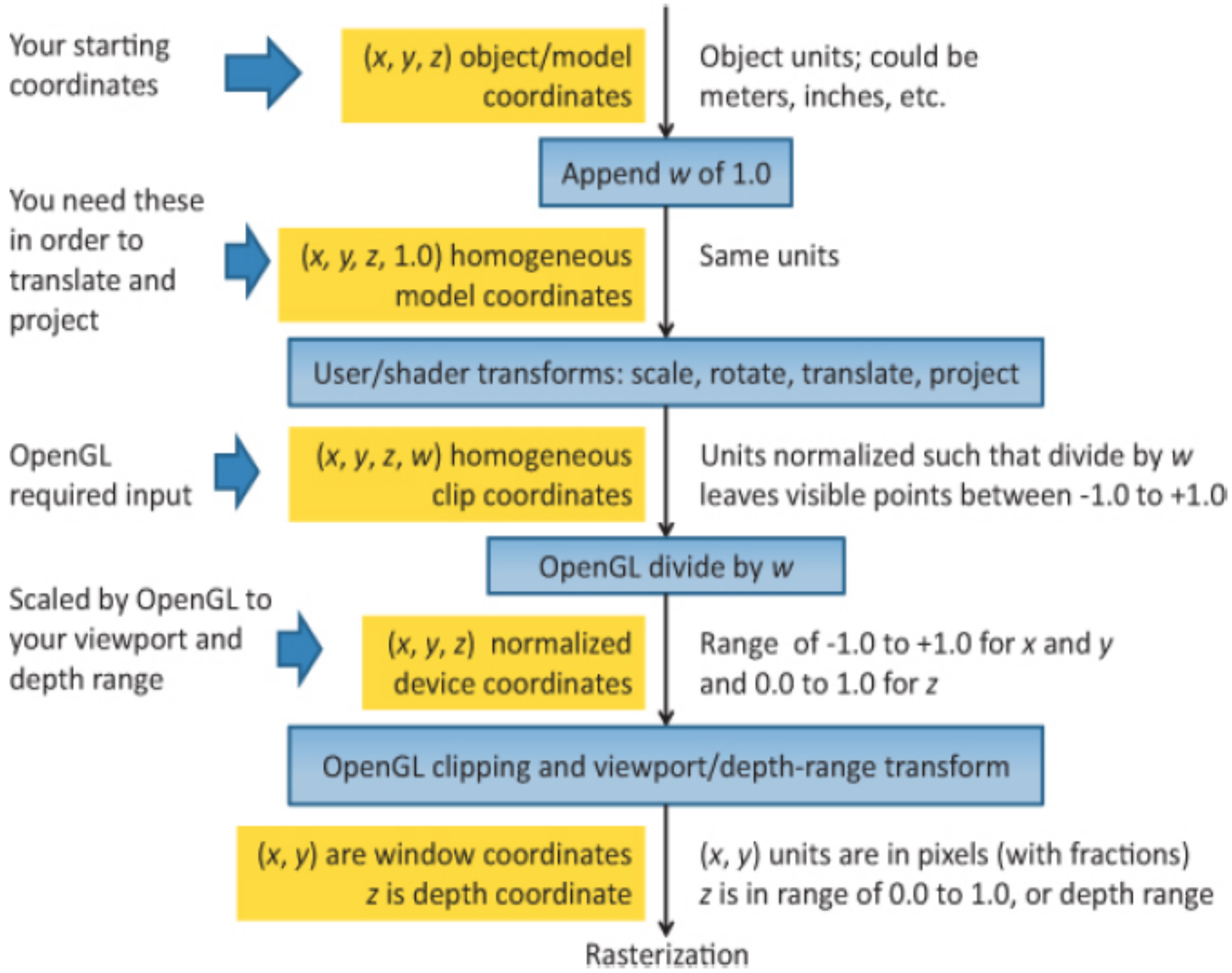
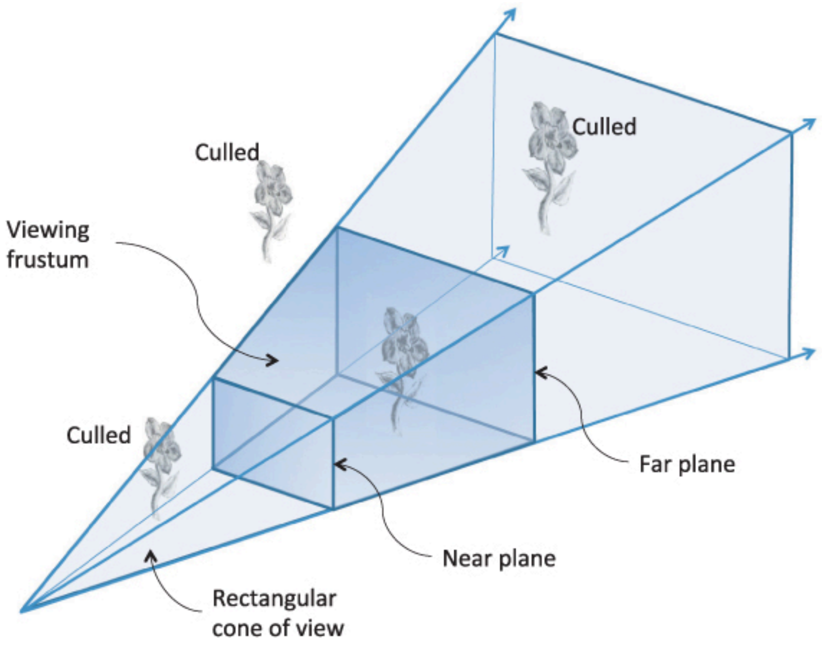

Your starting coordinates
(x, y, z) object/model coordinates
Object units; could be meters, inches, etc.
Append w of 1.0
You need these in order to translate and project
(x, y, z, 1.0) homogeneous model coordinates
Same units
User/shader transforms: scale, rotate, translate, project
OpenGL required input
(x, y, z, w) homogeneous clip coordinates
Units normalized such that divide by w leaves visible points between -1.0 to +1.0
OpenGL divide by w
Scaled by OpenGL to your viewport and depth range
(x, y, z) normalized device coordinates
Range of -1.0 to +1.0 for x and y and 0.0 to 1.0 for z
OpenGL clipping and viewport/depth-range transform
(x, y) are window coordinates z is depth coordinate
(x, y) units are in pixels (with fractions) z is in range of 0.0 to 1.0, or depth range
Rasterization

Figure 5.2. Coordinate systems required by OpenGL (The coordinate systems are the boxes on the left. The central boxes transform from one coordinate system to the next. Units are described to the right.)

Perspective

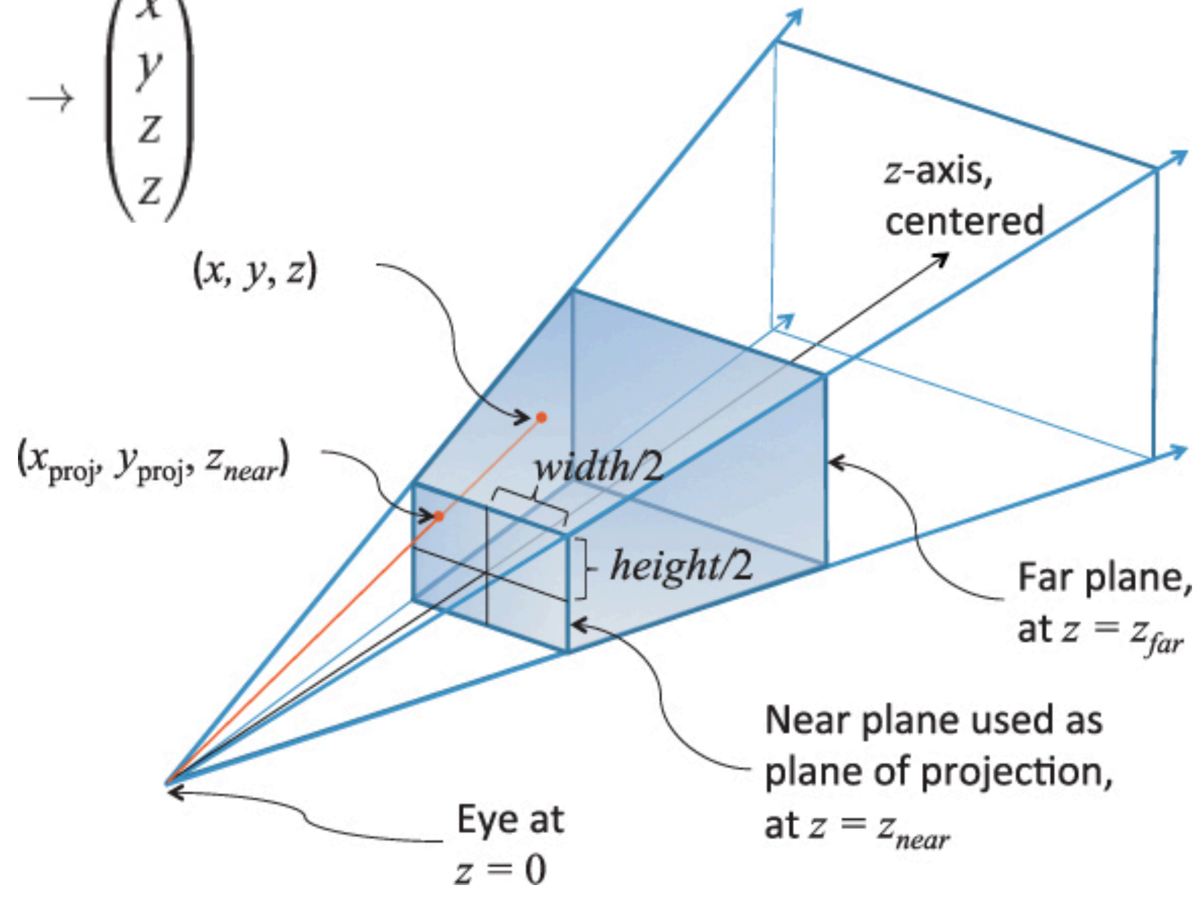
$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

P is defined as long as $l \neq r$, $t \neq b$, and $n \neq f$.



Details

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1.0 \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ z \\ z \end{pmatrix}$$



To Note

z values will end up at 1.0 !

Lose information about depth

No scaling result to the [–1.0, 1.0] range

Two Cases

Symmetric, centered frustum, where z-axis is centered in the cone

Asymmetric frustum- view thru a window and near it, but not toward its middle

Symmetric Case

- Project points in frustum onto near plane
- Line from (0, 0, 0) makes ratio of z to x same, and same for z to y for all points

$$\frac{z_{near}}{z} = \frac{x_{proj}}{x} \qquad \frac{z_{near}}{z} = \frac{y_{proj}}{y}$$

- xproj = x · znear, yproj = y · znear
- Scale window size to [–1.0, 1.0]

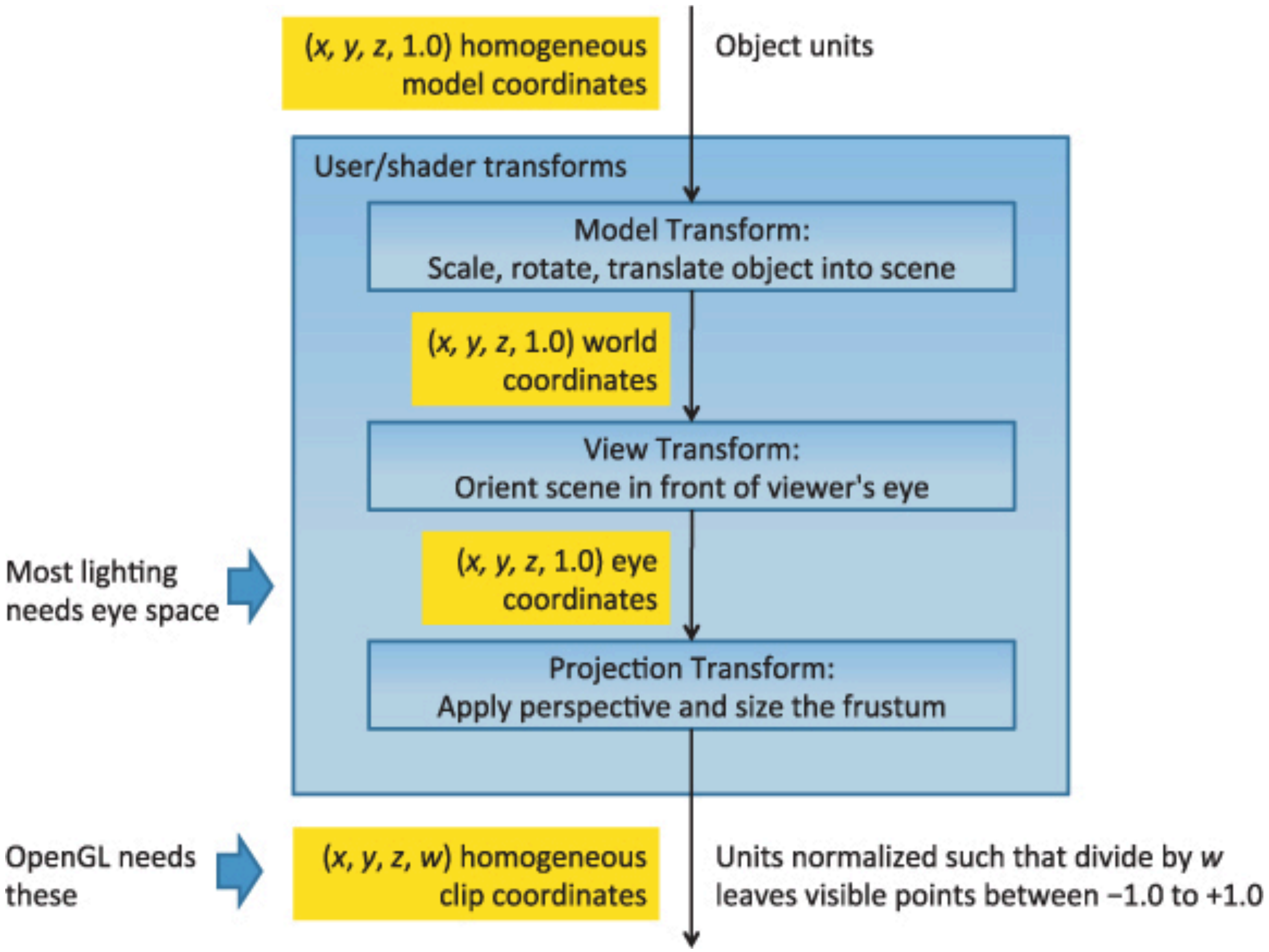
$$\begin{bmatrix} \frac{z_{near}}{width/2} & 0.0 & 0.0 & 0.0 \\ 0.0 & \frac{z_{near}}{height/2} & 0.0 & 0.0 \\ 0.0 & 0.0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{2z_{far}z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

Symmetric Case

$$\begin{bmatrix} \frac{z_{near}}{width/2} & 0.0 & 0.0 & 0.0 \\ 0.0 & \frac{z_{near}}{height/2} & 0.0 & 0.0 \\ 0.0 & 0.0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{2z_{far}z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

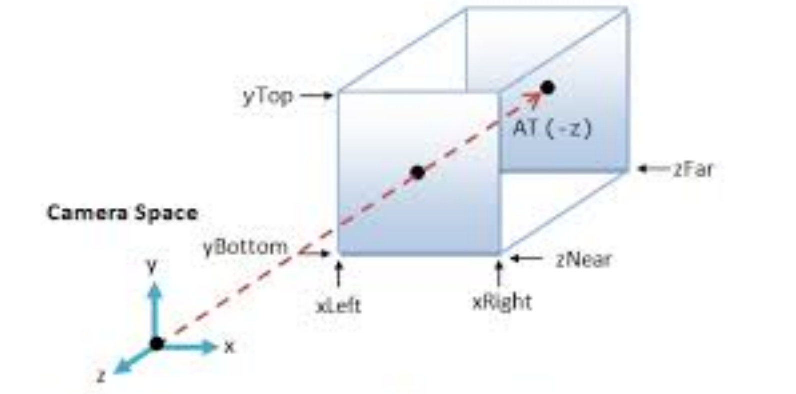
Asymmetric

$$\begin{bmatrix} \frac{z_{near}}{width/2} & 0.0 & \frac{left+right}{width/2} & 0.0 \\ 0.0 & \frac{z_{near}}{height/2} & \frac{top+bottom}{height/2} & 0.0 \\ 0.0 & 0.0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{2z_{far}z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$


(x, y, z, 1.0) homogeneous model coordinates
Object units
User/shader transforms
Model Transform: Scale, rotate, translate object into scene
(x, y, z, 1.0) world coordinates
View Transform: Orient scene in front of viewer's eye
(x, y, z, 1.0) eye coordinates
Most lighting needs eye space
Projection Transform: Apply perspective and size the frustum
OpenGL needs these
(x, y, z, w) homogeneous clip coordinates
Units normalized such that divide by w leaves visible points between −1.0 to +1.0

Parallel Viewing

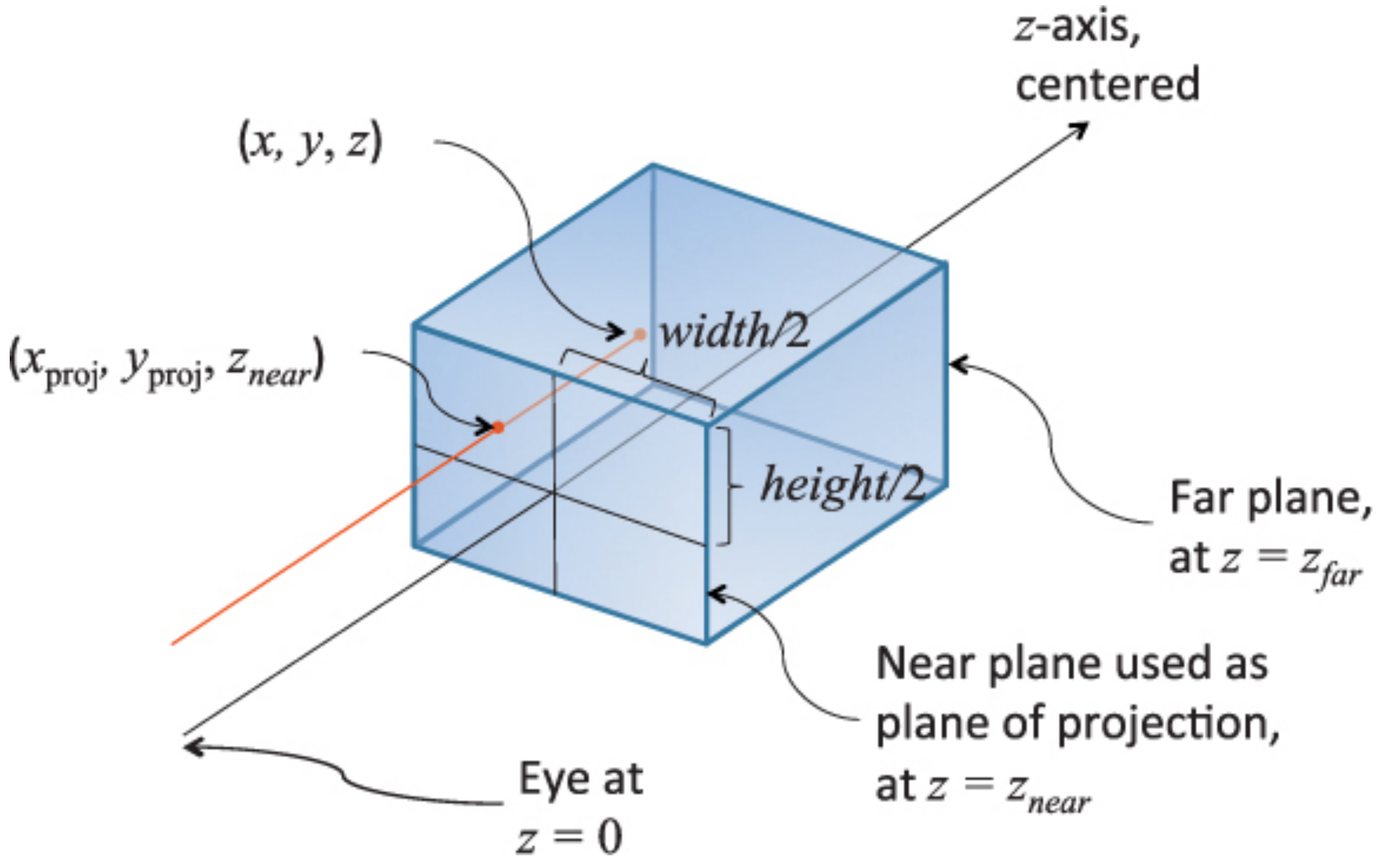
$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

P is defined as long as l ≠ r, t ≠ b, and n ≠ f.

Details


z-axis, centered
(x, y, z)
$(x_{\text{proj}}, y_{\text{proj}}, z_{near})$
$width/2$
$height/2$
Far plane, at $z = z_{far}$
Near plane used as plane of projection, at $z = z_{near}$
Eye at $z = 0$

Symmetric

$$\begin{bmatrix} \frac{1}{width/2} & 0.0 & 0.0 & 0.0 \\ 0.0 & \frac{1}{height/2} & 0.0 & 0.0 \\ 0.0 & 0.0 & -\frac{1}{(z_{far}-z_{near})/2} & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Asymmetric

$$\begin{bmatrix} \frac{1}{(right-left)/2} & 0.0 & 0.0 & -\frac{right+left}{right-left} \\ 0.0 & \frac{1}{(top-bottom)/2} & 0.0 & -\frac{top+bottom}{top-bottom} \\ 0.0 & 0.0 & -\frac{1}{(z_{far}-z_{near})/2} & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$