COSC 4370 – Computer Graphics

Lecture 4

AFFINE Transformations

Chapter 4.6 – 4.10

SUMMARY and NOTES

In this chapter, we have presented two different but ultimately complementary points of view regarding the **mathematics of computer graphics**. One is that **mathematical abstraction of the objects with which we work in computer graphics is necessary** if we are to understand the operations that we carry out in our programs. The other is that **transformations and the techniques for carrying them out, such as the use of homogeneous coordinates are the basis for implementations of graphics systems**.

Our mathematical tools come from the study of **vector analysis** and **linear algebra**.

We pursued a **coordinate-free approach** for two reasons. First, we wanted to show that all the basic concepts of **geometric objects** and of transformations are independent of the ways the latter are represented. Second, as object-oriented languages become more prevalent, application programmers will work directly with the **objects**, instead of with those **objects** representations.

Homogeneous coordinates provided a wonderful example of the power of mathematical abstraction. By going to an abstract mathematical space the **affine space** we were able to find a tool that led directly to efficient software and hardware methods.

Finally, we provided the **set of affine transformations supported in OpenGL** and discussed ways that we could concatenate them to provide all affine transformations.

Camera Analogy

Projection transformations

adjust the lens of the **camera**

Viewing transformations

tripod-define position and orientation of the viewing volume in the world

Modeling transformations

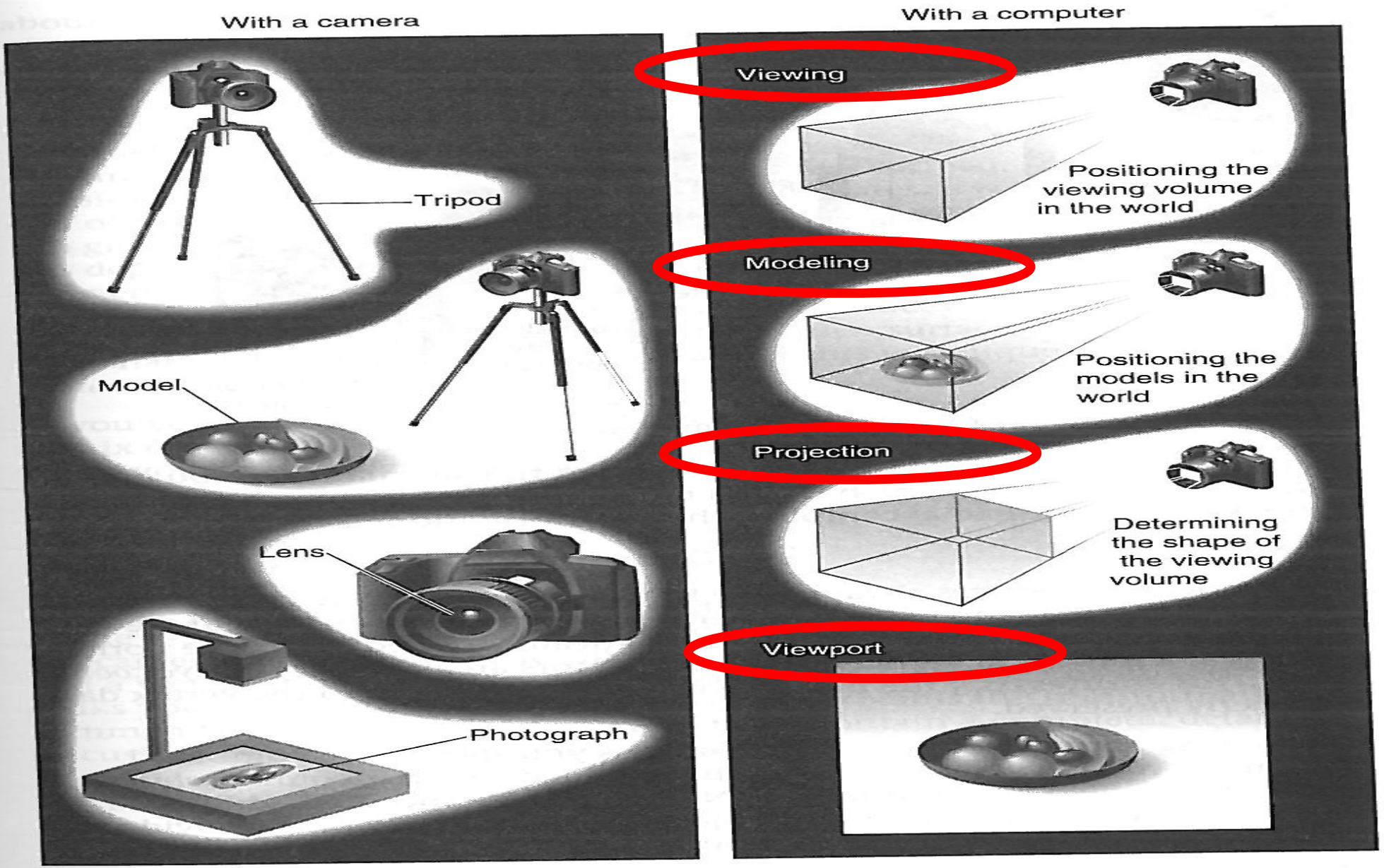
moving the **model** (**Object**)

Viewport transformations

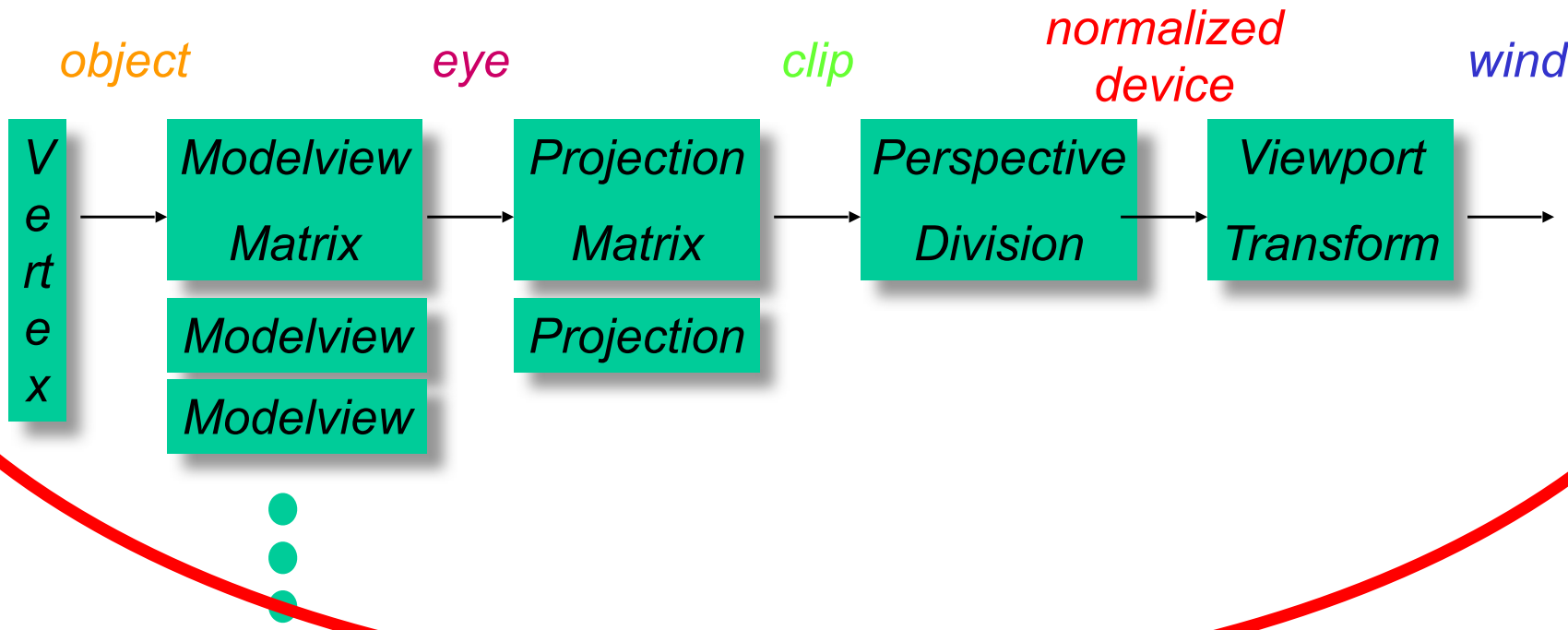
enlarge or reduce the physical photograph

The University of New Mexico

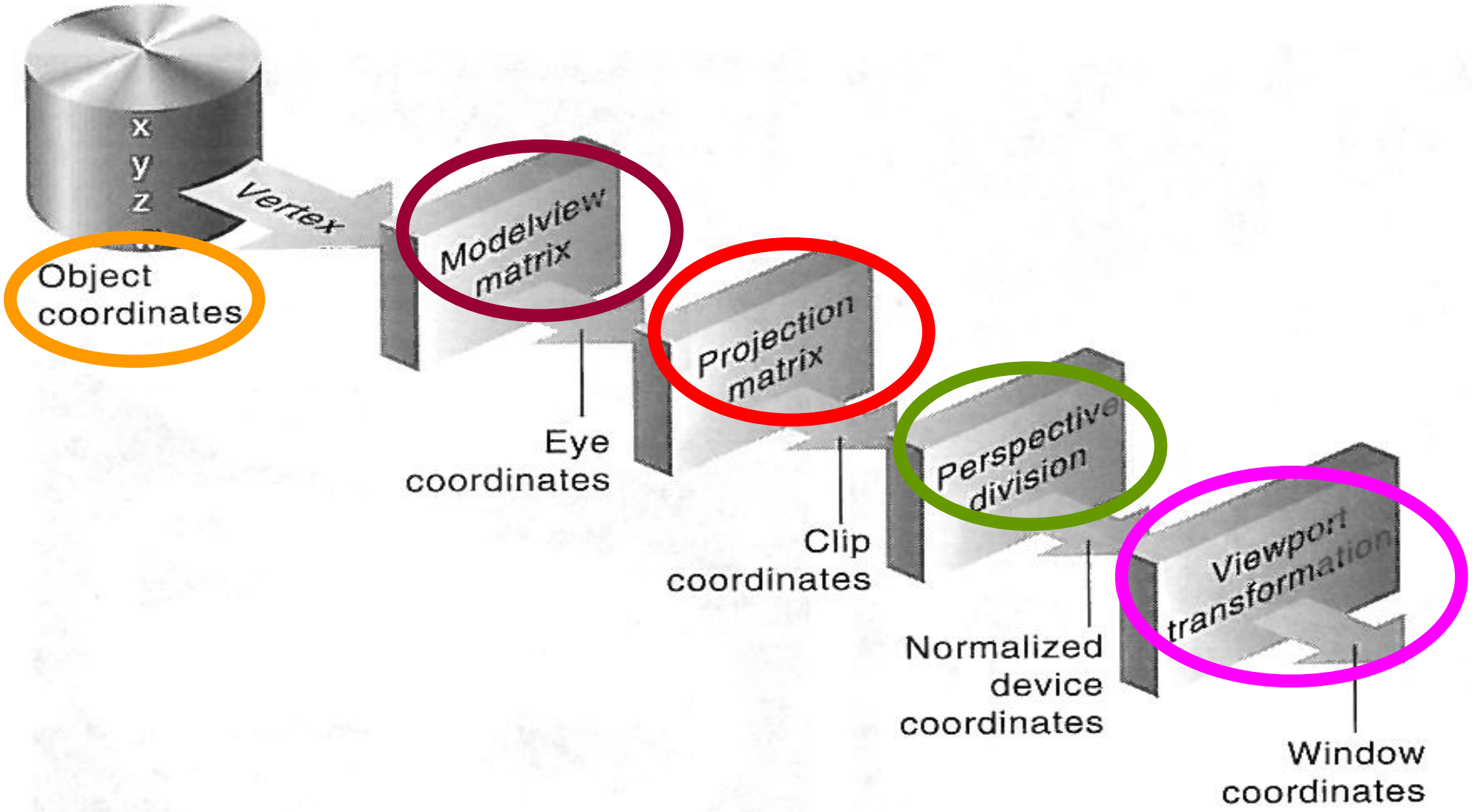
The Camera Analogy



Transformation Pipeline



Stages of Vertex Transformations



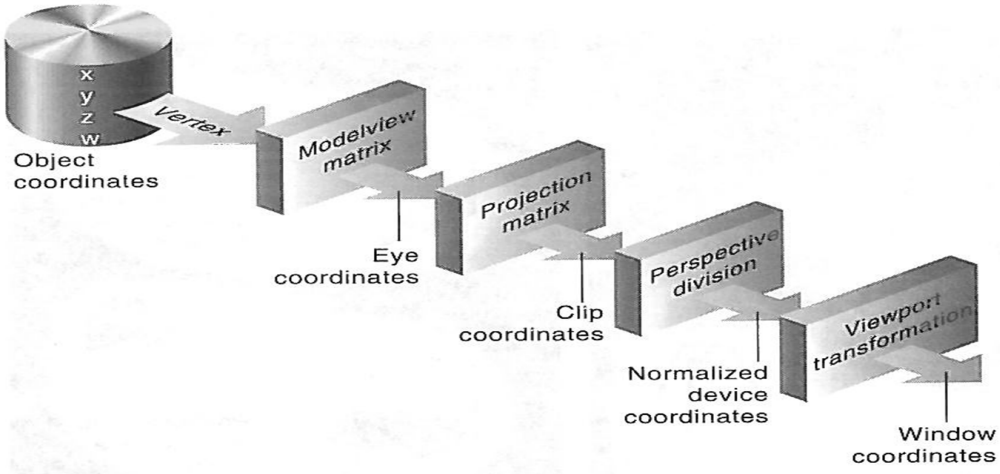
Matrix Operations

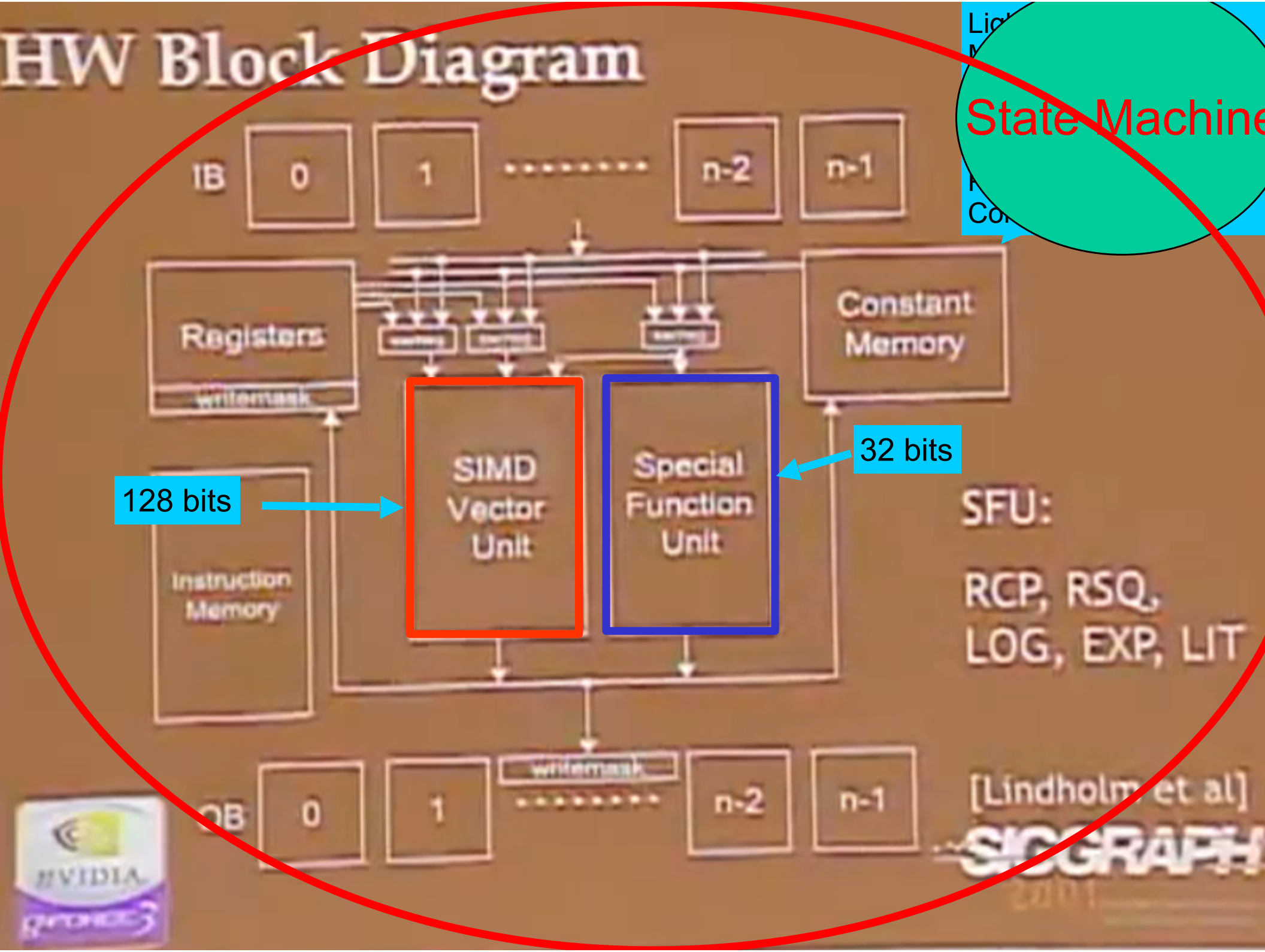
- Specify **C**urren**T** Matrix Stack (**CT**)

```
glMatrixMode(GL_MODELVIEW or GL_PROJECTION)
```

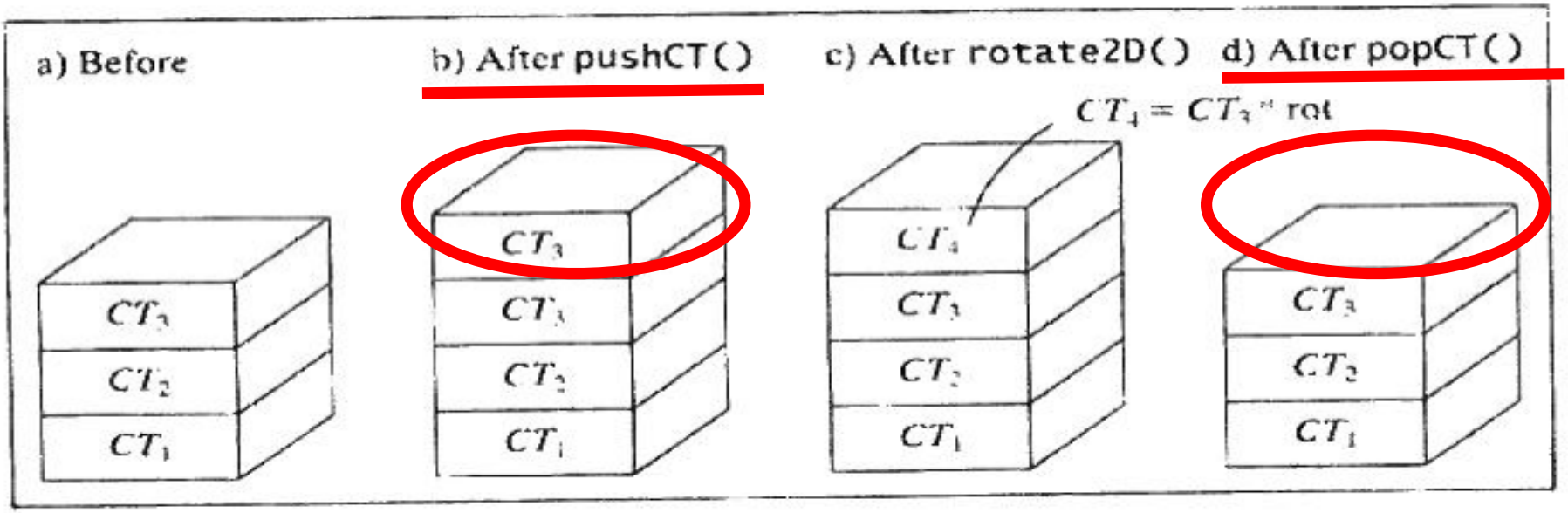
- Other Matrix or **Stack** Operations

```
glLoadIdentity()
glPushMatrix()
glPopMatrix()
```




HW Block Diagram
State Machine
IB
0
1
n-2
n-1
Registers
writemask
Constant Memory
SIMD Vector Unit
Special Function Unit
32 bits
128 bits
Instruction Memory
SFU:
RCP, RSQ,
LOG, EXP, LIT
writemask
OB
0
1
n-2
n-1
[Lindholm et al]
SIGGRAPH
NVIDIA

CT Matrix Stack




```
void pushCT(void)
{
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();                 // push a copy of the top matrix
}
void checkStack(void)
{
  // pop the CT stack only if the stack depth is greater than 1
        if (glGet (GL_MODELVIEW_STACK_DEPTH() = 1) )
                // do something different
        else
                popCT();
}
void popCT(void)
{
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix(); // pop the top matrix from the stack
}
```

Transformation hierarchy example

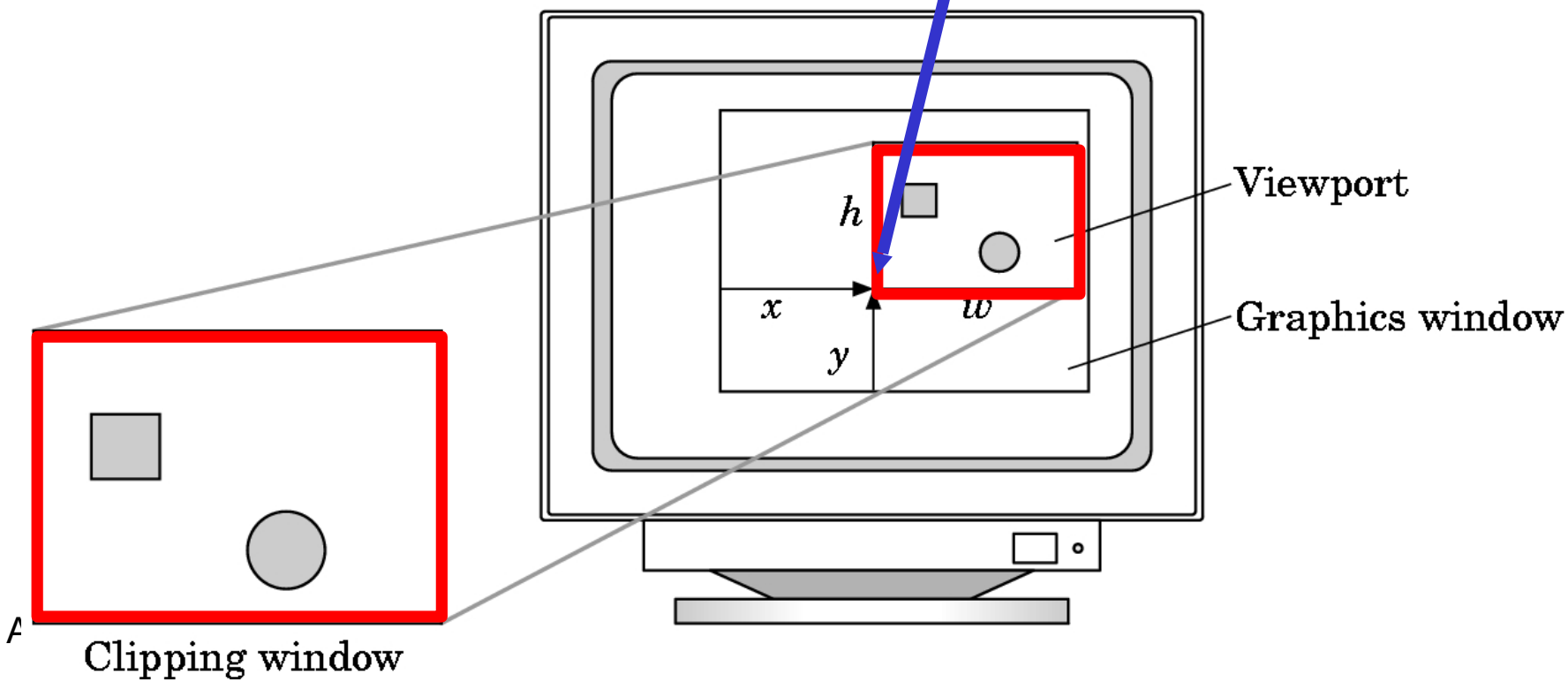
```
glLoadIdentity()
Transformation T1
Draw something (T1)
glPushMatrix()
        Transformation T2
        glPushMatrix()
                Transformation T3
                Draw something (T1*T2*T3)
                glLoadIdentity()
                Draw something (I)
      glPopMatrix()
         Draw something (T1*T2)
glPopMatrix()
Draw something (T1)
```

Viewport Transformation

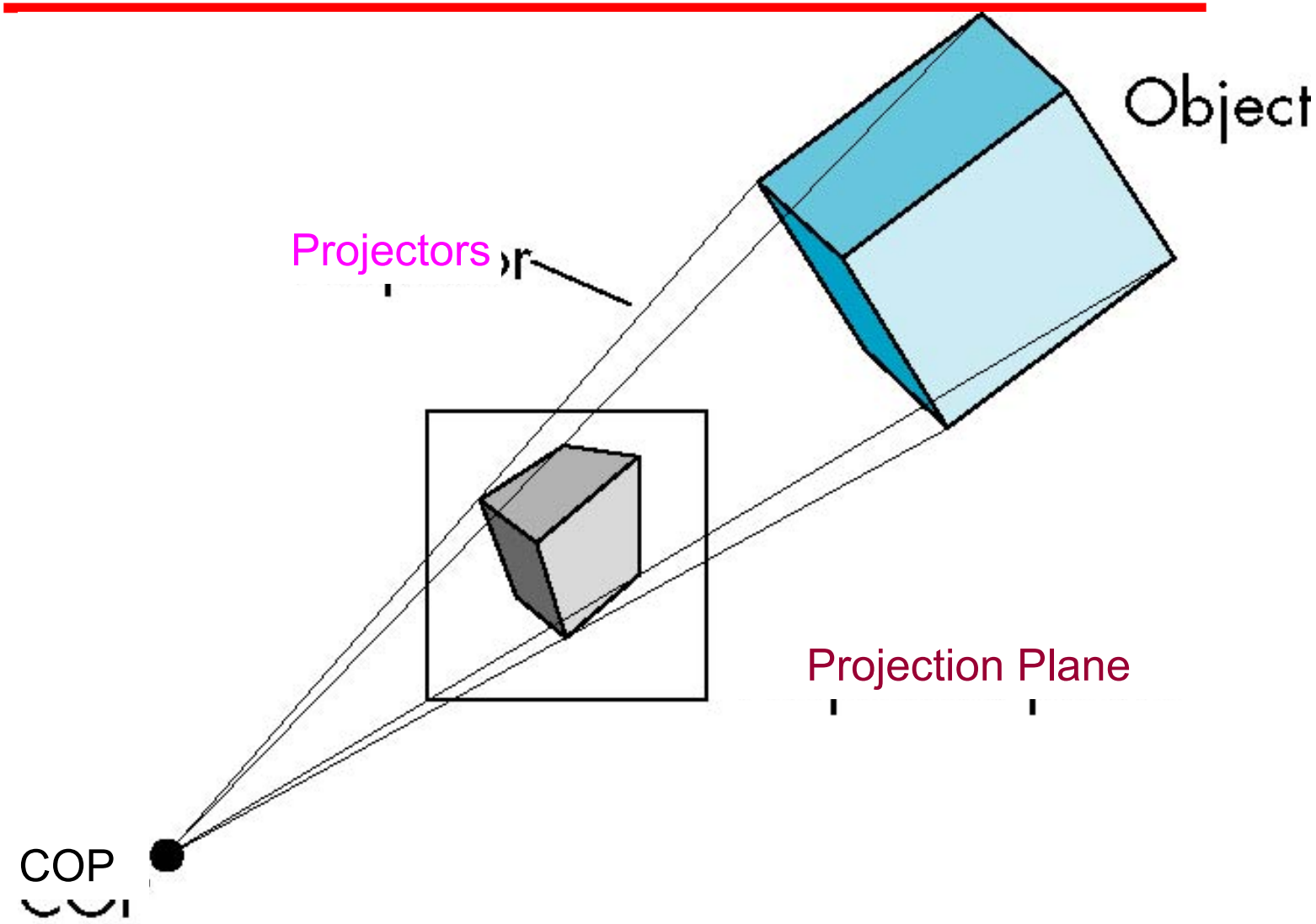
glViewport(x, y, width, height);

Usually same as window size

Viewport aspect ratio should be same as projection transformation or resulting image may be distorted

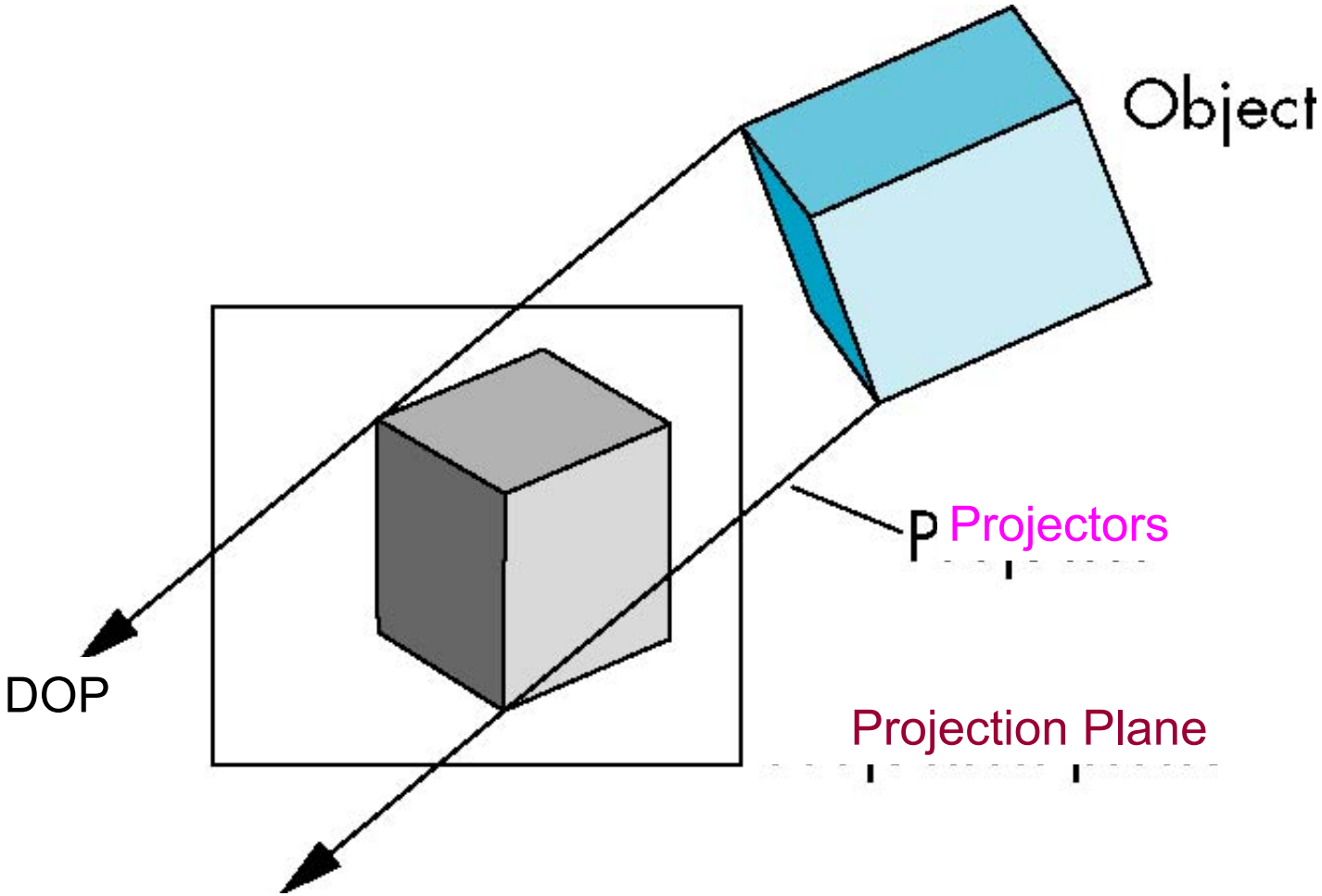


Perspective Projection



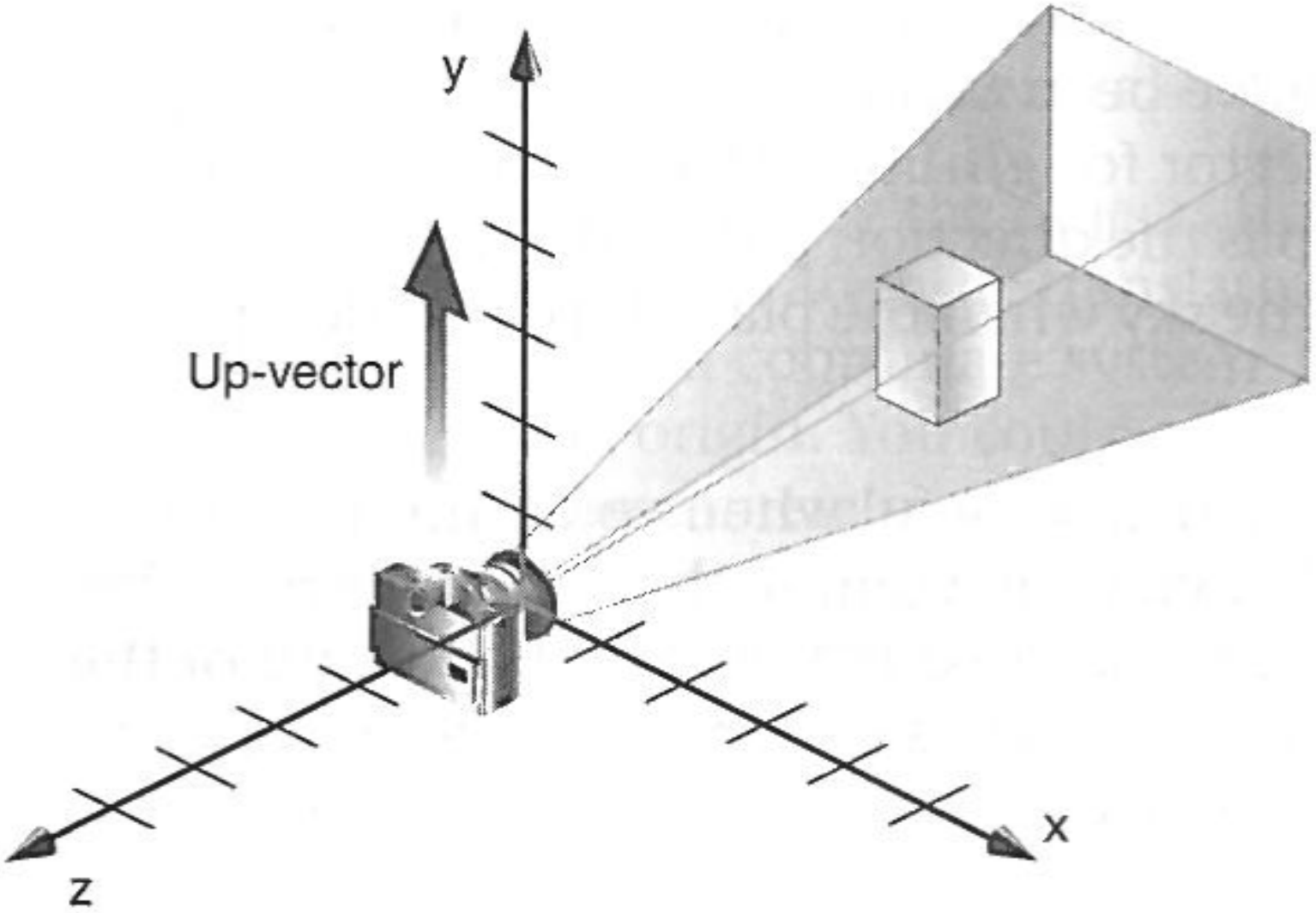
Angel: In http://www.cosc.canterbury.ac.nz/mukundan/cogr/Projns.html#

Parallel Projection



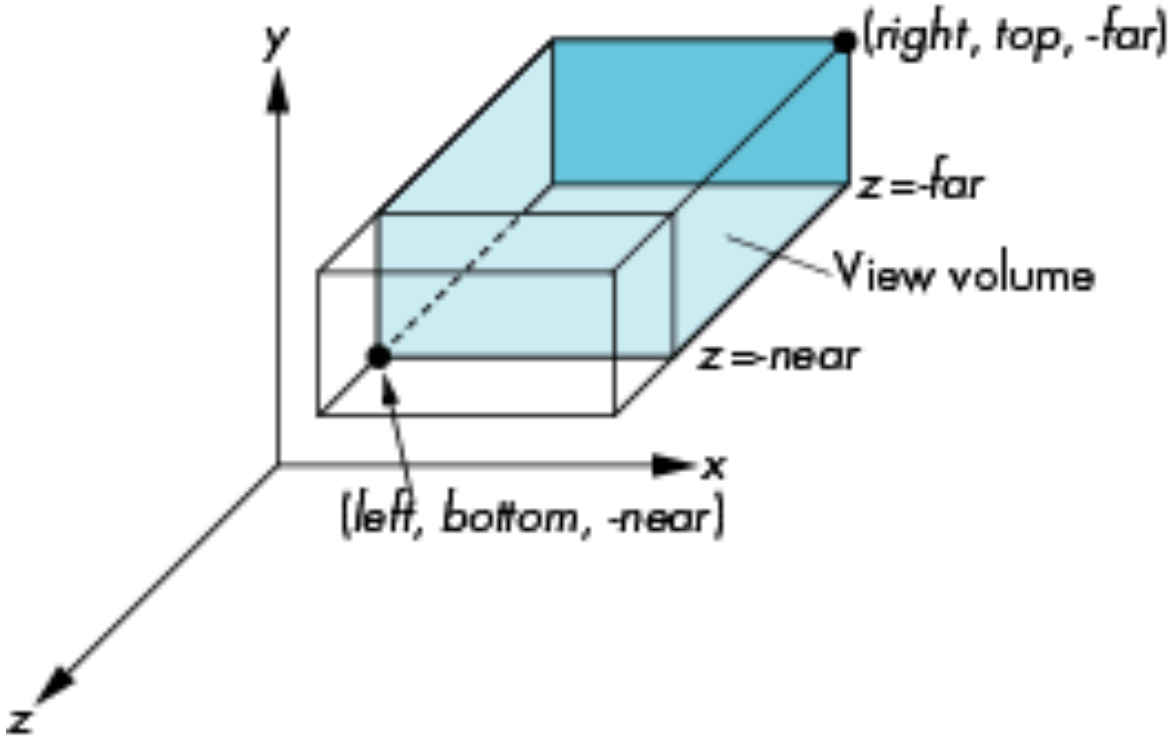
Angel: In http://www.cosc.canterbury.ac.nz/mukundan/cogr/Projns.html#

Default Camera Position



OpenGL Orthogonal Viewing

`glOrtho(left,right,bottom,top,near,far)`



`near` and `far` measured from camera

Default OpenGL Camera

If we do not specify a viewing volume, OpenGL uses its default, a 2 X 2 X 2 cube, with the origin in the center.

In terms of our two- dimensional plane, the bottom- left corner is at (- 1.0, - 1.0), and the upper- right corner is at (1.0, 1.0).

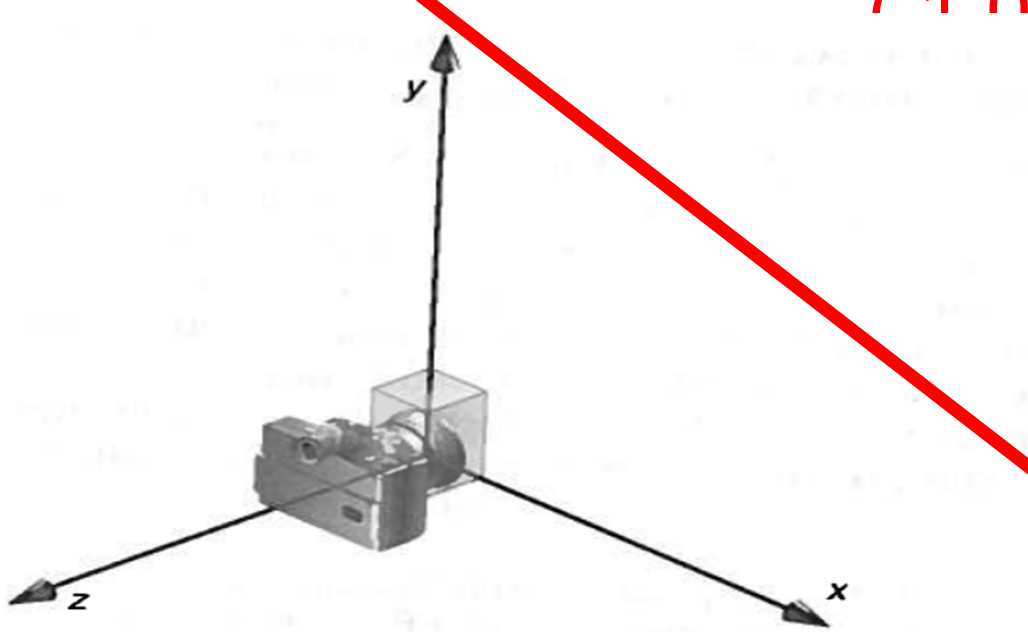
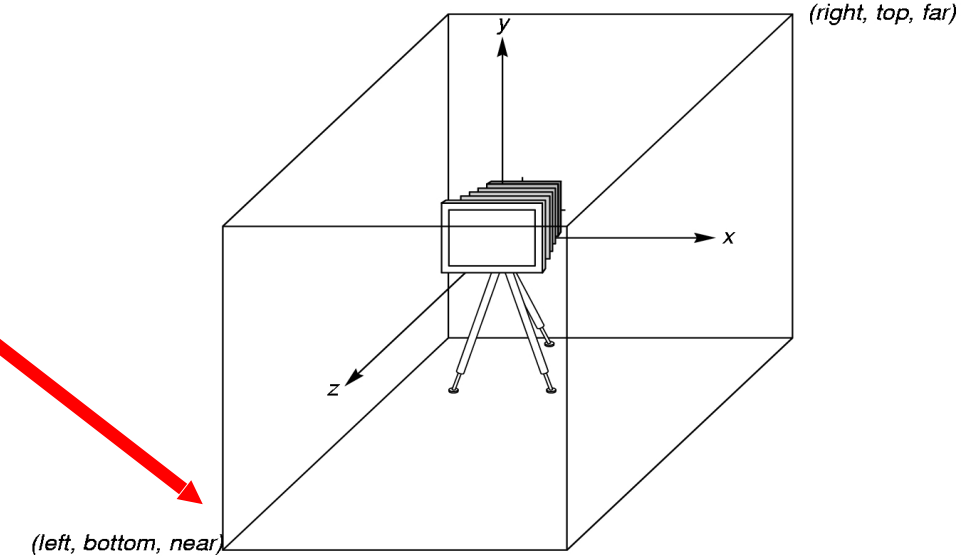


Figure 3-9 Object and Viewpoint at the Origin



OpenGL Camera

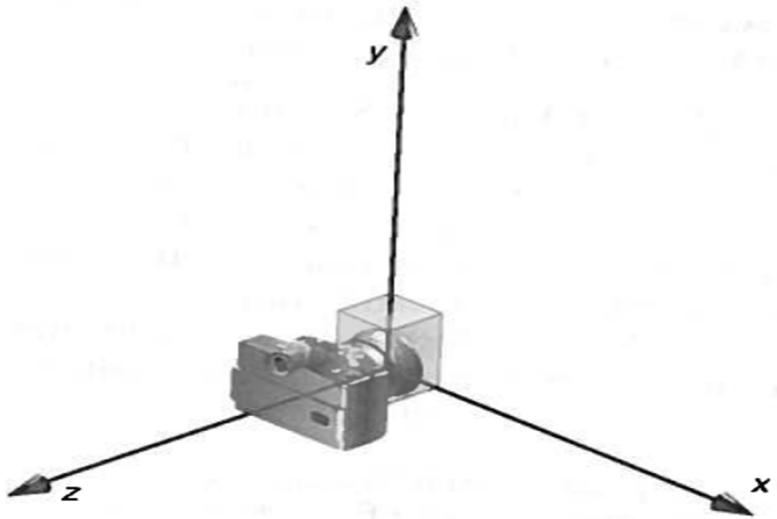


Figure 3-9 Object and Viewpoint at the Origin

```
glTranslatef(0.0, 0.0, -5.0);
```

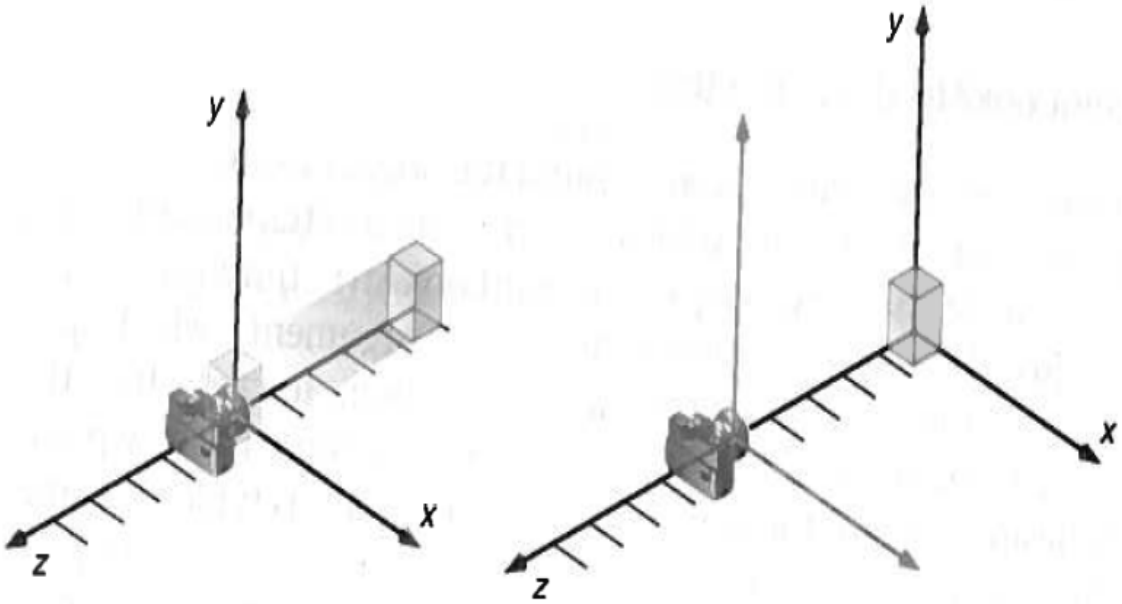
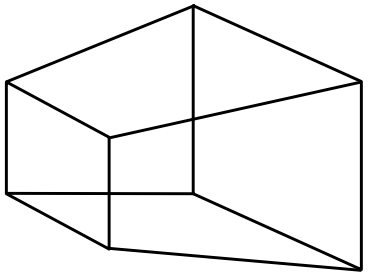


Figure 3-10 Separating the Viewpoint and the Object

Projection Transformation

adjust the lens of the camera

Shape of viewing frustum



Perspective projection

```
gluPerspective( fovy, aspect, zNear, zFar)
glFrustum( left, right, bottom, top, zNear, zFar)
```

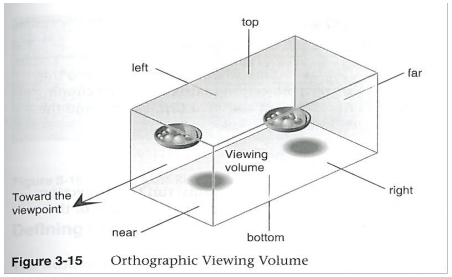


Figure 3-15 Orthographic Viewing Volume

Orthographic parallel projection

```
glOrtho( left, right, bottom, top, zNear, zFar)
gluOrtho2D( left, right, bottom, top )
```

calls `glOrtho` with z values near zero

Viewing Transformations

tripod-define position and orientation of the viewing volume in the world

Position the **camera**/eye in the **scene**

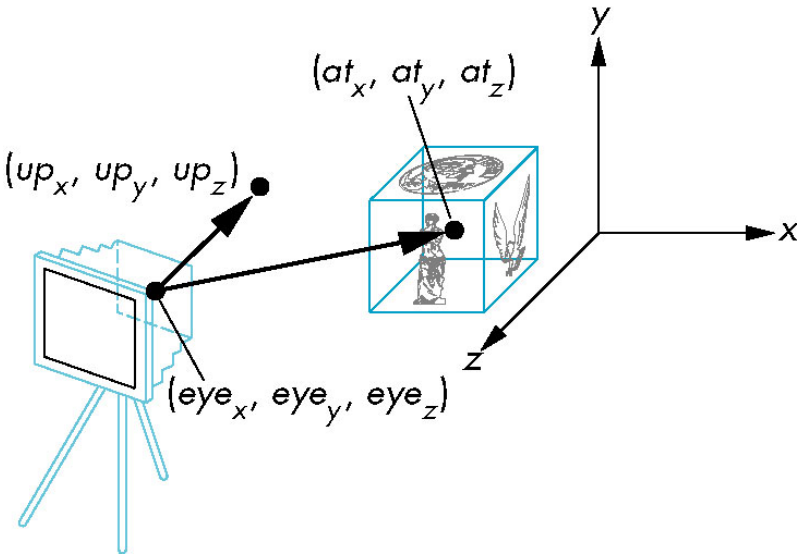
place the tripod down; aim **camera**

To "**fly through**" a **scene**

change **viewing transformation** and **redraw scene**

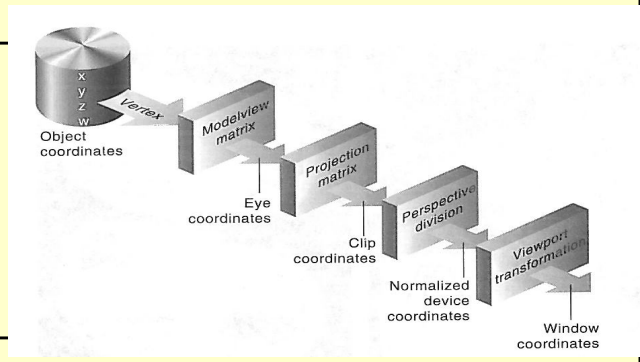
```
gluLookAt( eyex, eyey, eyez,
           aimx, aimy, aimz,
           upx, upy, upz )
```

(**up vector** determines unique orientation)



OpenGL functions for setting up transformations

modelling transformation *modelview matrix*	glTranslatef() glRotatef() glScalef()
viewing transformation *modelview matrix*	gluLookAt()
projection transformation *projection matrix*	glFrustum() gluPerspective() glOrtho() gluOrtho2D()
viewing transformation	glViewport()



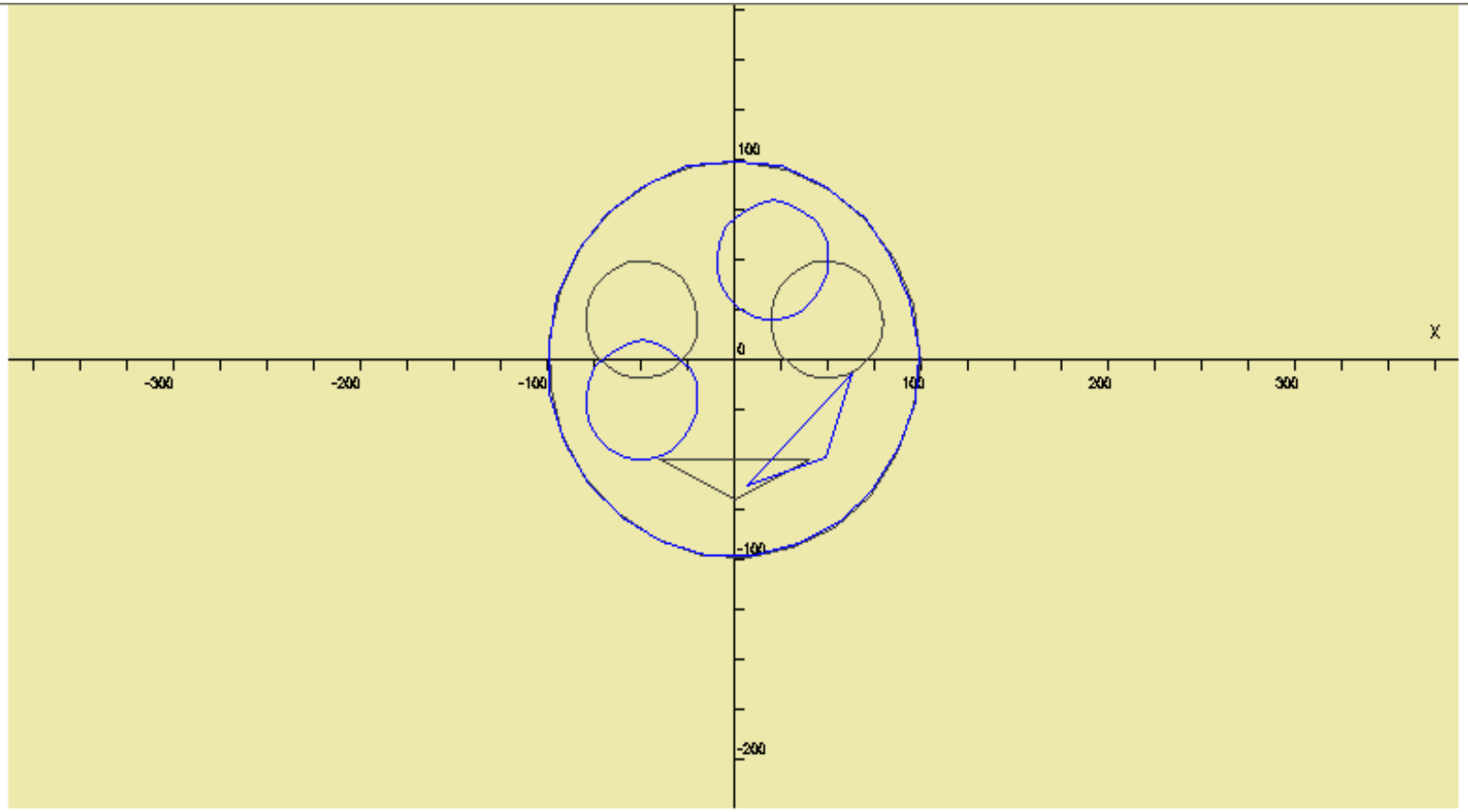
Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive **homogeneous coordinate transformation matrices**
- Learn to build **arbitrary transformation matrices from simple transformations**

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive **homogeneous coordinate transformation matrices**
- Learn to build **arbitrary transformation matrices from simple transformations**

2D Transforms Applet



rotate (angle in degrees) | translate (by (x,y) pixels)

45 | 0.0, 0.0

Face | House | Transform 1 then 2 | Transform 2 then 1 | Reset | Store current transform | Show original (grey) | Show stored (red)

http://www.cs.princeton.edu/~min/cs426/classes/transform.html

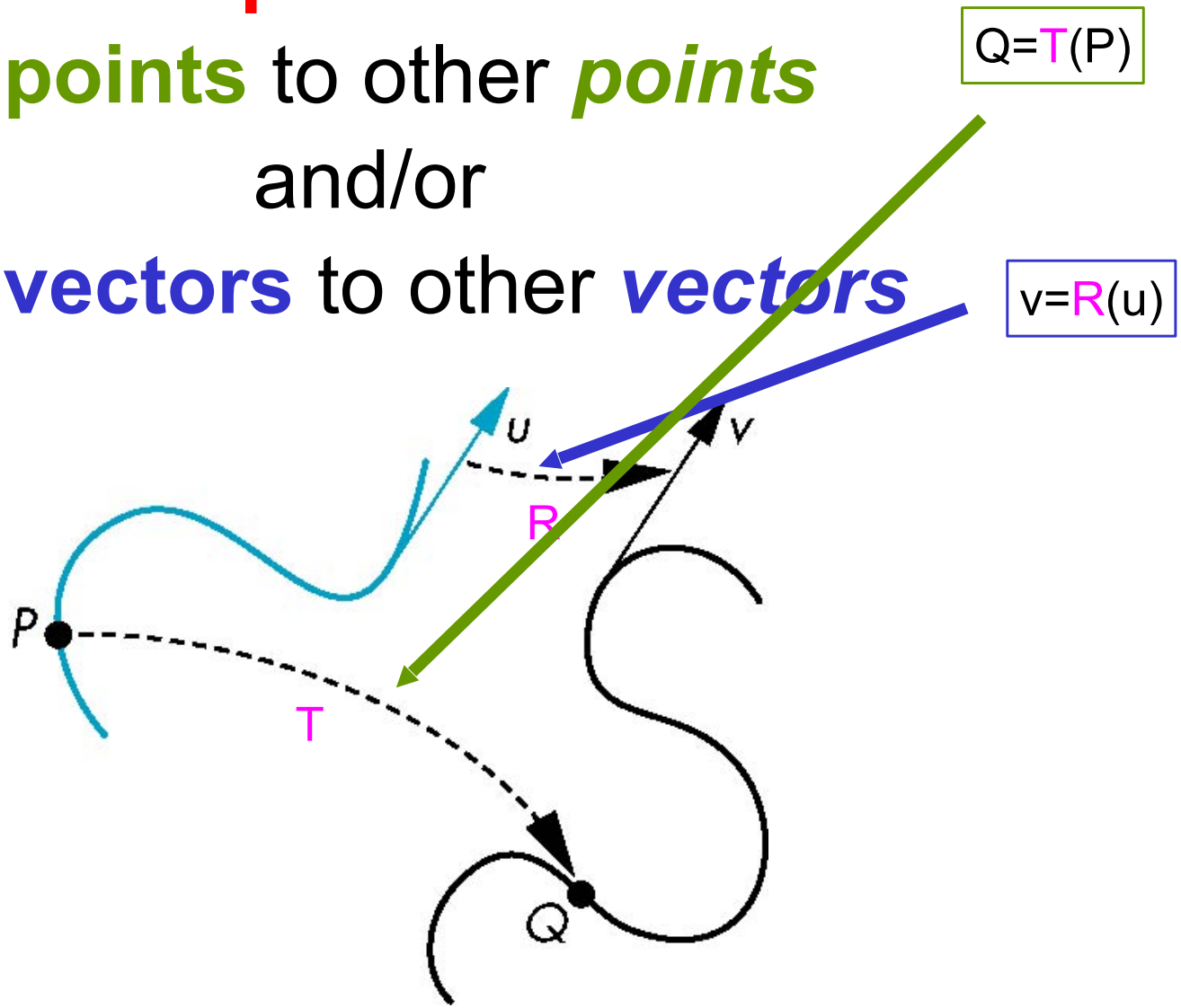
General Transformations

A transformation **maps**

points to other ***points***

and/or

vectors to other ***vectors***



Affine Transformations

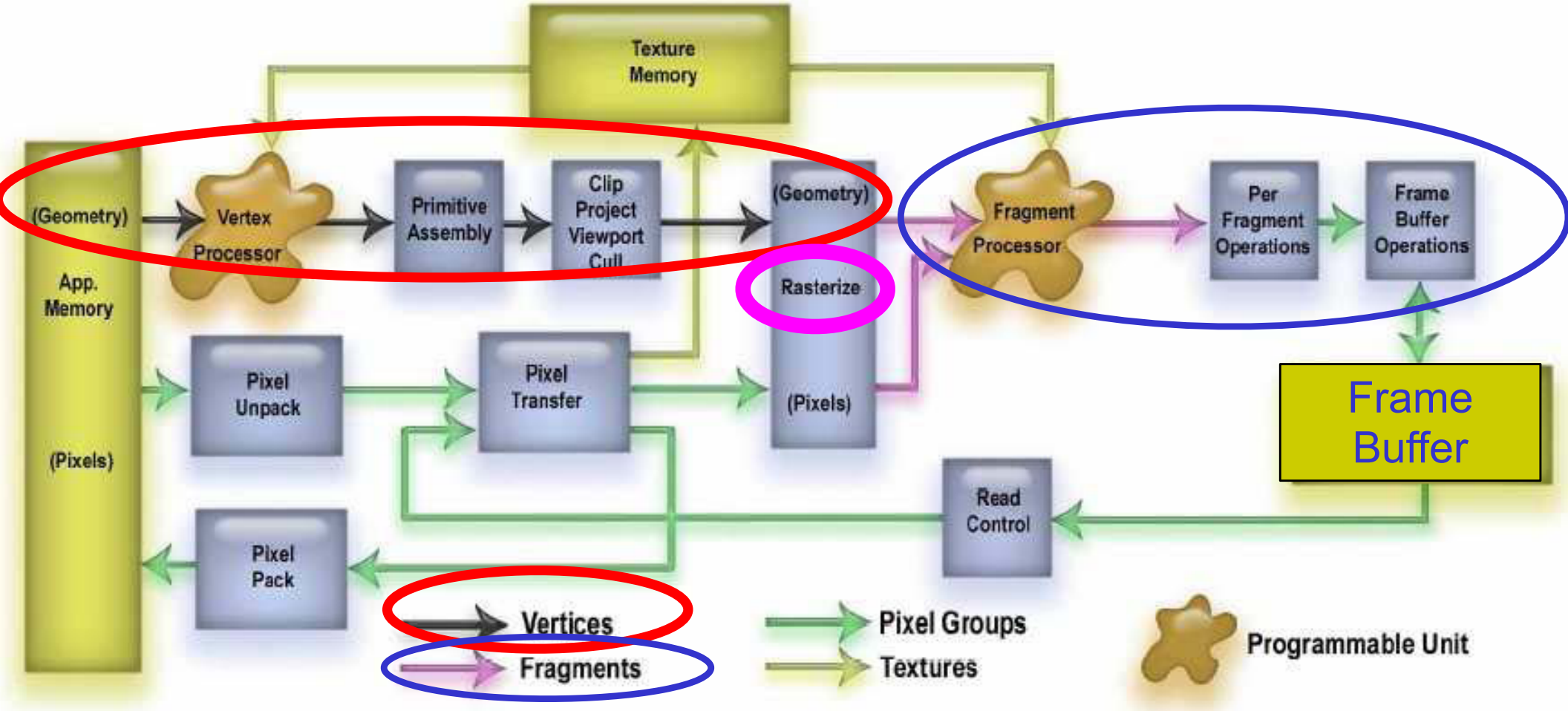
Line preserving

Characteristic of **many physically important transformations**

- Rigid body transformations: Rotation, Translation
- Scaling, Shear

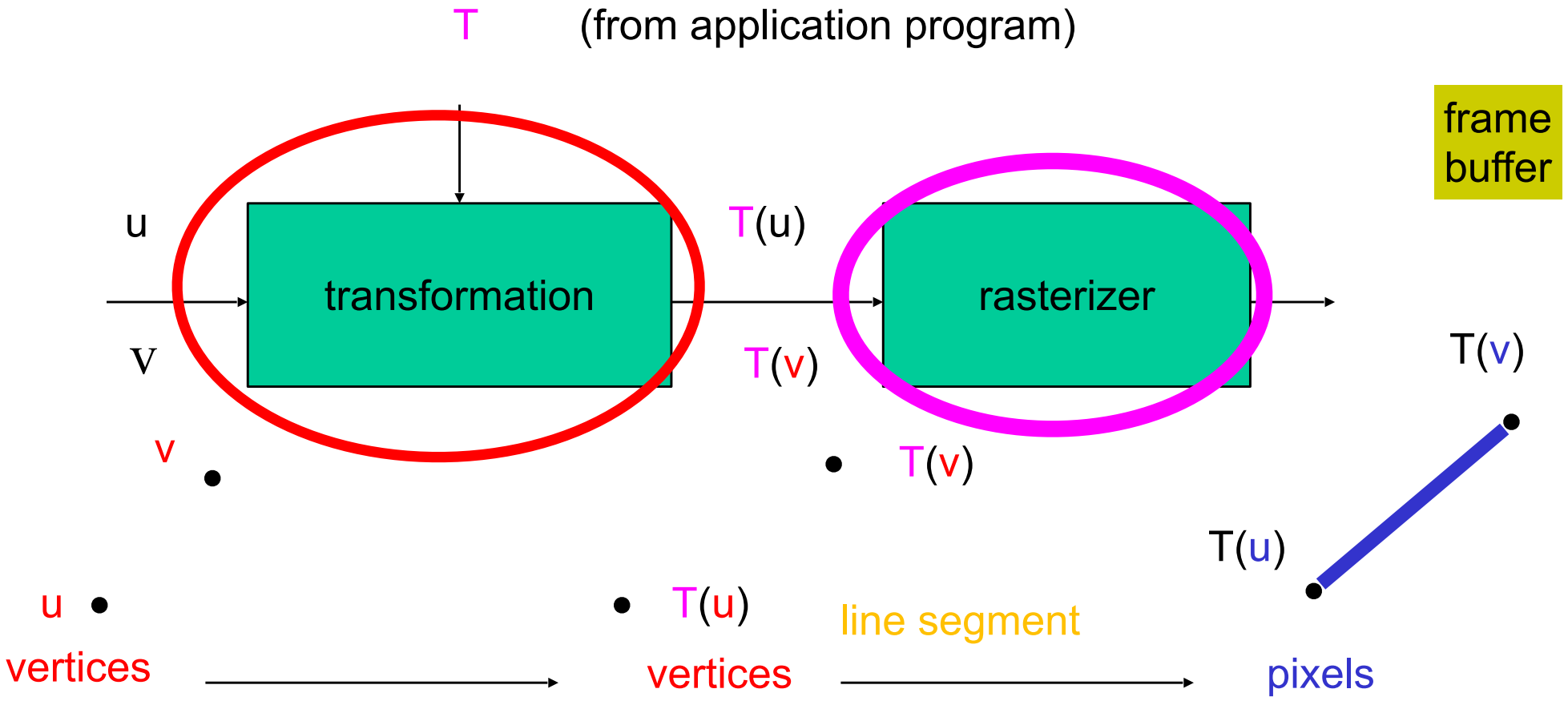
Importance in graphics is that we **need only transform** endpoints of line segments and **let implementation draw line segment between the transformed endpoints**

OpenGL Graphics Pipeline



Pipeline Implementation

We need only to transform the homogeneous-coordinate representation of the endpoints of a line segment to determine completely a transformed line. Thus, we can implement our graphics systems as a pipeline that passes endpoints through affine transformation units, and generates the interior points at the rasterization stage.



Notations

Will be working with both **coordinate-free representations** of transformations and **representations within a particular frame**

P,Q, R: points in an **Affine Space**

u, **v**, **w**: vectors in an **Affine Space**

α, β, γ: scalars

p, **q**, **r**: representations of points

-array of **4 scalars in homogeneous coordinates**

u, **v**, **w**: representations of vectors

-array of **4 scalars in homogeneous coordinates**

Notations

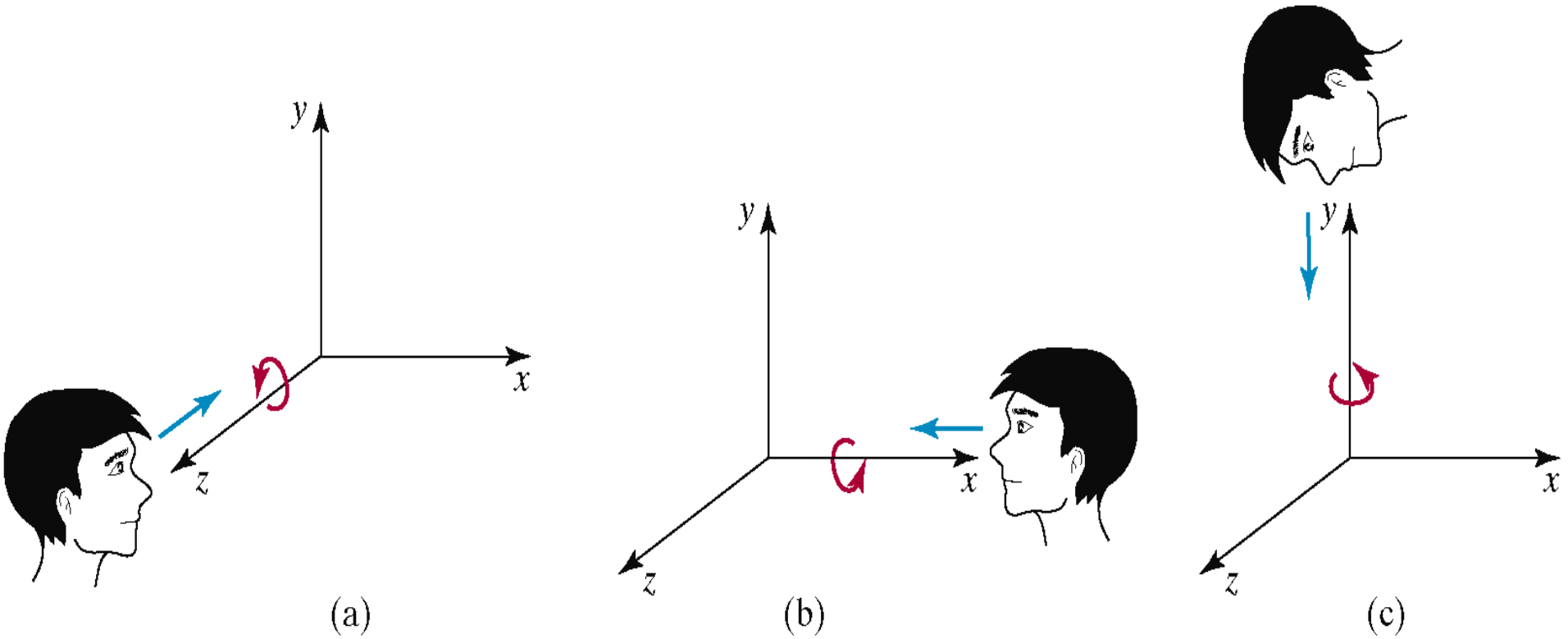


Figure 5-36

Positive rotations about a coordinate axis are counterclockwise, when looking along the positive half of the axis toward the origin.

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive **homogeneous coordinate transformation matrices**
- Learn to build **arbitrary transformation matrices from simple transformations**

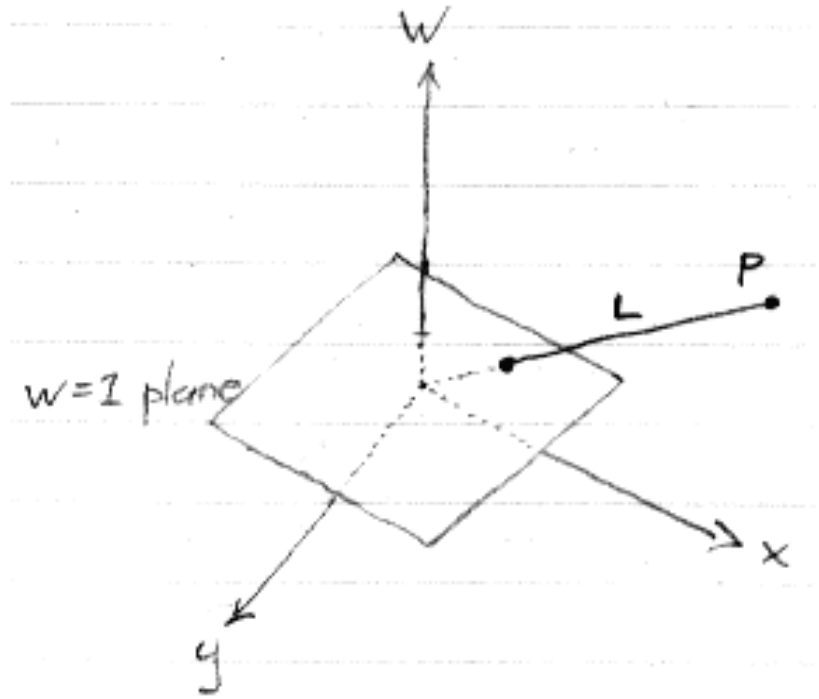
Homogeneous Coordinates

- Homogeneous coordinates to the rescue
  2D cartesian (x,y) --> 3D **homogeneous** (x,y,**w**)

homogeneous cartesian

$$(x, y, w) \xrightarrow{/\ w} \left(\frac{x}{w}, \frac{y}{w}\right)$$

Homogeneous coordinates



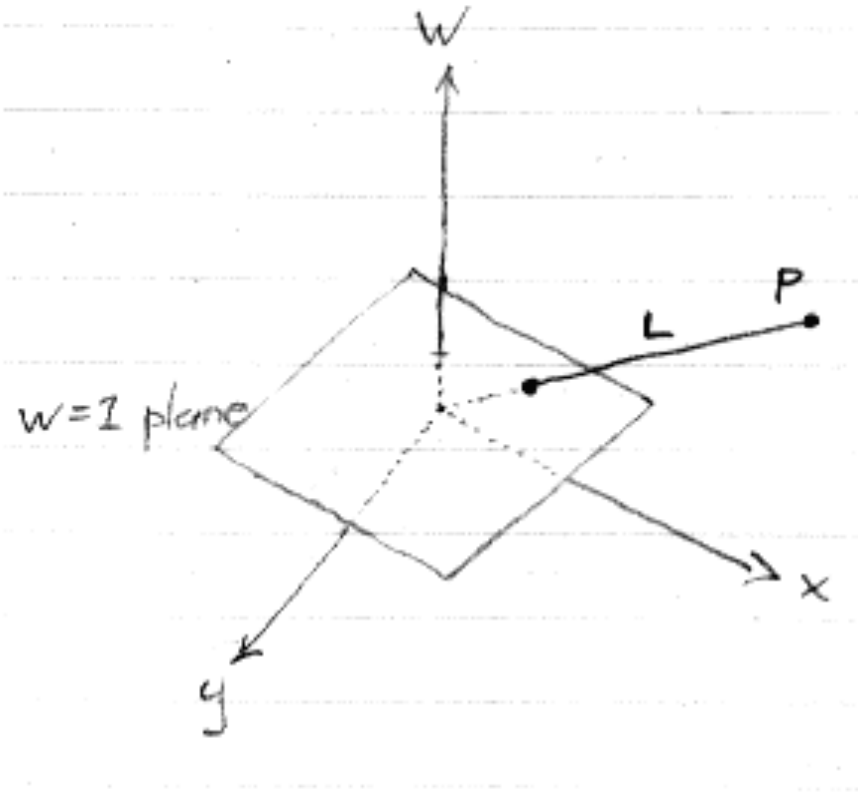
- point in 2D cartesian + weight w =

  point P in 3D homogeneous coordinates

- multiples of (x,y,w)
  - represent same point in 2D cartesian
  - a line L in 3D homogeneous

- homogenize a point in 3D:
  - divide by w to get (x/w, y/w, 1)
  - projects point onto w=1 plane

Homogeneous coordinates



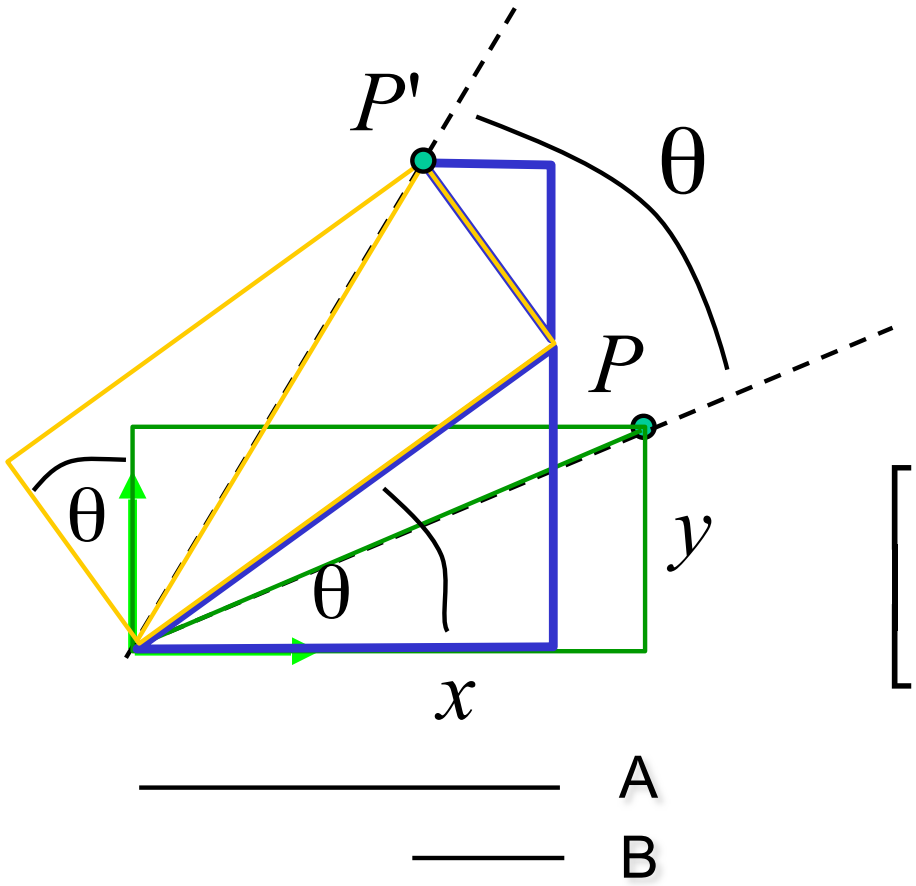
- w=0 denotes point at infinity
  - think of as direction
  - cannot be homogenized
  - lies on x-y plane
- (0,0,0) is not allowed

The University of New Mexico

Transformations

- Rotation

$$Rotate(z,\theta)$$



$$x' = x\cos\theta - y\sin\theta$$

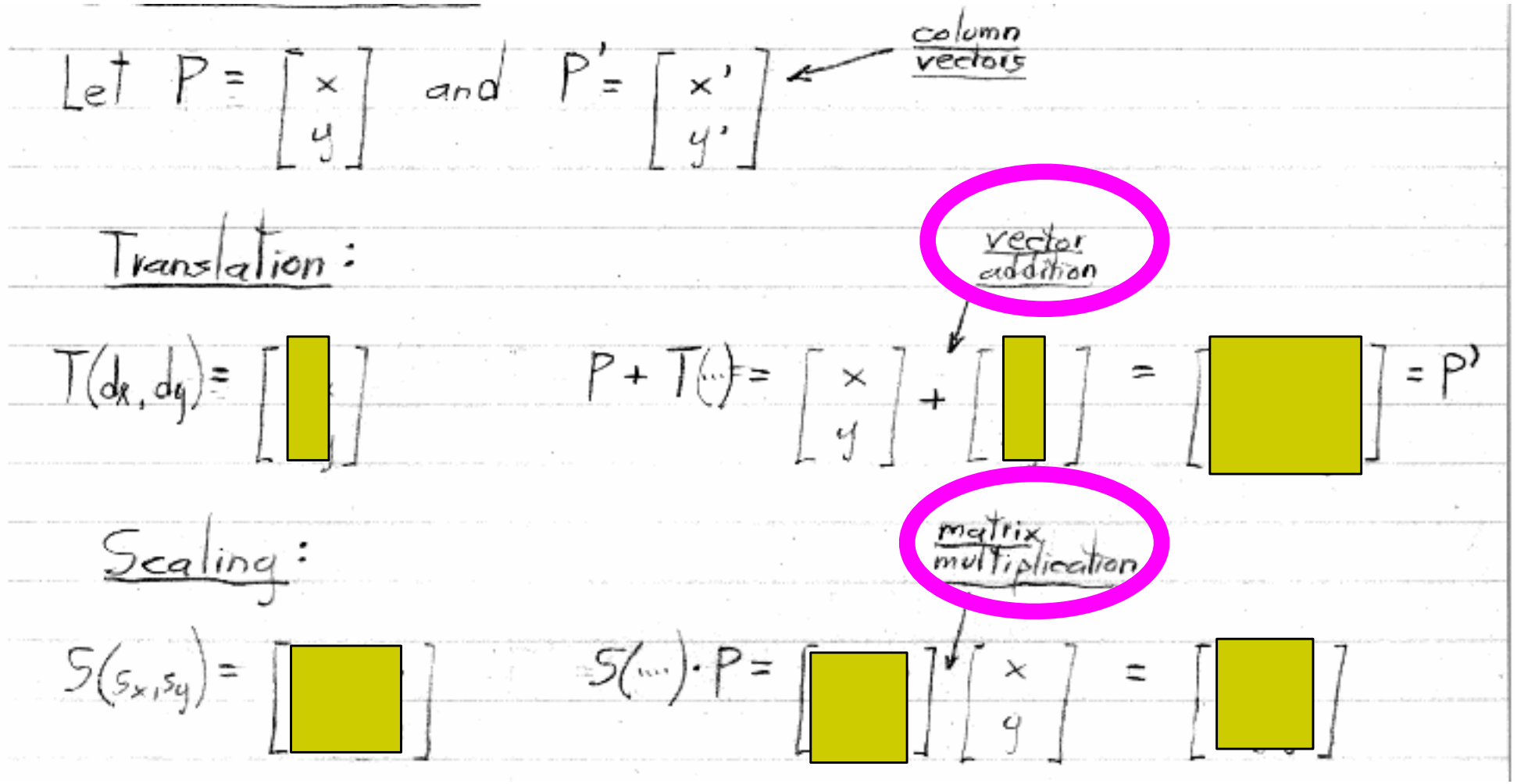
$$y' = x\sin\theta + y\cos\theta$$

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

P' P

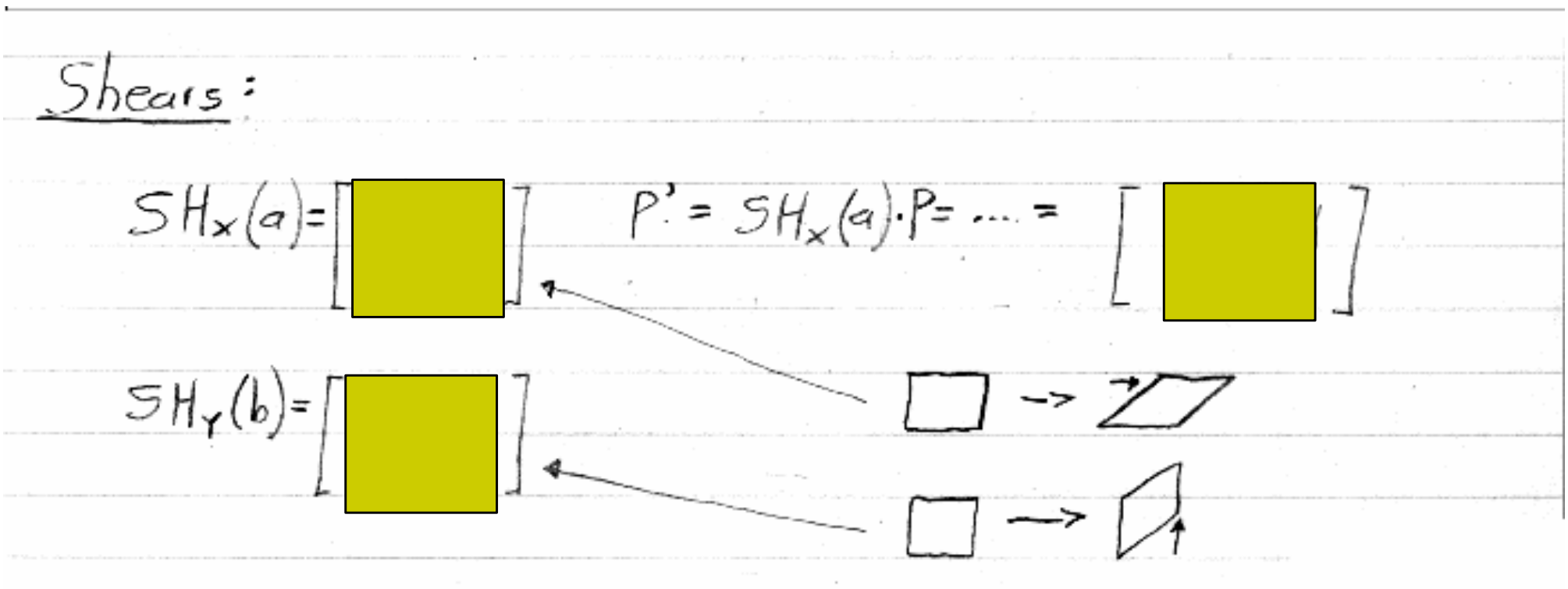
$$P_x' = A - B = x\cos\theta - y\sin\theta$$

2D Transformations



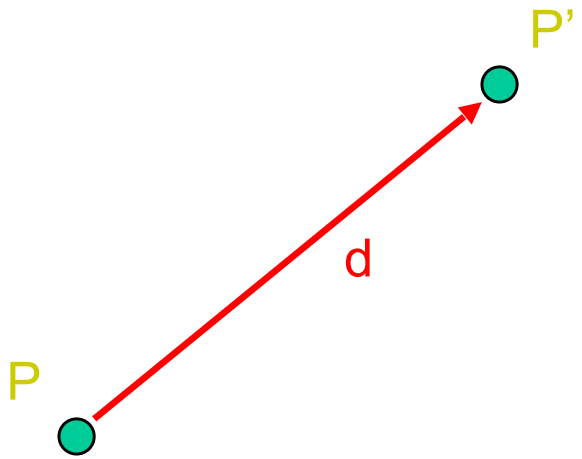
The University of New Mexico

2D transformations



Translation

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



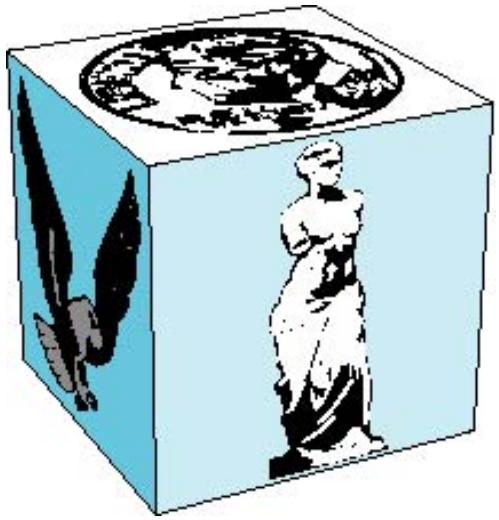
- Displacement determined by a vector **d**
  - Three degrees of freedom
  - P'=P+**d**

http://www.shodor.org/interactivate/activities/Transmographer/
http://www.shodor.org/interactivate/activities/TransmographerTwo/

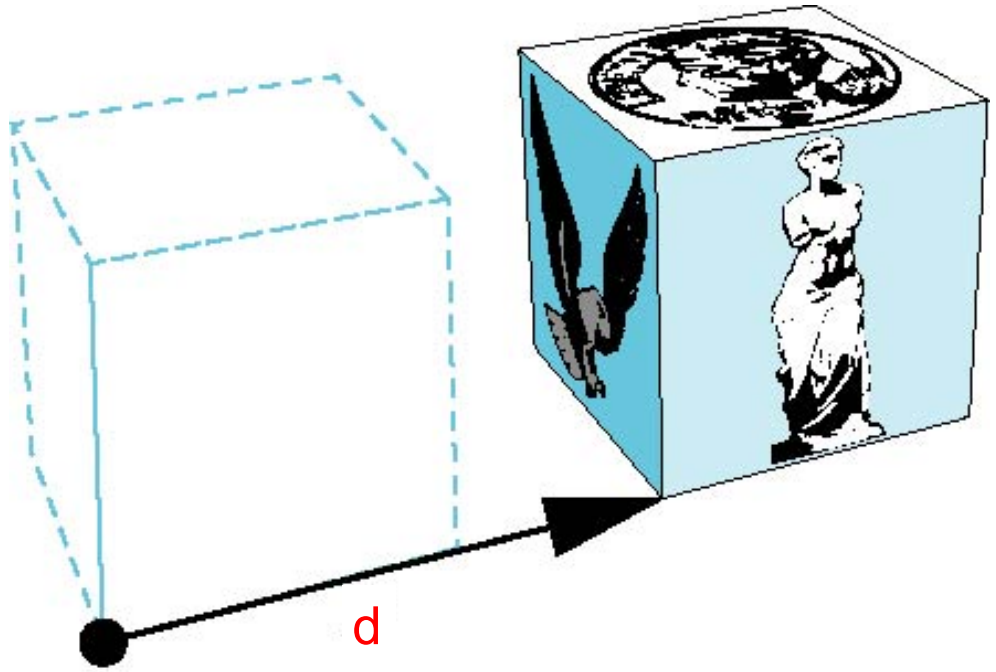
The University of New Mexico

How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually **only one way**



object



translation: every point displaced by same vector d

Translation Using Representations

Using the **homogeneous coordinate representation** in some frame

$p=[\ x \quad y \quad z \quad 1]^T$

$p'=[x' \ y' \ z' \ 1]^T$

$\mathbf{d}=[\mathbf{dx}\ \mathbf{dy}\ \mathbf{dz}\ \mathbf{0}]^T$

Hence $p' = p + \mathbf{d}$ or

$x' = x + \mathbf{d_x}$

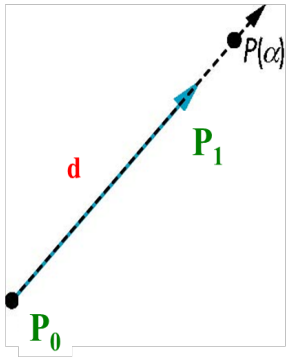
$y' = y + \mathbf{d_y}$

$z' = z + \mathbf{d_z}$

- Consider all **Points** of the form

$P(\alpha) = \mathbf{P_0} + \alpha\ \mathbf{d}$

- Set of all **Points** that pass through $\mathbf{P_0}$ in the direction of the vector $\mathbf{d}$



$\mathbf{P_0} = (x_0, y_0)$

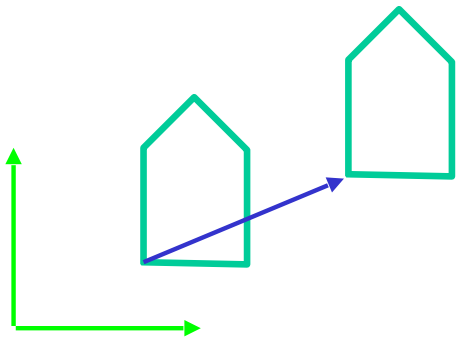
$$P(\alpha) = \mathbf{P_0} + \alpha\ \mathbf{d}$$
$$= \mathbf{P_0} + \alpha\ (\mathbf{P_1} - \mathbf{P_0})$$
$$= \alpha\ \mathbf{P_1} + (1 - \alpha)\ \mathbf{P_0}$$

note that this expression is in four dimensions and expresses point = point + vector

Translation

translate(a,b,c)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & & & a \\ & 1 & & b \\ & & 1 & c \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



glTranslatef(a,b,c);
glTranslated(a,b,c);

Translation Matrix

Express translation using a **4 x 4** matrix **T** in **homogeneous coordinates**

p'=Tp where

$$\mathbf{T} = \mathbf{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This form is better for implementation because **all affine transformations** can be expressed this way and **multiple transformations** can be **concatenated together**

The University of New Mexico

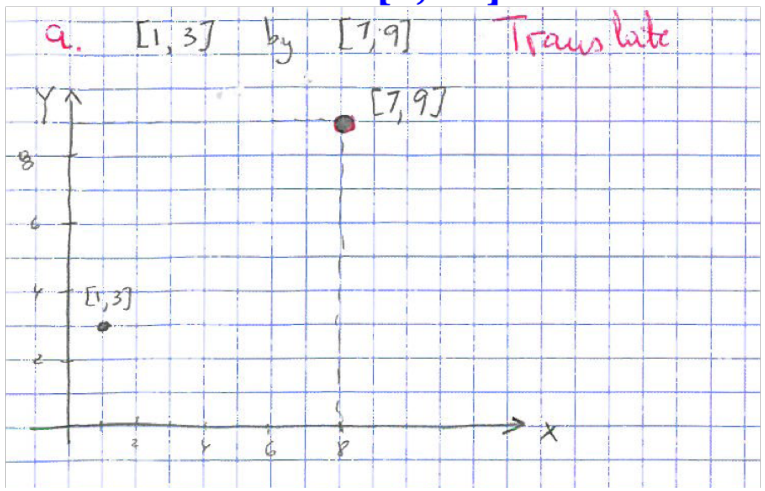
Translation Matrix- Example

Translate [1,3] by [7,9]

1. Write the Translation Matrix using **homogeneous coordinates** (**hint: 3 X 3**)

$$\begin{bmatrix} 1 & 0 & 7 \\ 0 & 1 & 9 \\ 0 & 0 & 1 \end{bmatrix}.$$

[8, 12]



2. Premultiply the Translation Matrix with point [1,3] and find it's new location

CLASS PARTITICIPATION a!

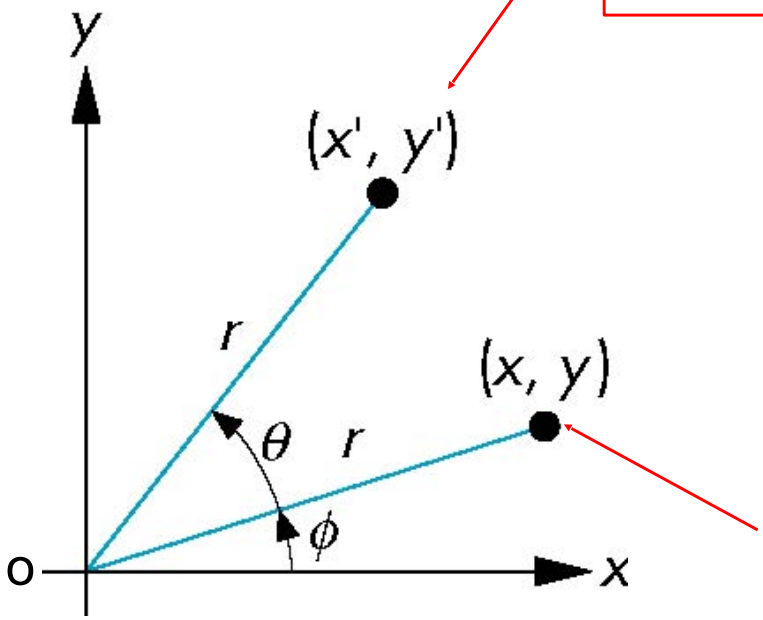
The University of New Mexico

Rotation (2D)
fixed point of origin

Consider rotation about the origin by θ degrees

radius stays the same, angle increases by θ

$x' = r \cos(\phi + \theta)$
$y' = r \sin(\phi + \theta)$



$x' = x \cos\theta - y \sin\theta$
$y' = x \sin\theta + y \cos\theta$

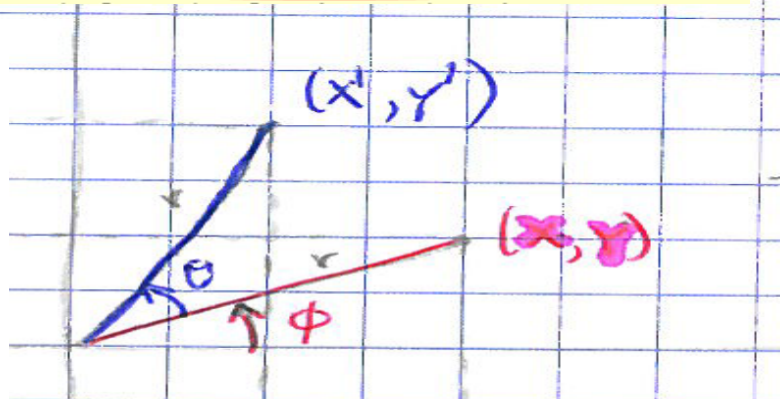
$x = r \cos\phi$
$y = r \sin\phi$

http://www.cosc.canterbury.ac.nz/mukundan/cogr/Rot2D.html

Rotation (2D)

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

$$\sin(s+t) = \sin s \cos t + \cos s \sin t$$
$$\cos(s+t) = \cos s \cos t - \sin s \sin t$$



$$r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$r\sin(\phi+\theta) = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$

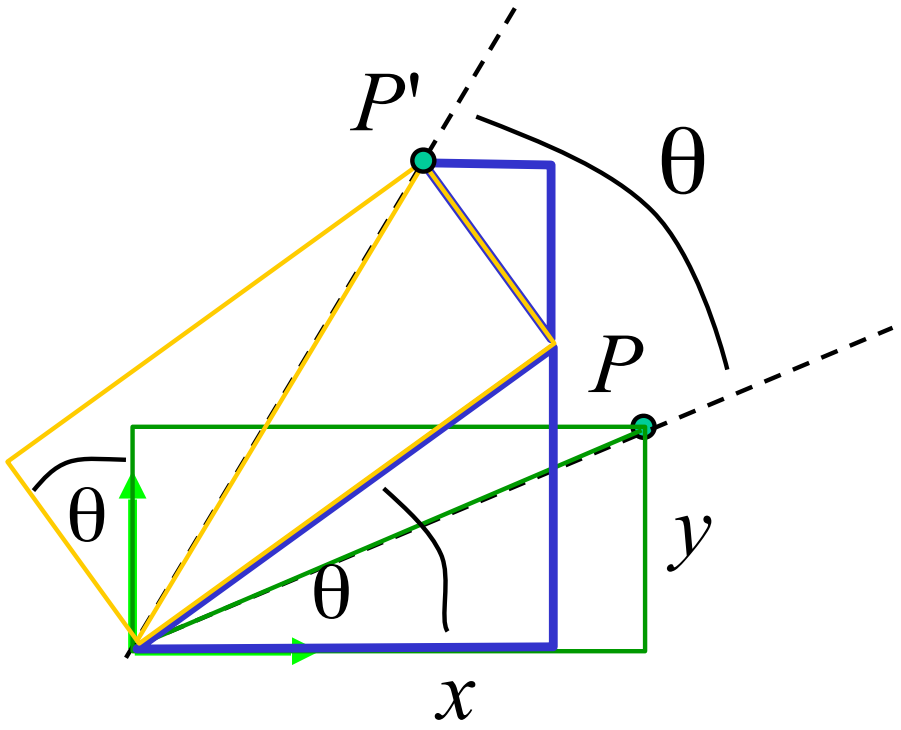
$$x = r\cos\phi$$
$$y = r\sin\phi$$

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

CLASS PARTITICIPATION!
calculate x', y' !

The University of New Mexico

Rotation around z fixed point of origin



$$Rotate(z,\theta)$$

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

$$z' = z$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & & \\ \sin\theta & \cos\theta & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

glRotatef(angle,x,y,z);
glRotated(angle,x,y,z);

Rotation about the z axis (3D) fixed point of origin

- 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

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

or in **homogeneous coordinates**

$$\mathbf{p}' = \mathbf{R}_z(\theta)\,\mathbf{p}$$

Rotation Matrix (3D) fixed point of origin

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

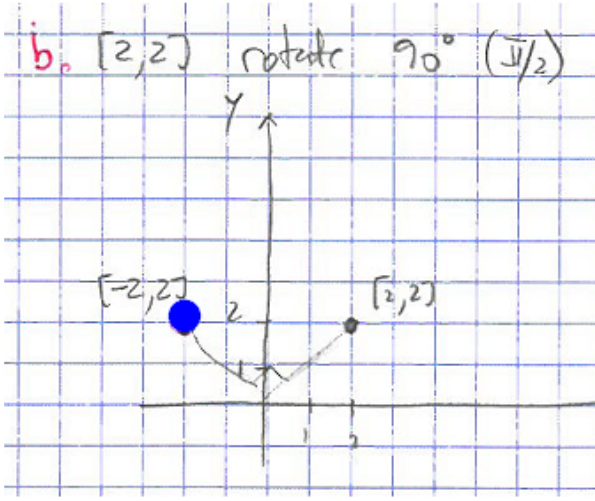
??????

Rotation Matrix- Example

Rotate [2,2] by 90° (π/2)

1. Write the Rotation Matrix using homogeneous coordinates (**hint: 3 X 3**)

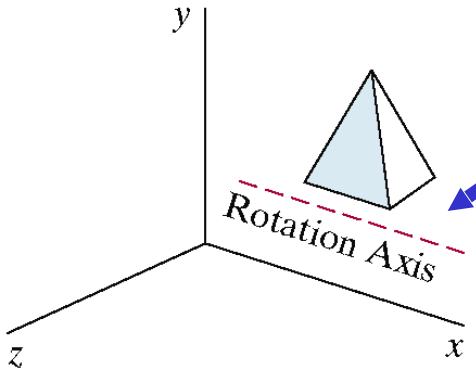
$$\begin{bmatrix} \cos(\pi/2) & -\sin(\pi/2) & 0 \\ \sin(\pi/2) & \cos(\pi/2) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$



2. Premultiply the Rotation Matrix with point [2,2] and find it's new location

CLASS PARTITICIPATION b!

Sequence of Transformations for Rotating an Object about an Axis that is parallel to x Axis



(a)
Original Position of Object

Rotation about x and y axes fixed point of origin

- Same argument as for rotation about *z* axis
  - For rotation about x axis, x **is unchanged**
  - For rotation about y axis, y is unchanged

$$R = R_x(\theta) = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

??????

$$R = R_y(\theta) = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$$

??????

The University of New Mexico

Scaling (3D) fixed point of origin

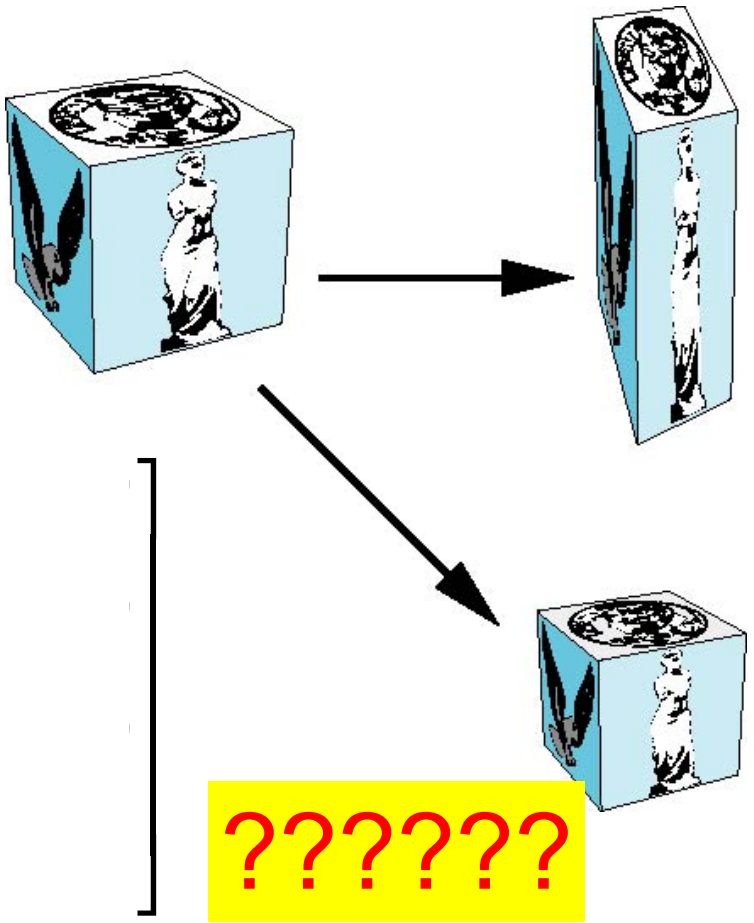
Expand or contract along each axis (fixed point of origin)

$x' = s_x x$

$y' = s_y x$

$z' = s_z x$

$p' = S\ p$



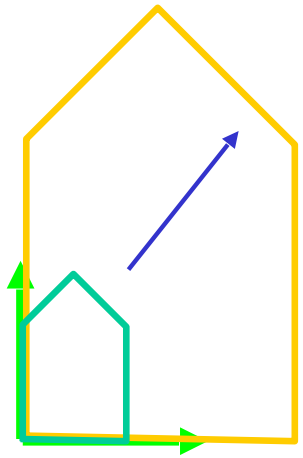
$$S = S(s_x, s_y, s_z) = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

http://www.cosc.canterbury.ac.nz/mukundan/cogr/Scl2D.html

The University of New Mexico

Scaling
fixed point of origin

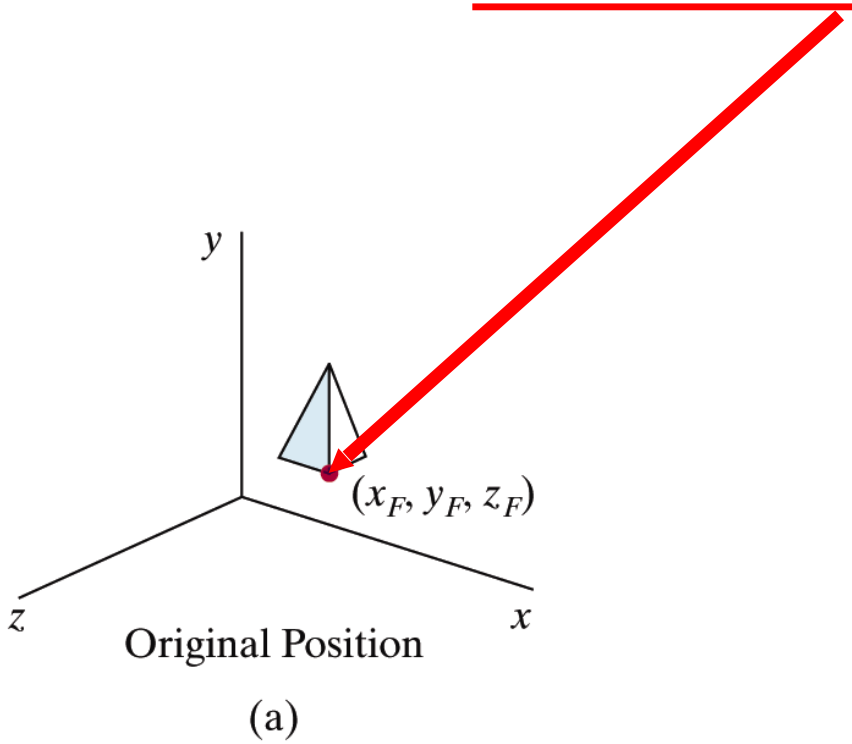
scale(a,b,c)



$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & & & \\ & b & & \\ & & c & \\ & & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

glScalef(a,b,c);
glScaled(a,b,c);

Sequence of Transformations for Scaling an Object relative to a selected fixed point

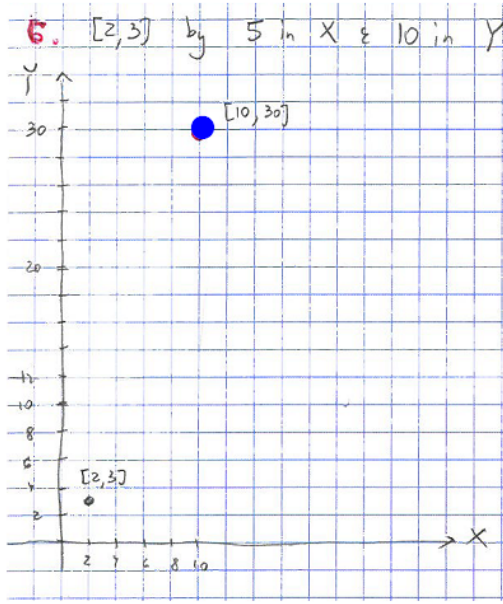


Scaling Matrix- Example

Scale [2,3] by 5 in the X direction and 10 in the Y direction

1. Write the Scaling Matrix using **homogeneous coordinates** (**hint: 3 X 3**)

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$



2. Premultiply the Scaling Matrix with point [2,3] and find it's new location

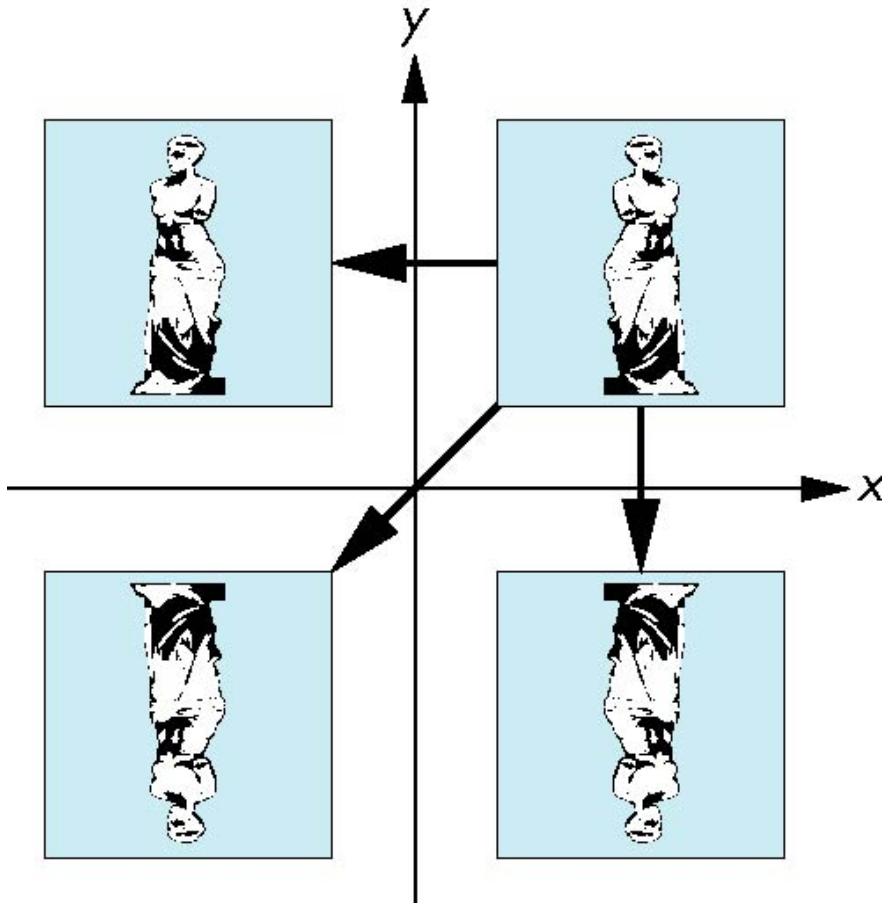
CLASS PARTITICIPATION c!

The University of New Mexico

Reflection

corresponds to negative scale factors

$s_x = -1$ $s_y = 1$



original

$s_x = -1$ $s_y = -1$

$s_x = 1$ $s_y = -1$

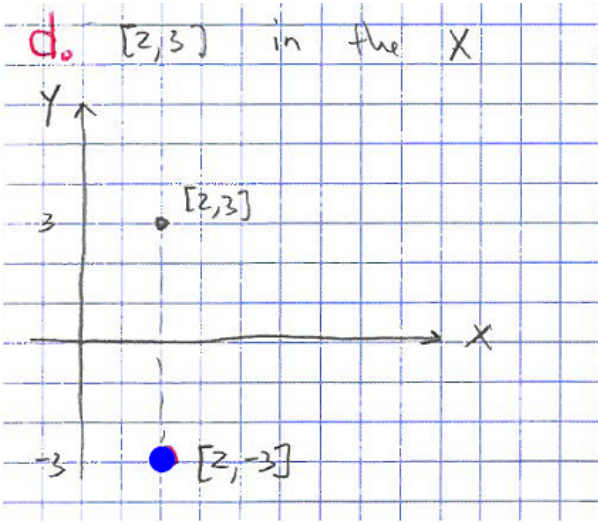
http://www.shodor.org/interactivate/activities/Transmographer/
http://www.shodor.org/interactivate/activities/TransmographerTwo/

Reflection Matrix- Example

Reflect [2,3] in the X direction

1. Write the Reflection Matrix using **homogeneous coordinates** (**hint: 3 X 3**)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

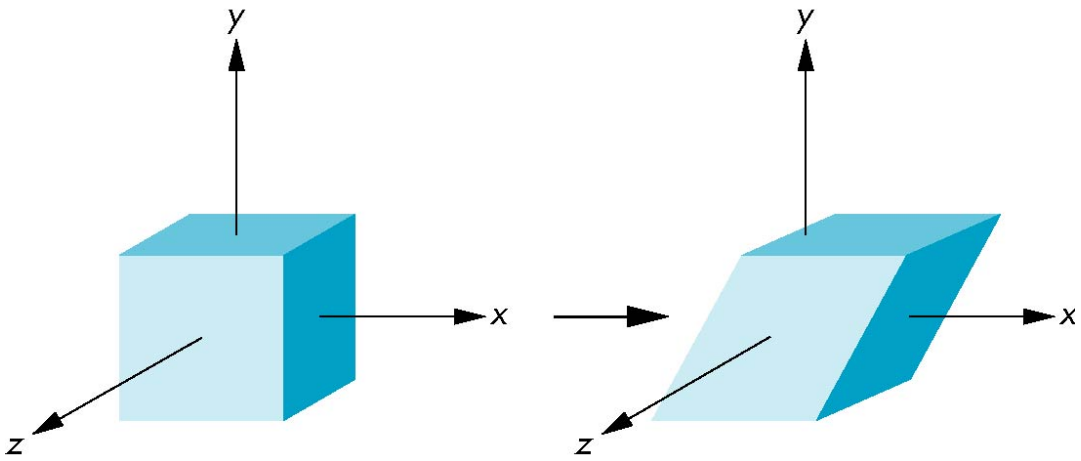


2. Premultiply the Reflection Matrix with point [2,3] and find it's new location

CLASS PARTITICIPATION d!

Shear

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



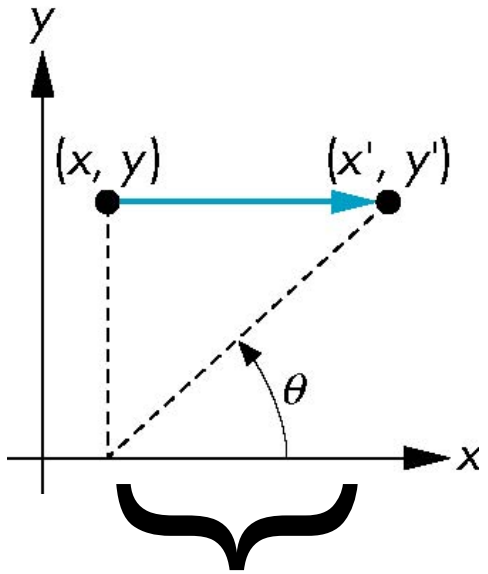
http://www.cosc.canterbury.ac.nz/mukundan/cogr/Shear2D.html

Shear Matrix

Consider simple shear along x axis

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

$$H(\theta) = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$



??????

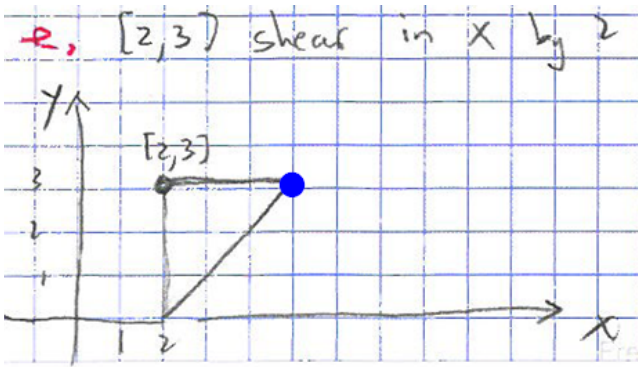
The University of New Mexico

Shear Matrix- Example

Shear [2,3] in the X direction by 3

1. Write the Shear Matrix using **homogeneous coordinates** (**hint: 3 X 3**)

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



$$\cot\theta = \cot 45 = 1$$

2. Premultiply the Shear Matrix with point [2,3] and find it's new location

CLASS PARTITICIPATION e!

Inverses

- Although we could compute inverse matrices by general formulas, we can use simple geometric observations

Translation: $T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z)$

Rotation: $R^{-1}(\theta) = R(-\theta)$

- Holds for any rotation matrix
- Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$

$$R^{-1}(\theta) = R^{T}(\theta)$$

Scaling: $S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z)$

2D Homogeneous Coordinate Transformations

The University of New Mexico

- Consider the rotation matrix:

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

- The 2 x 2 submatrix **columns** are:
  - unit vectors (length=1)
  - perpendicular (dot product=0)
  - vectors into which X-axis and Y-axis rotate

$$1 = \sin^2\theta + \cos^2\theta$$

- The 2 x 2 submatrix **rows** are:
  - unit vectors
  - perpendicular
  - vectors that rotate into X-axis and Y-axis
- **Preserves lengths and angles of original geometry**. Therefore, matrix is a "rigid body" transformation.

Objectives

- Introduce standard transformations
  - Rotation
  - Translation
  - Scaling
  - Shear
- Derive **homogeneous coordinate transformation matrices**
- Learn to build **arbitrary transformation matrices from simple transformations**

Concatenation

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

Because the same transformation is applied to many **vertices**, the **cost of forming a matrix** $M=ABCD$ is not significant compared to the cost of computing $\mathbf{M}\mathbf{p}$ for **many vertices** $\mathbf{p}$

The difficult part is how to form a **desired transformation** from the specifications in the application

Order of Transformations

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

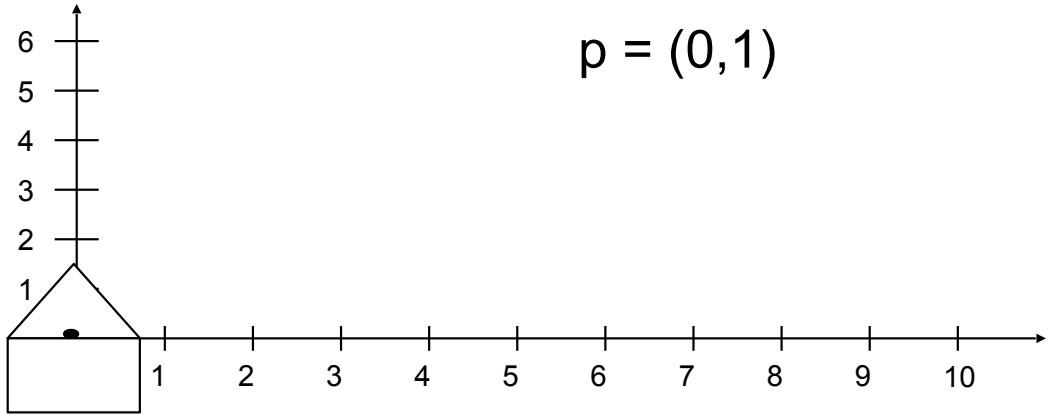
$$\mathbf{p'} = \mathbf{ABCp} = \mathbf{A(B(Cp))}$$

- Note many references use column matrices to represent points. In terms of **column matrices**

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

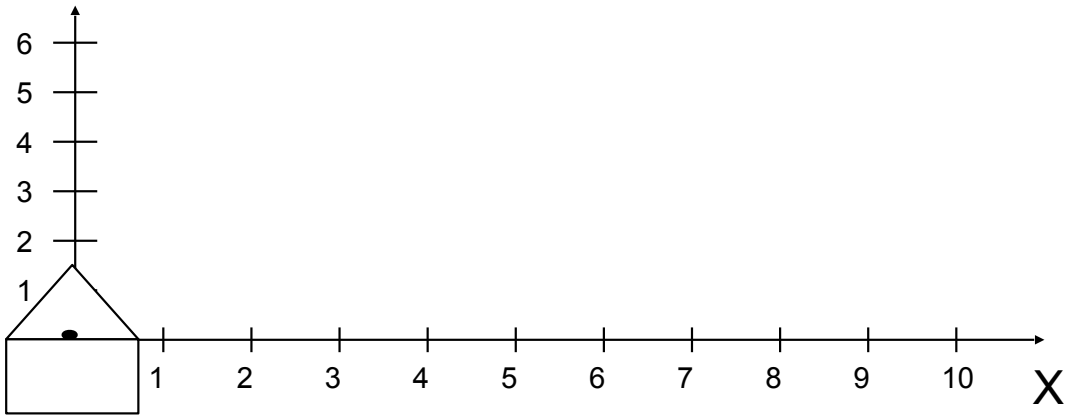
Matrix Multiplication is NOT Commutative

Translate by x=6, y=0
then Rotate by 45º



p = (0,1)

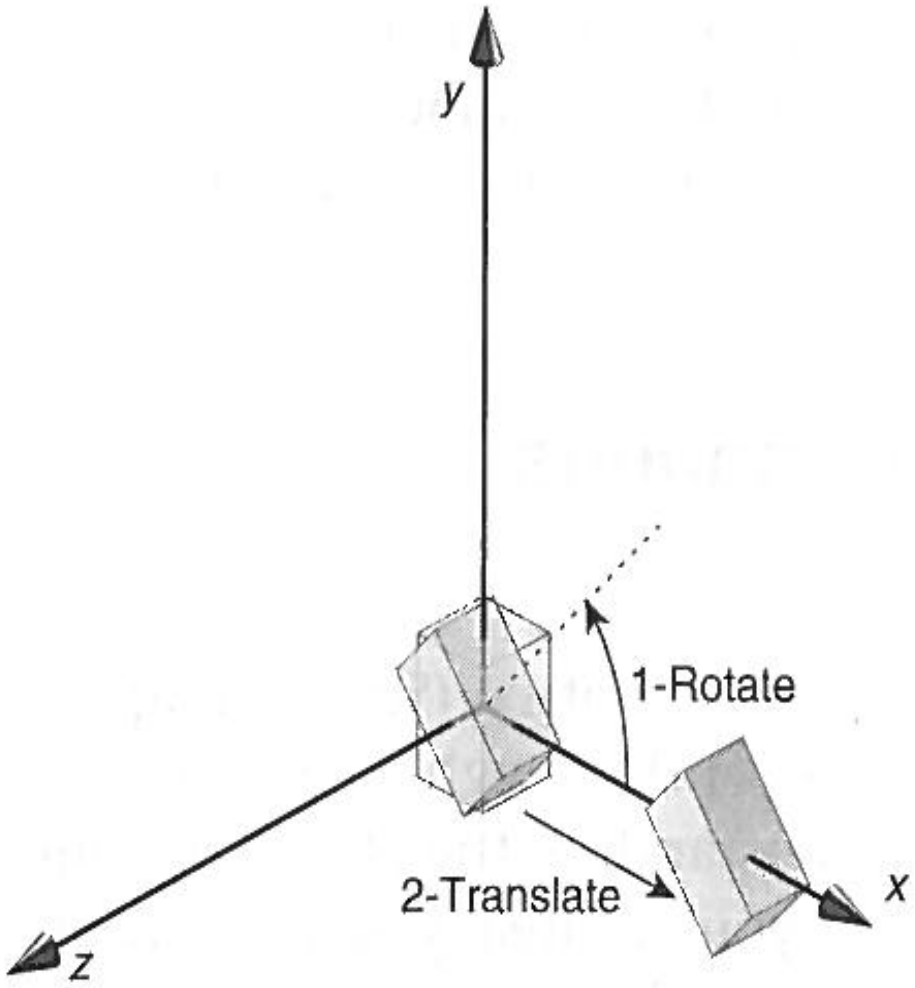
Rotate by 45º then
Translate by x=6, y=0

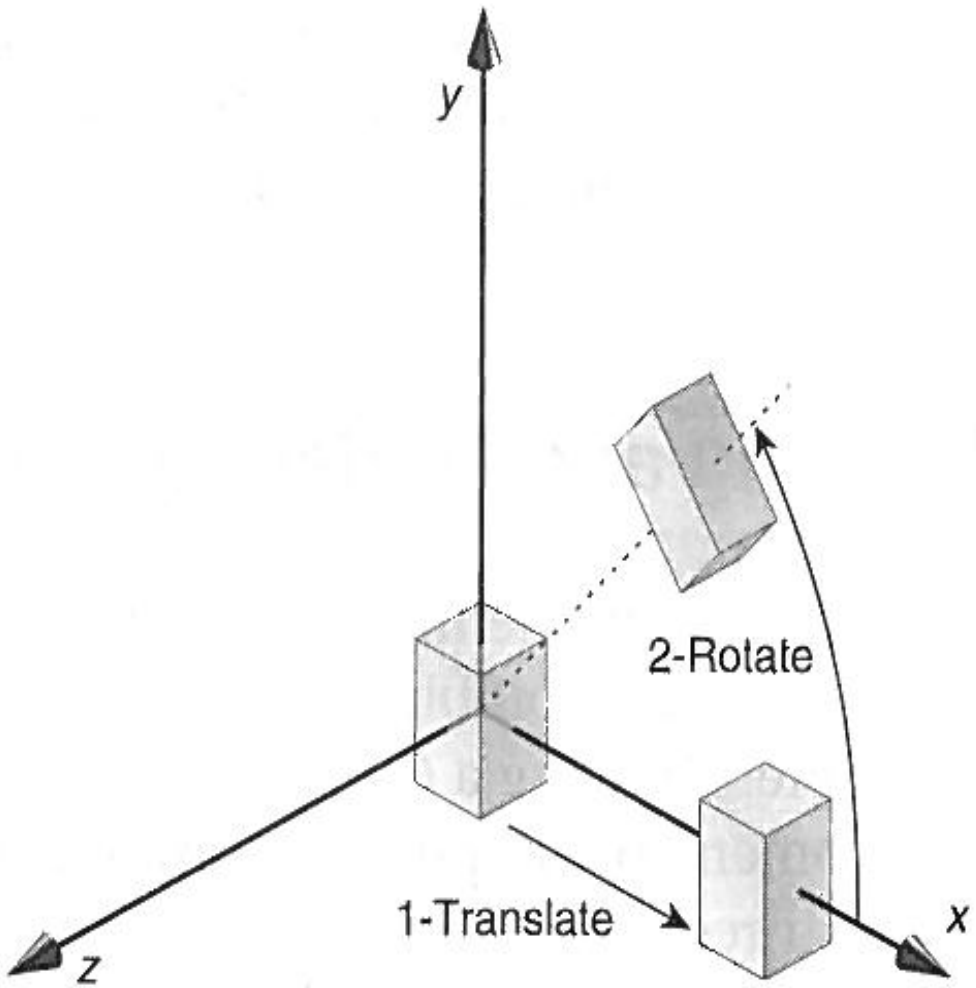


CLASS PARTITICIPATION f!

Matrix Multiplication is NOT Commutative

Rotating first or Translating first





Matrix Multiplication is NOT Commutative

$$\cos 45 = 1/\sqrt{2}$$

$$\sin 45 = 1/\sqrt{2}$$

Translate by x=6, y=0 then rotate by 45°

$$\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 6/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} & 6/\sqrt{2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5/\sqrt{2} \\ 7/\sqrt{2} \\ 1 \end{bmatrix}$$

Rotate by 45° then translate by x=6, y=0

$$\begin{bmatrix} 1 & 0 & 6 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 6 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 - 1/\sqrt{2} \\ 1/\sqrt{2} \\ 1 \end{bmatrix}$$

CLASS PARTITICIPATION f!

3D Basic Transformations

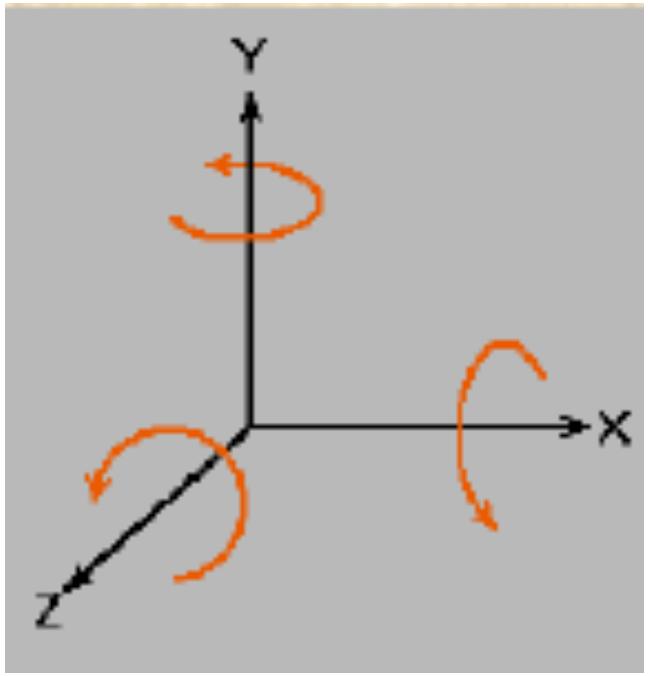
(right-handed coordinate system)

- **Translation**

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **Scaling**

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



3D Basic Transformations

(right-handed coordinate system)

- **Rotation about X-axis**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation about Y-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation about Z-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

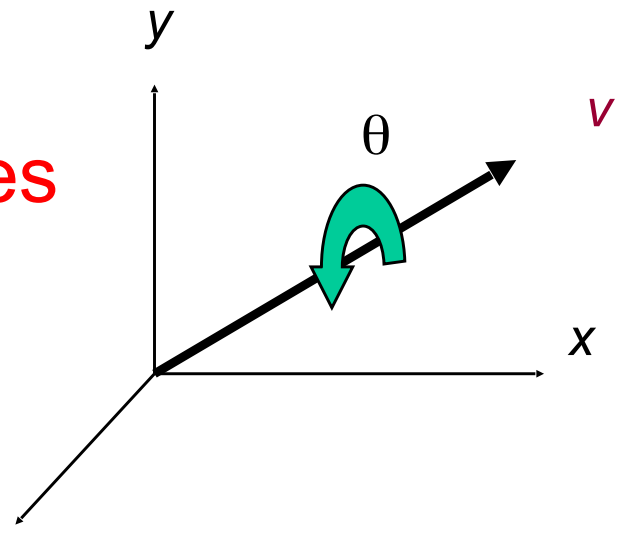
General Rotation About the Origin

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

$$R(\theta) = R_z(\theta_z)\, R_y(\theta_y)\, R_x(\theta_x)$$

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

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



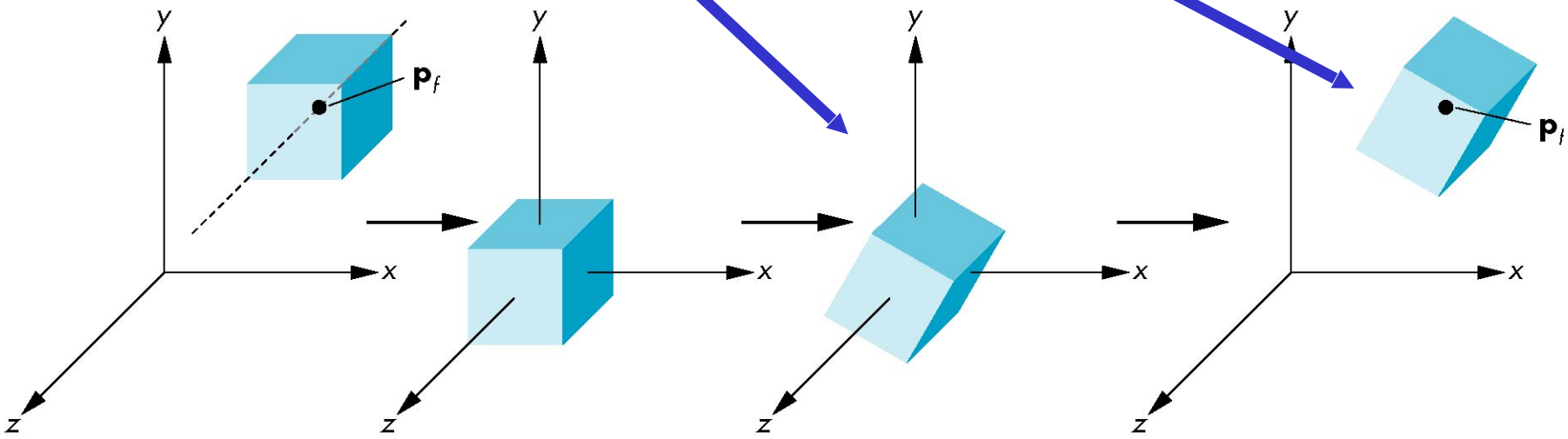
http://www.cosc.canterbury.ac.nz/mukundan/cogr/Rot3D.html

Rotation About a Fixed Point other than the Origin

1. Move fixed point to origin
2. Rotate
3. Move fixed point back

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

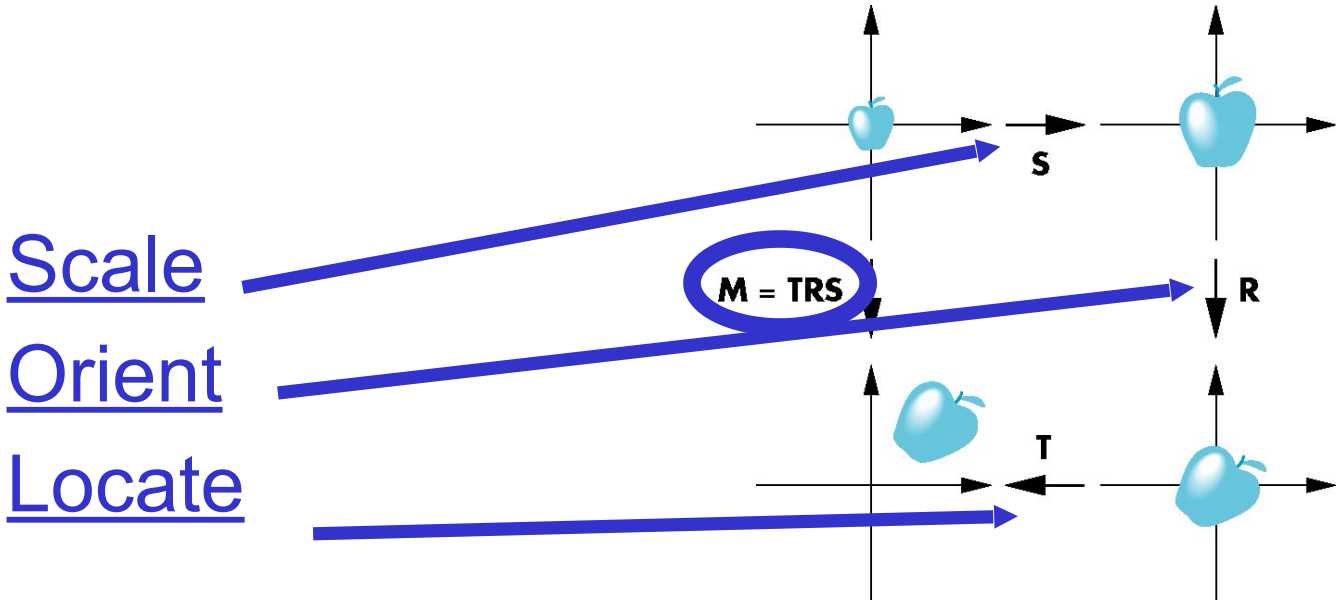
??????



Instancing

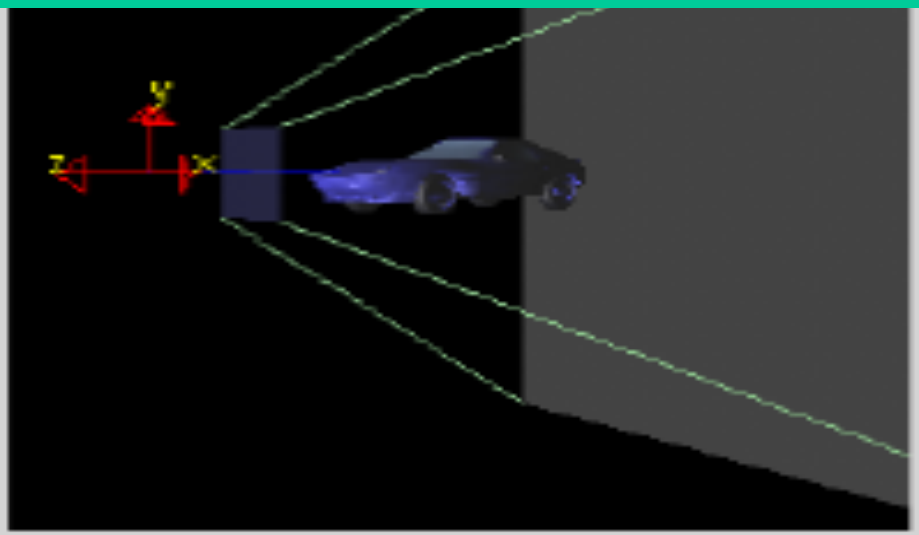
In Modeling, we often start with a simple **Object centered at the origin**, oriented with the axis, and a a standard size

We apply an *instance transformation* **to its Vertices** to



The transformation tutorial program demonstrates how the basic transformations of rotate, translate and scale operate in OpenGL. The order of the transforms can be changed to see how that effects rendering.

World-space view



Screen-space view

Command manipulation window

```
glTranslatef( 0.00  , 0.00  , 0.00  );
   glRotatef( 0.0   , 0.00  , 1.00  , 0.00  );
    glScalef( 1.00  , 1.00  , 1.00  );
    glBegin( . . . );
```

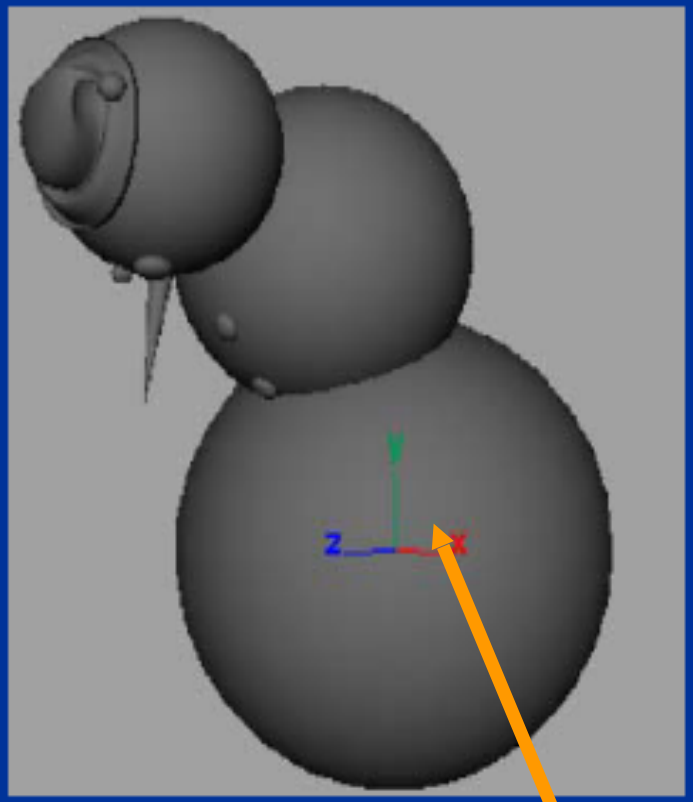
TRANSFORMATIONS DEMO!

KINGSTON:\COSC 4370 SUMMER 2010\NATE ROBINS OPENGL TUTORIALS\tutors\transformations.exe

The University of New Mexico

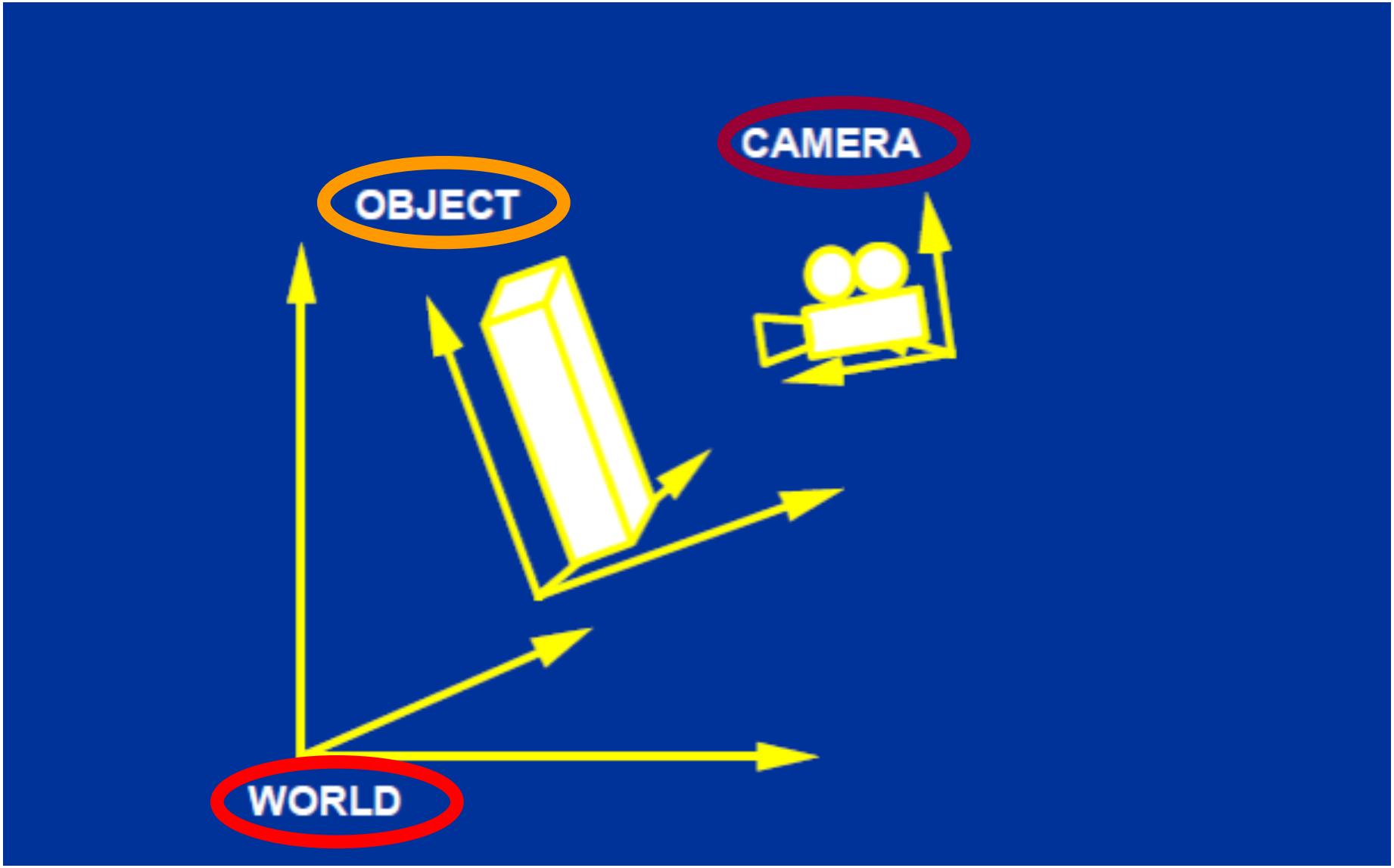
Object Modeling

Modeling with primitive shapes

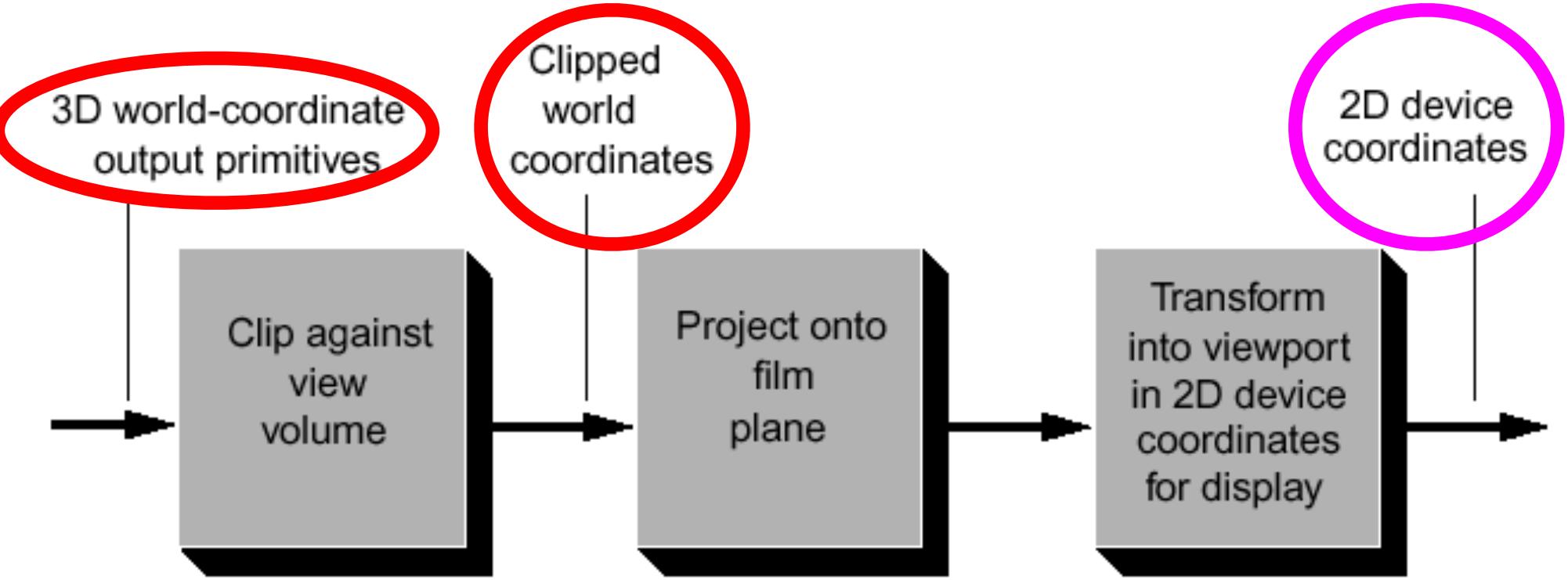


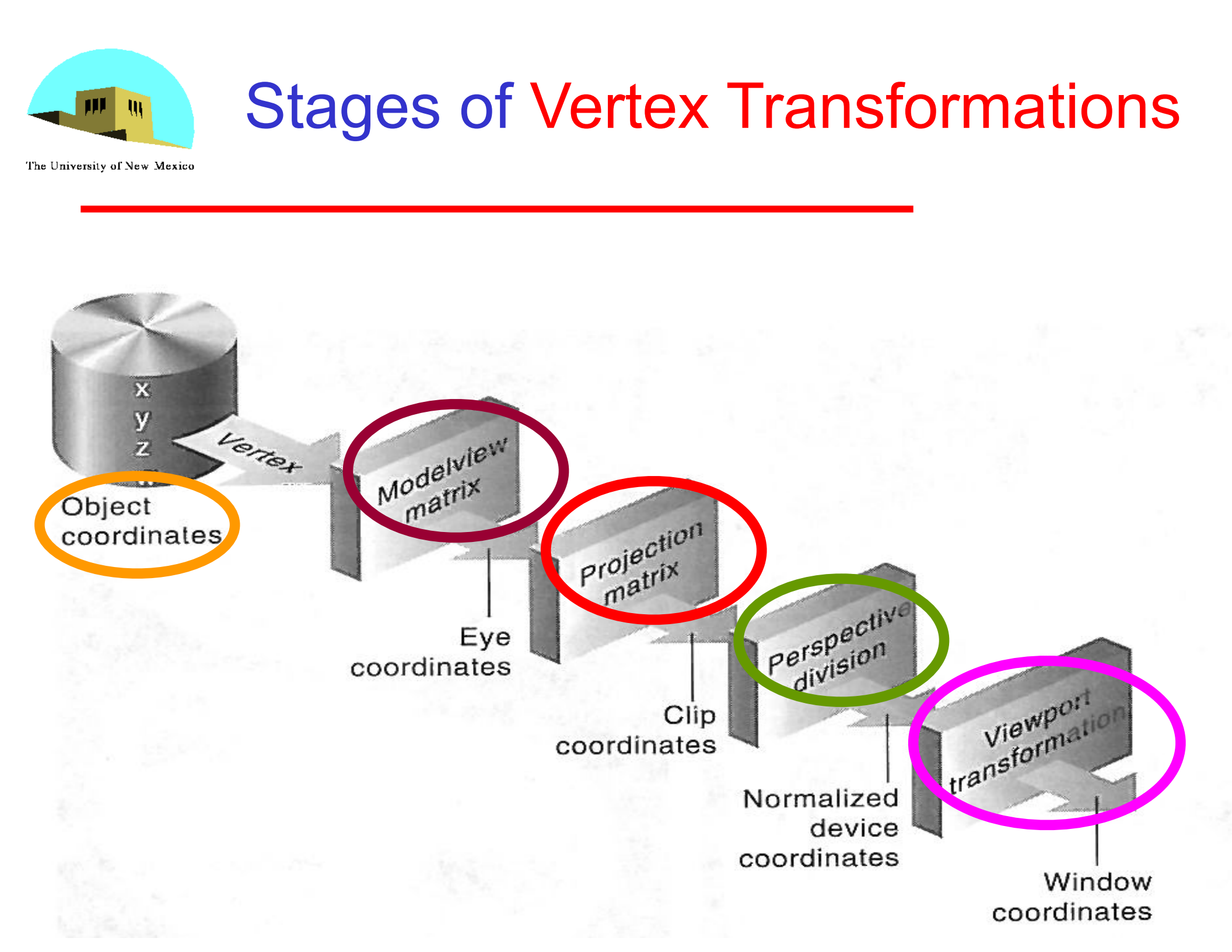
Object frame

Viewing transformations

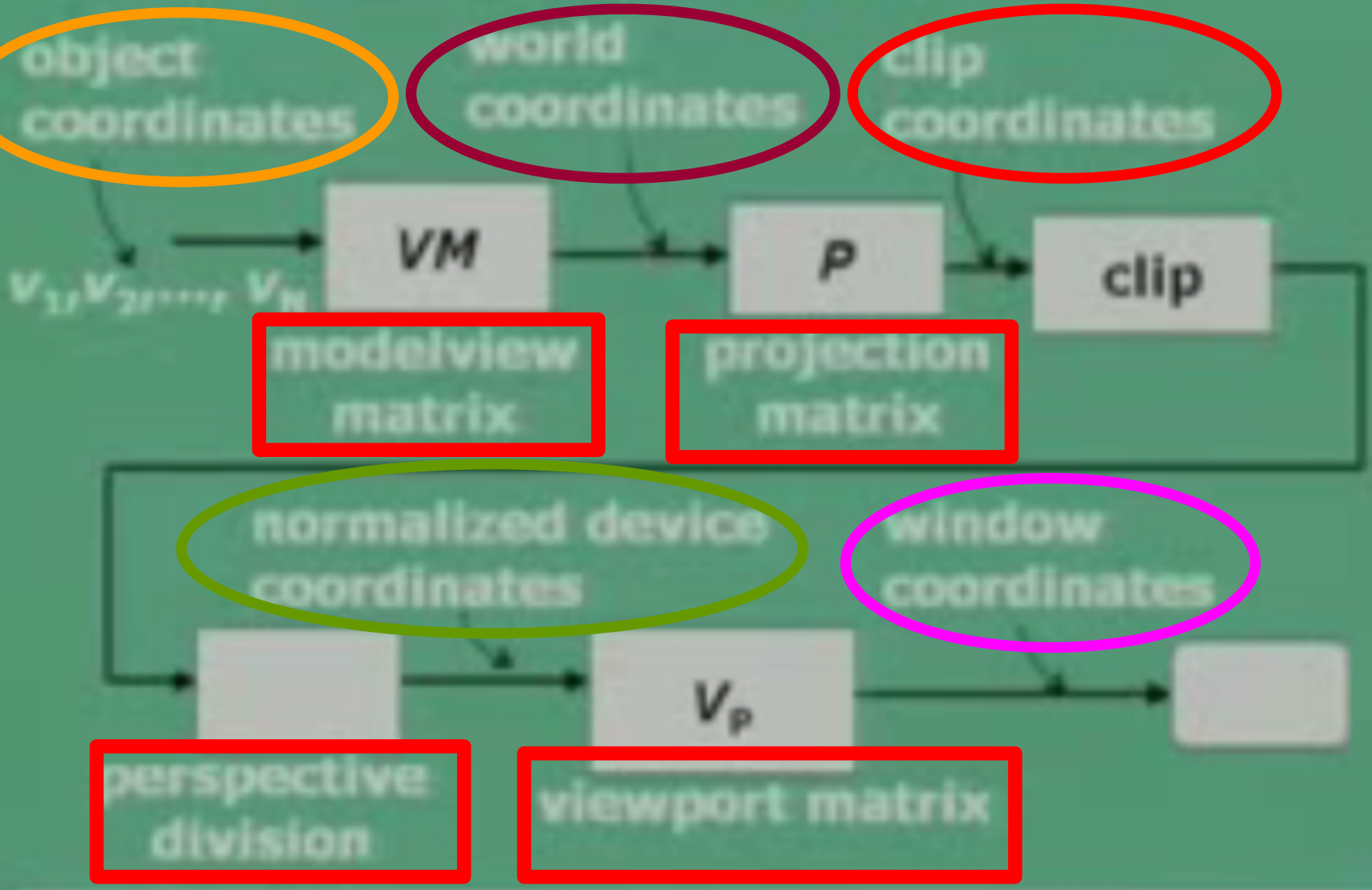


Viewing transformations



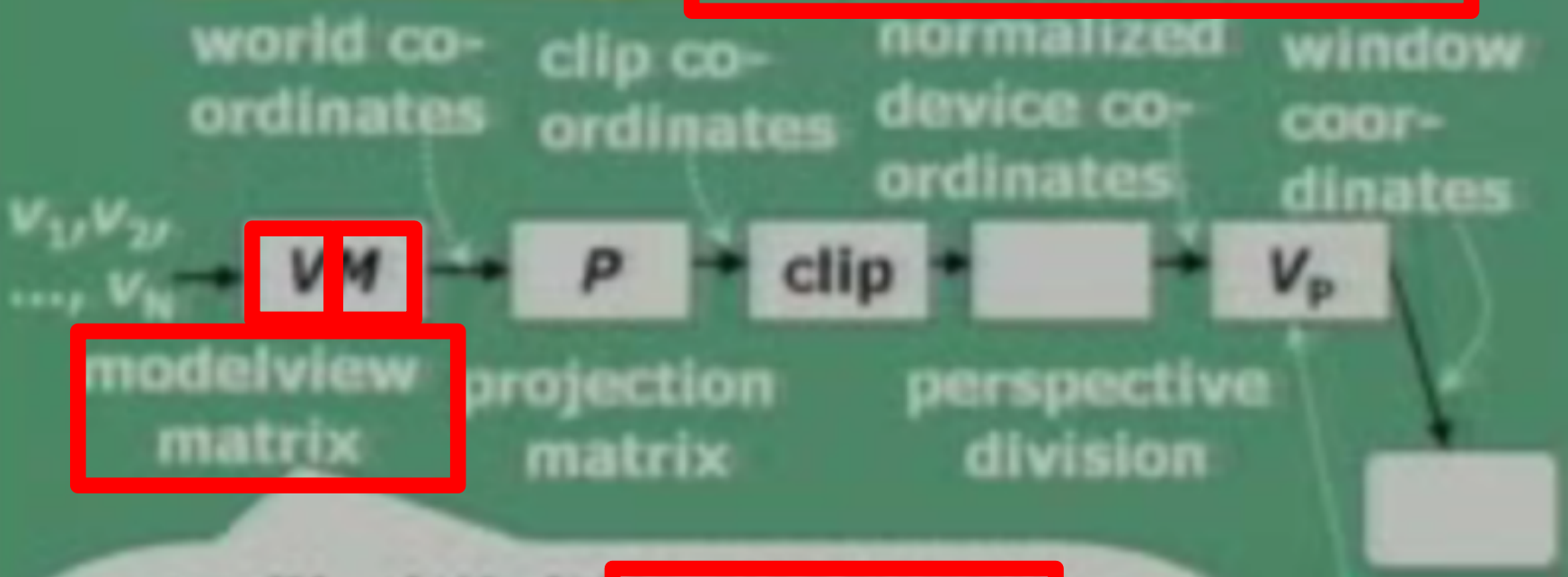

The University of New Mexico
Stages of Vertex Transformations
x
y
z
Vertex
Object coordinates
Modelview matrix
Eye coordinates
Projection matrix
Clip coordinates
Perspective division
Normalized device coordinates
Viewport transformation
Window coordinates

3D Viewing Pipeline



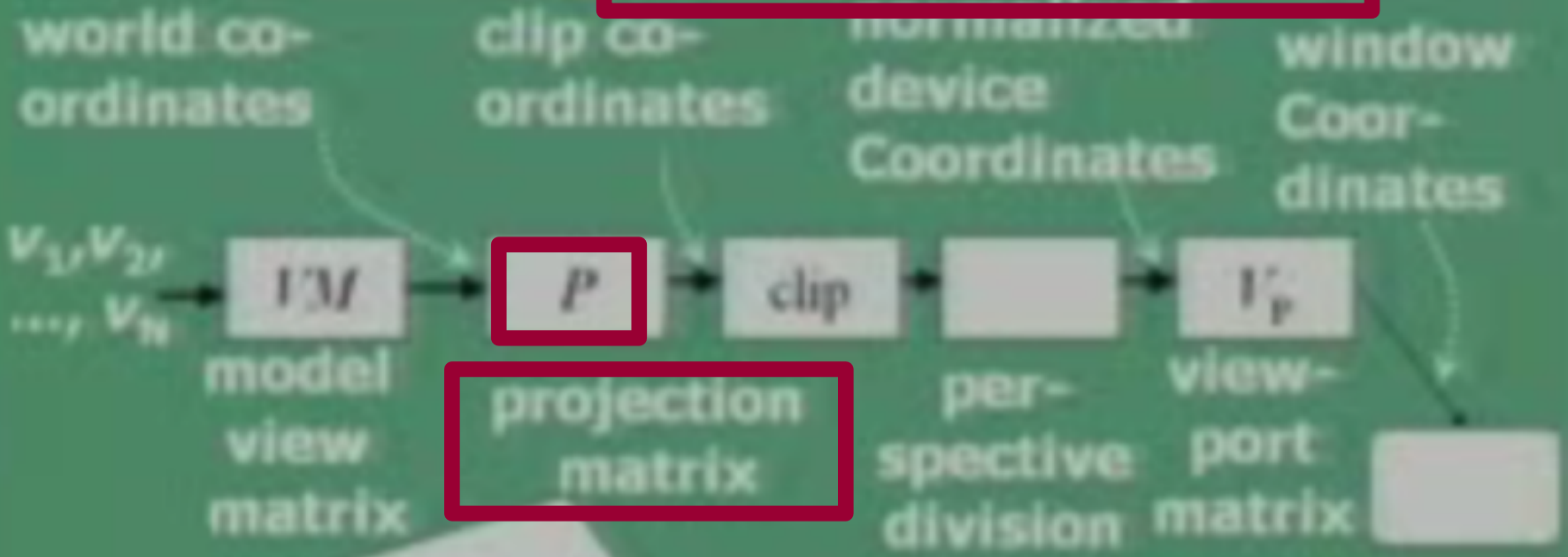
From F. S. Hill Jr., Computer Graphics using OpenGL

3D Viewing – ModelView Matrix




```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// viewing transform
    gluLookAt( eyeX, eyeY, eyeZ,
lookAtX, lookAtY, lookAtZ, upX, upY, upZ);
// model transform
    glTranslatef(delX, delY, delZ);
    glRotatef(angle, i, j, k);
    glScalef(multX,multY, multZ);
```

3D Viewing – Projection Matrix




```
glMatrixMode( GL_PROJECTION );
glLoadIdentity()
// perspective transform
gluPerspective( viewAngle, aspectRatio,nearZ,farZ );
// other commands for setting projection matrix
glFrustum(left, right, top, bottom);
glOrtho(left, right, top, bottom);
gluOrtho2D(left, right, top, bottom);
```

Structure of a GLUT Program

```
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
    GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Interactive rotating
    cube"); // with size & position
```

Structure of a GLUT Program

```
int main(int argc, char **argv) {

    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_DOUBLE |
    GLUT_RGB | GLUT_DEPTH);

    glutCreateWindow("Interactive rotating
    cube"); // with size & position

    glutDisplayFunc(display);
    // display callback, routines for drawing
```

Structure of a GLUT Program

```
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
    GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Interactive rotating
    cube"); // with size & position
    glutDisplayFunc(display);
    // display callback, routines for drawing
    glutKeyboardFunc(myKeyHandler);
    // keyboard callback
```

Structure of a GLUT Program

```
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
    GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Interactive rotating
    cube"); // with size & position
    glutDisplayFunc(display);
    // display callback, routines for drawing
    glutKeyboardFunc(myKeyHandler);
    // keyboard callback
    glutMouseFunc(myMouseClickHandler);
    // mouse callback
```

```
glutMotionFunc(myMouseMotionHandler);
// mouse move callback
```

```
    glutMotionFunc(myMouseMotionHandler);
    // mouse move callback
    init();
    glutMainLoop();
}
```

```
    glutMotionFunc(myMouseMotionHandler);
    // mouse move callback

    init();

    glutMainLoop();

}

void display() {...}

void myKeyHandler( unsigned char key, int x,
int y) {...}
```

```
    glutMotionFunc(myMouseMotionHandler);
    // mouse move callback

    init();

    glutMainLoop();
}
void display() {...}

void myKeyHandler( unsigned char key, int x,
int y) {...}

void myMouseClickHandler( int button, int
state, int x, int y ) {...}

void myMouseMotionHandler( int x, int y) {...}
```

Viewing in 2D

```
void init(void) {

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0f, 0.0f, 1.0f);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluOrtho2D(
            0.0, // left
            screenWidth, // right
            0.0, // bottom
            screenHeight); // top
}
```

Drawing in 2D

```
glBegin(GL_POINTS);
    glVertex2d(x1, y1);
    glVertex2d(x2, y2);
    .
    .
    .
    glVertex2d(xn, yn);
glEnd();
```

Drawing in 2D

```
glBegin(GL_POINTS);
        glVertex2d(x1, y1);
        glVertex2d(x2, y2);
        .
        .
        .
        glVertex2d(xn, yn);
glEnd();
```

GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_POLYGON

Assigning Colours

Current drawing colour maintained as a state.

Colour components - red, green, blue in range [0...1] as float or [0...255] as unsigned byte

```
GLfloat myColour[3] = {0, 0, 1}; // blue

glColor3fv( myColour ); // using vector of floats
```

```
glColor3f(1.0, 0.0, 0.0); // red using floats
```

```
glColor3ub(0, 255, 0);   // green using unsigned bytes
```

Colour Interpolation

If desired, a polygon can be smoothly shaded to interpolate colours between vertices.

This is accomplished by using the GL_SMOOTH shading mode (the OpenGL default) and by assigning a desired colour to each vertex.

```
glShadeModel(GL_SMOOTH);
// as opposed to GL_FLAT

glBegin(GL_POLYGON);
        glColor3f(1.0, 0, 0 ); // red
        glVertex2d(0, 0);
        glColor3f(0, 0, 1.0 ); // blue
        glVertex2d(1, 0);
        glColor3f(0, 1.0, 0 ); // green
        glVertex2d(1, 1);
        glColor3f(1.0, 1.0, 1.0 ); // white
        glVertex2d(0, 1);
glEnd();
```

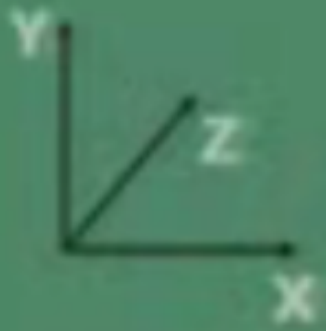
Lighting up the 3D World

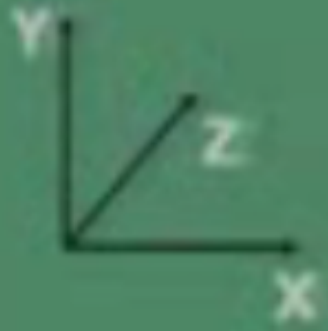
Ambient light

(source at infinity)

Diffuse light

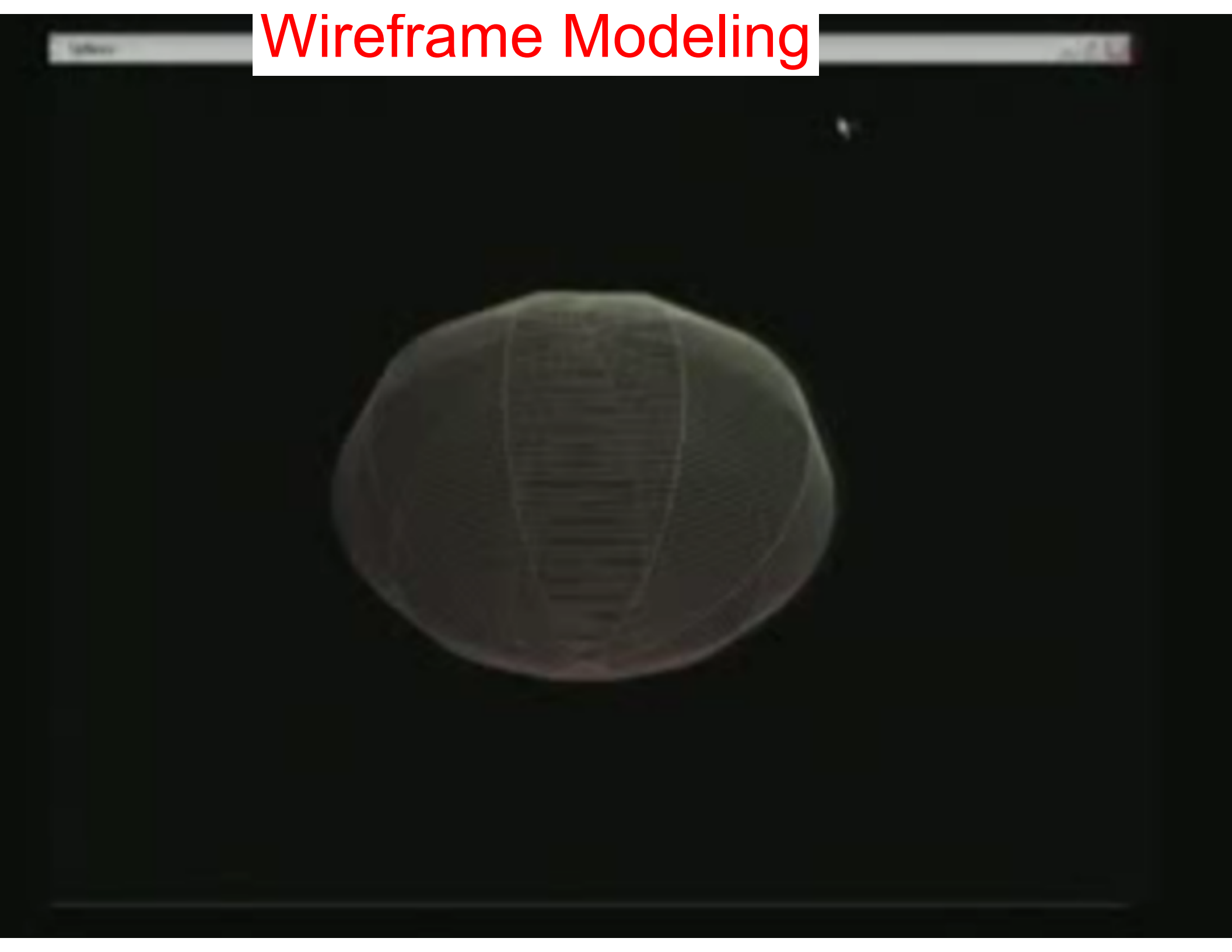
(from a point source)



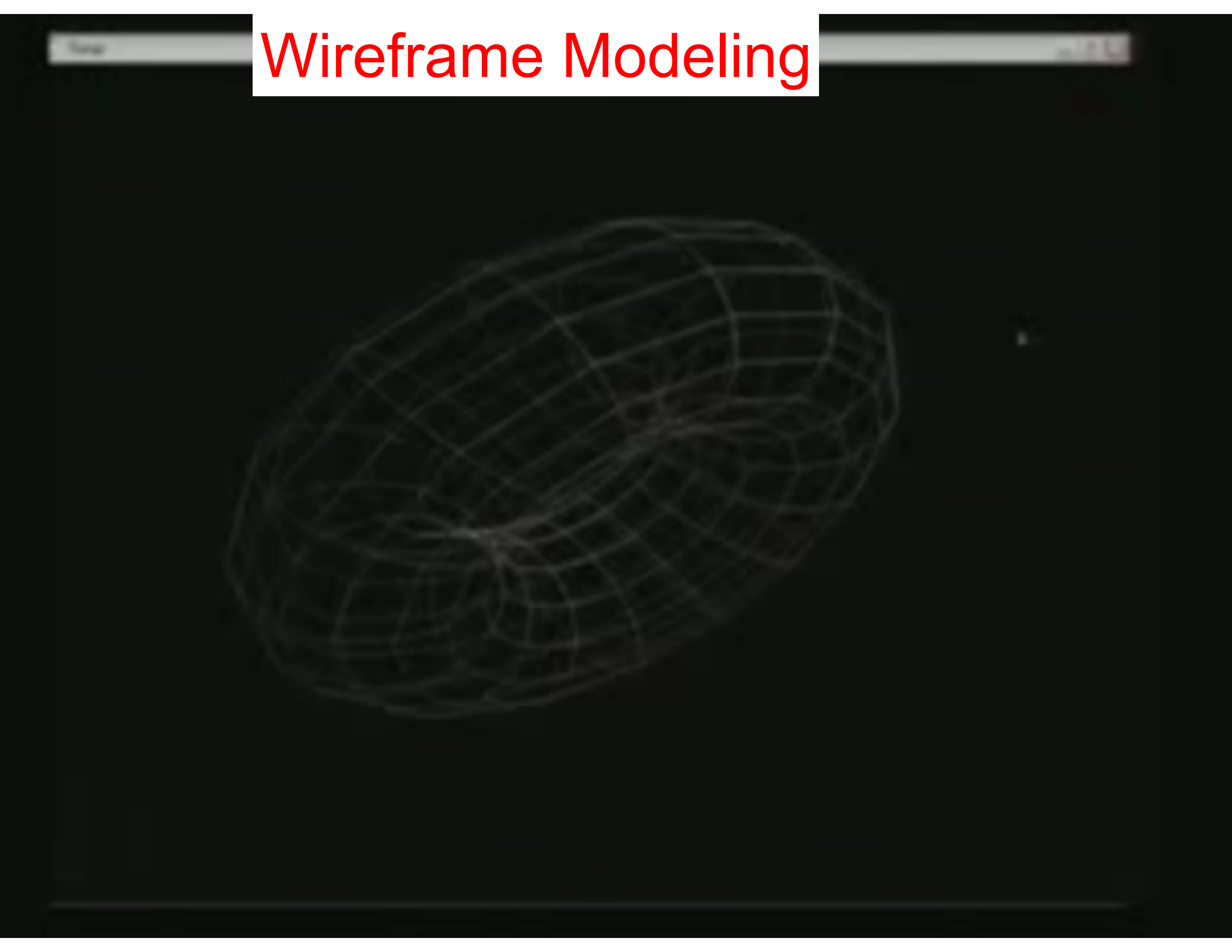


```
GLfloat light0_colour[] = {1, 1.0, 1, 1.0};
GLfloat light0_position[] = {0.0, 1.0, 0.0, 0.0};

 // Setting up light type and position
glLightfv(GL_LIGHT0, GL_AMBIENT,
light0_colour); // use GL_DIFFUSE for diffuse

glLightfv(GL_LIGHT0, GL_POSITION,
light0_position);

 //  Enable light
glEnable(GL_LIGHT0); // can have other lights
glEnable(GL_LIGHTING);
glShadeModel(GL_SMOOTH);
```

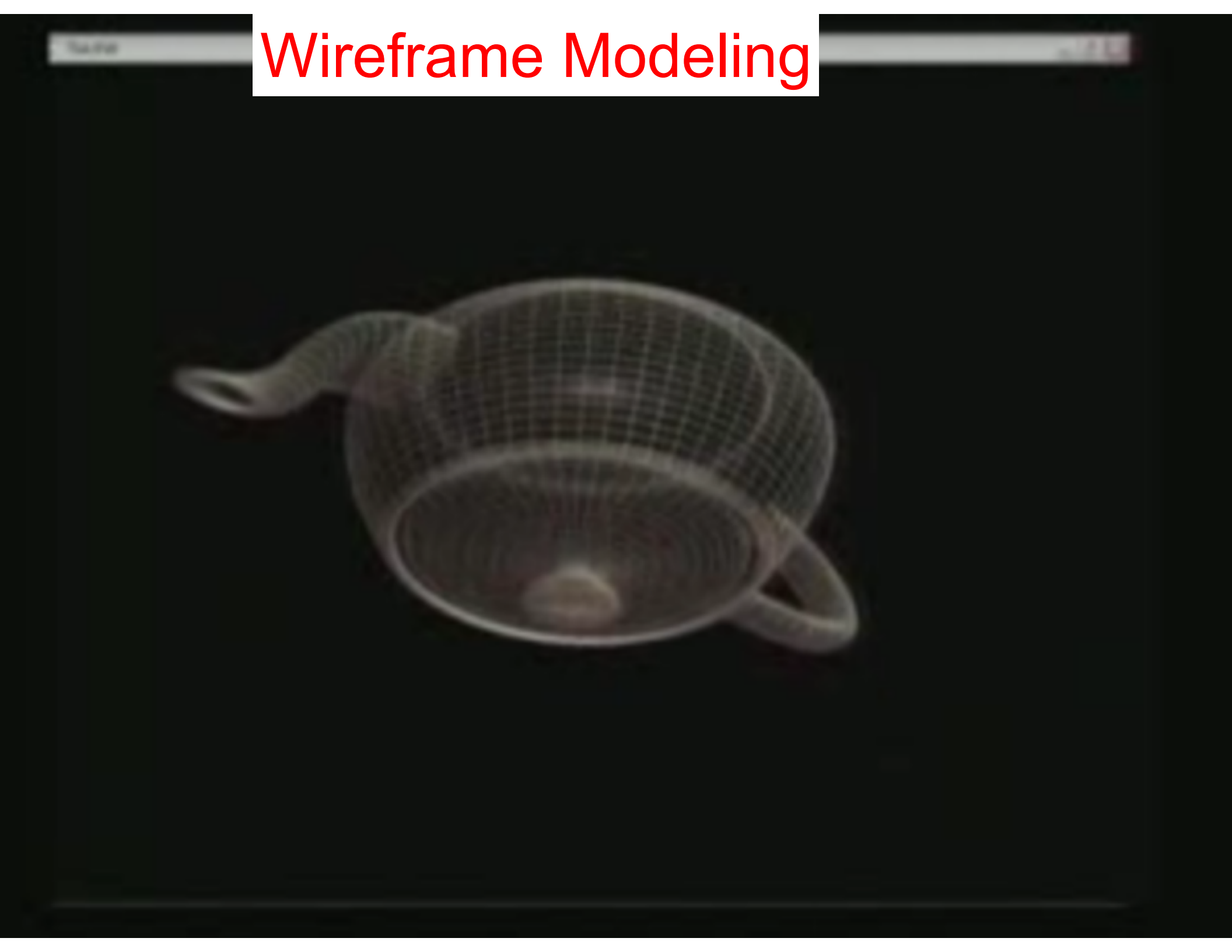
Wireframe Modeling



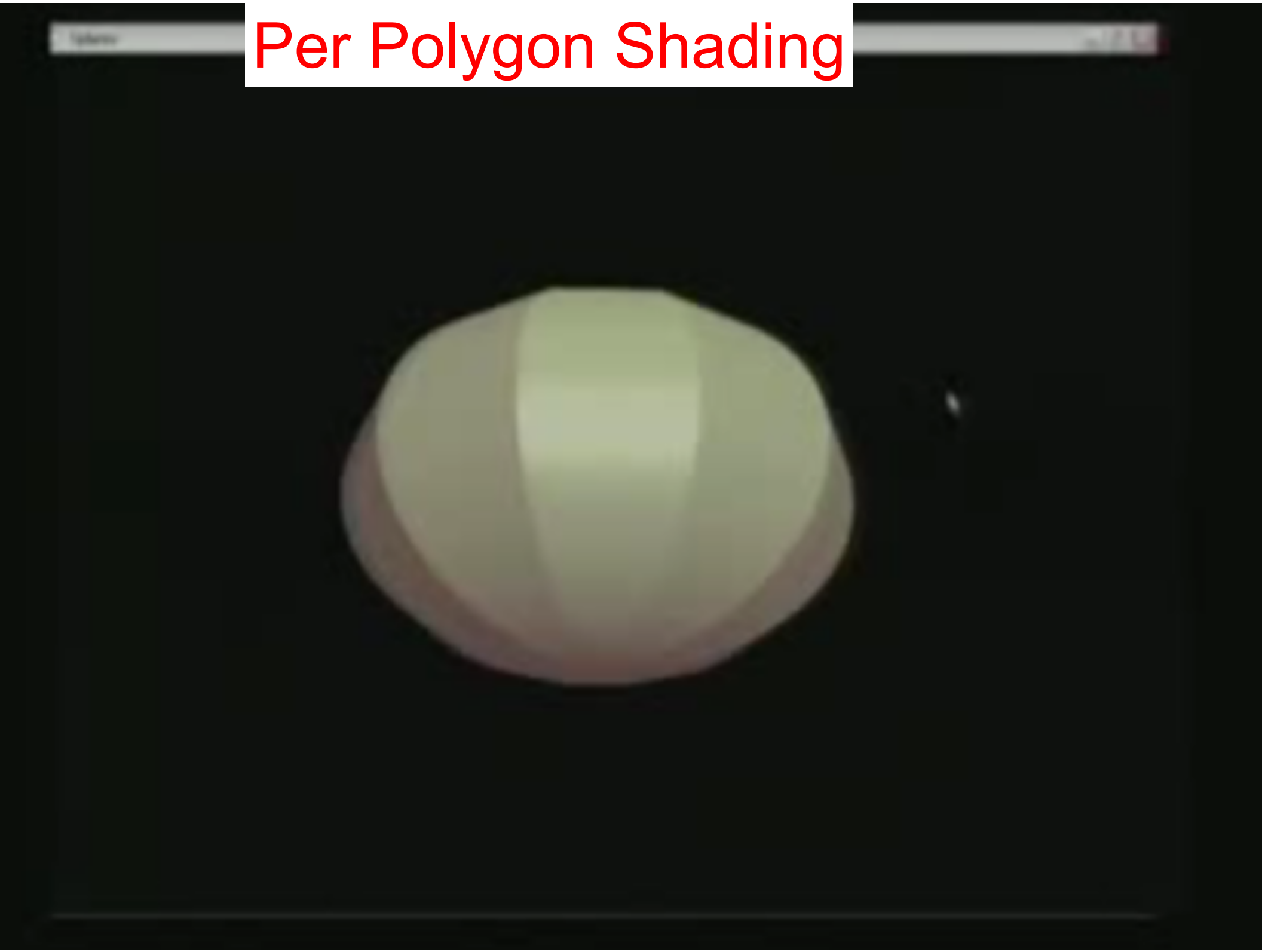
Wireframe Modeling



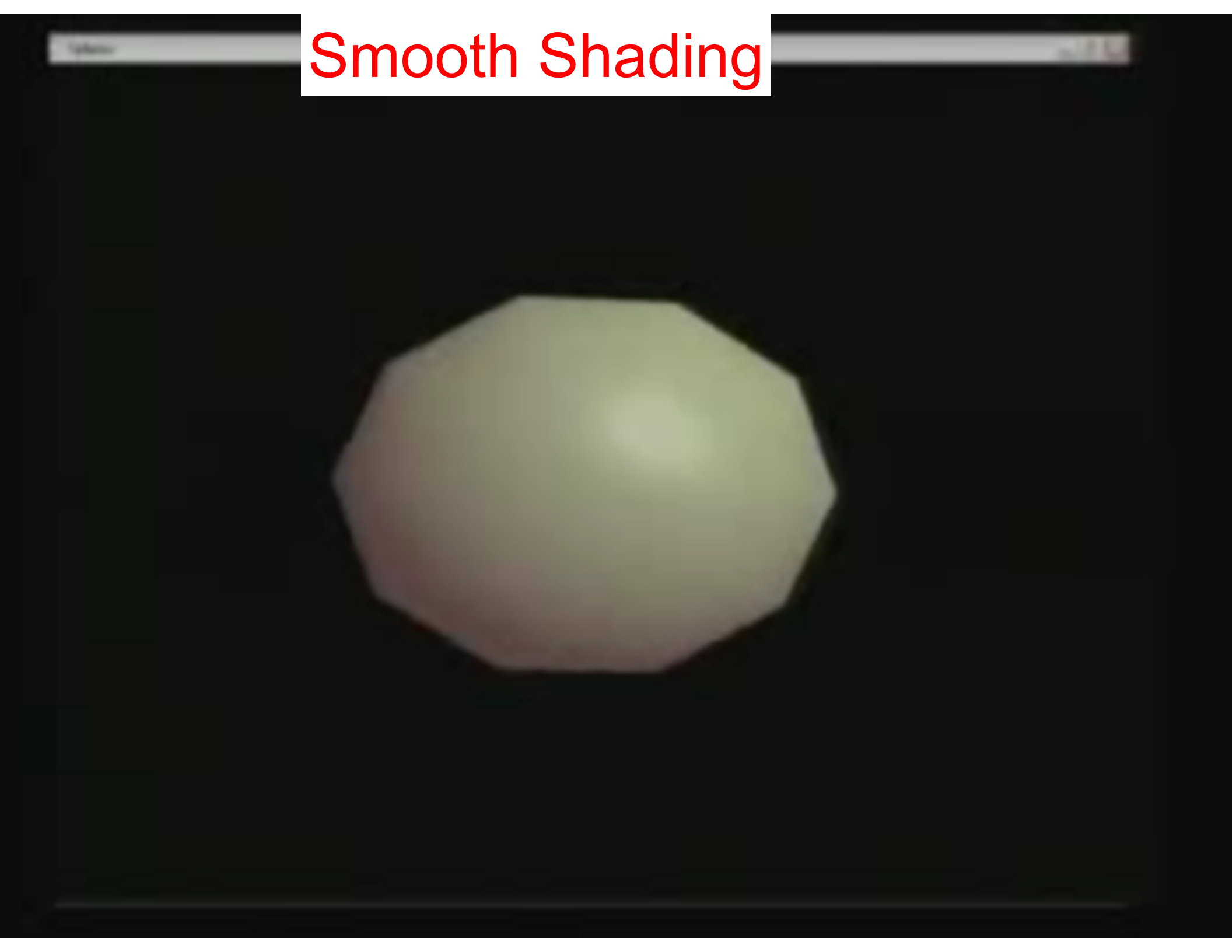
Wireframe Modeling



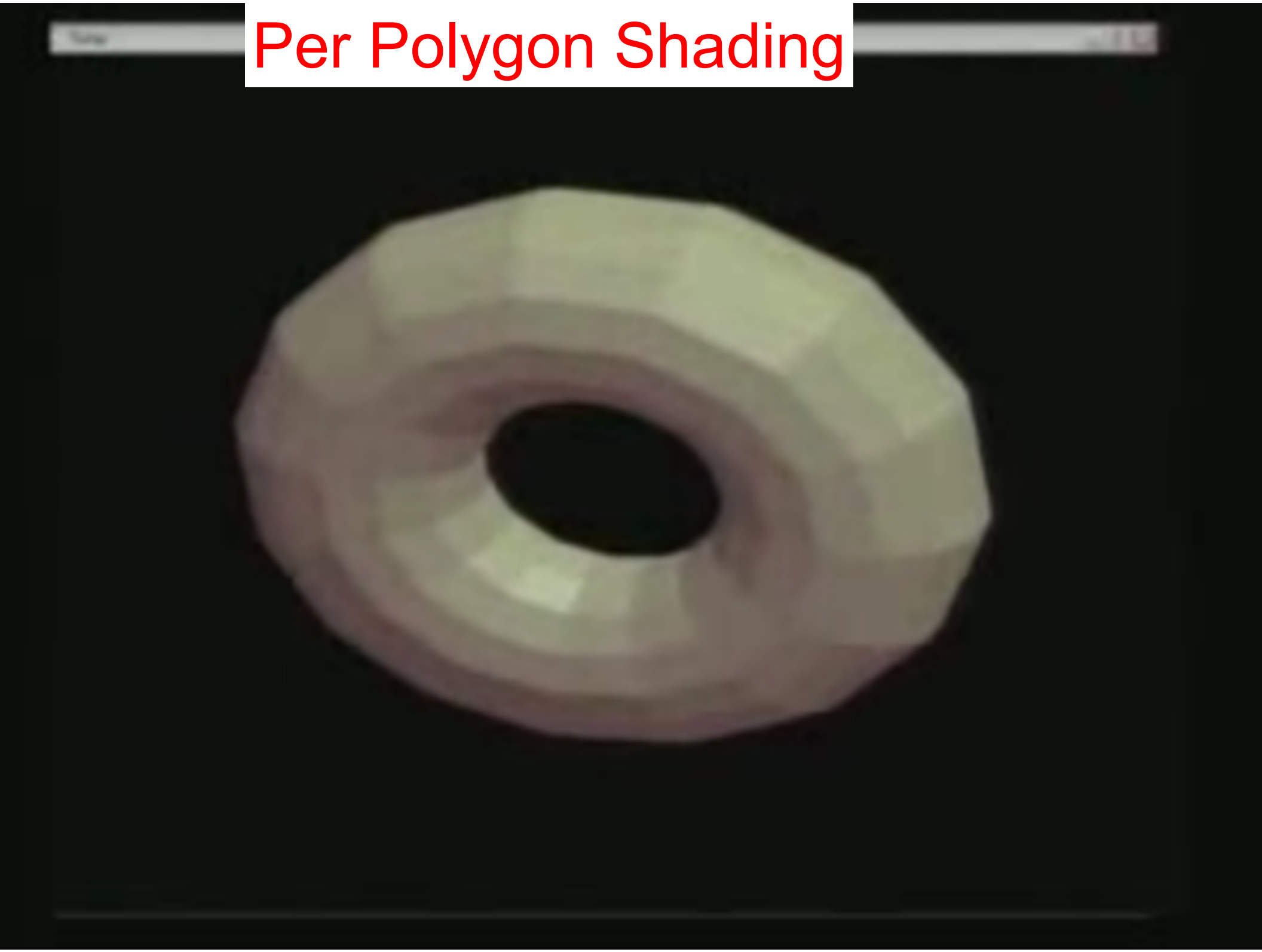
Per Polygon Shading



Smooth Shading

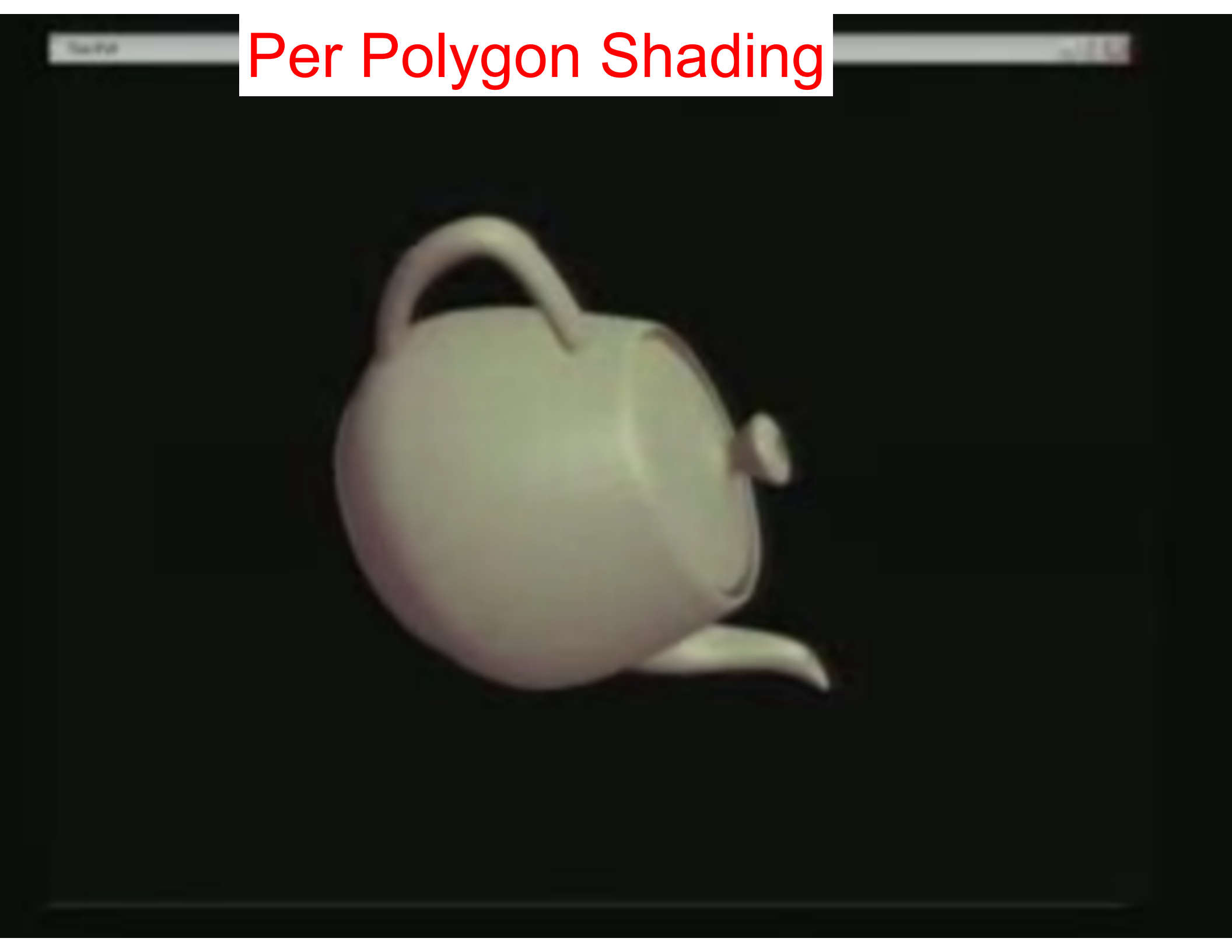


Per Polygon Shading

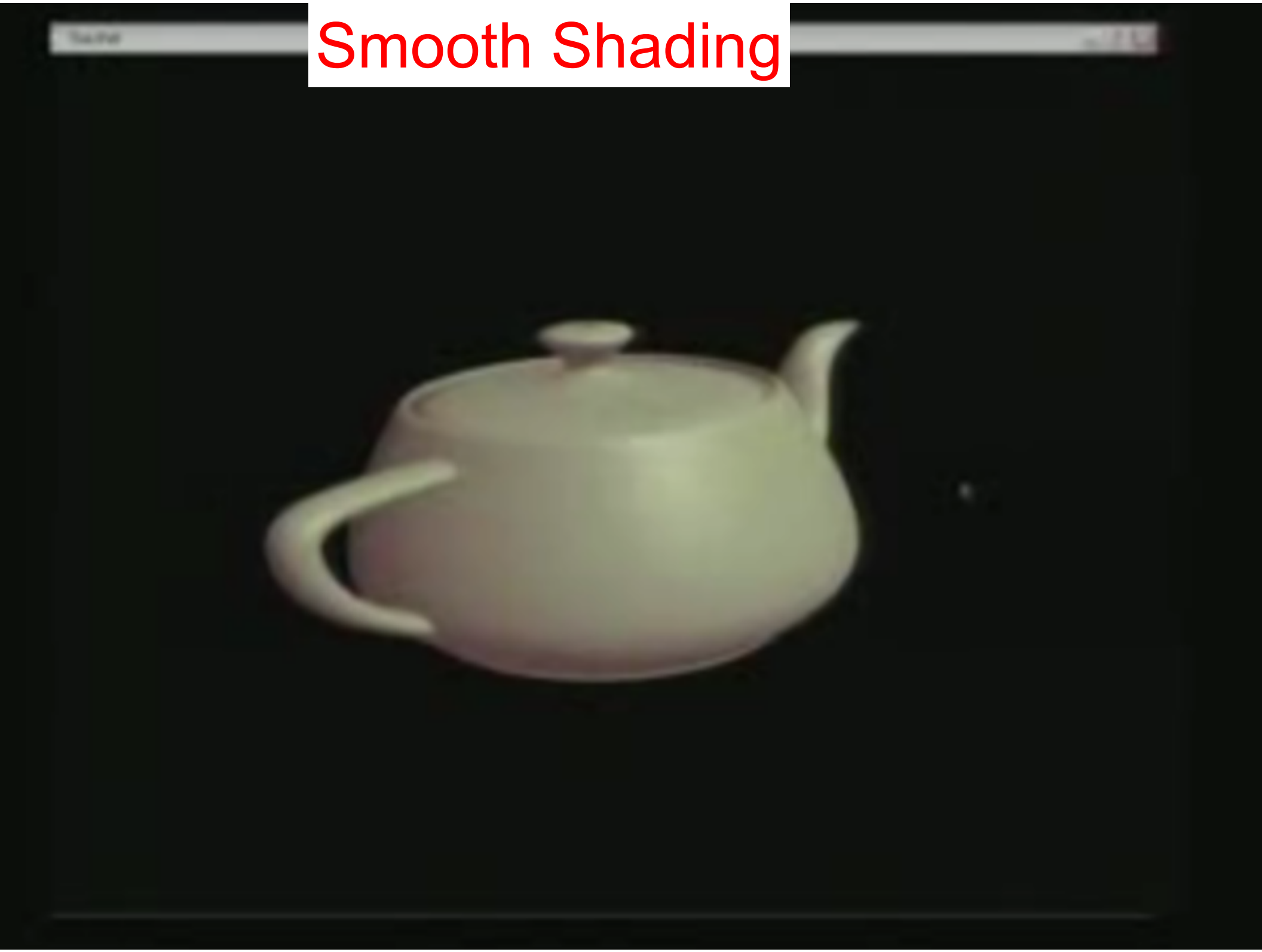


Smooth Shading

Per Polygon Shading

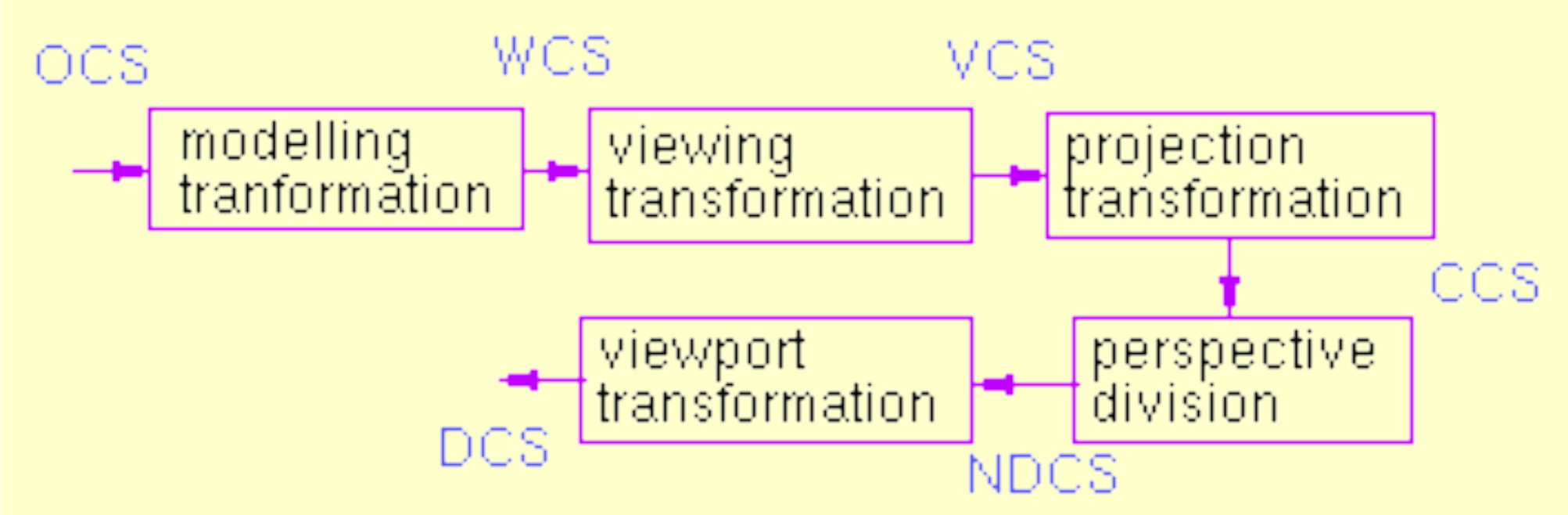


Smooth Shading



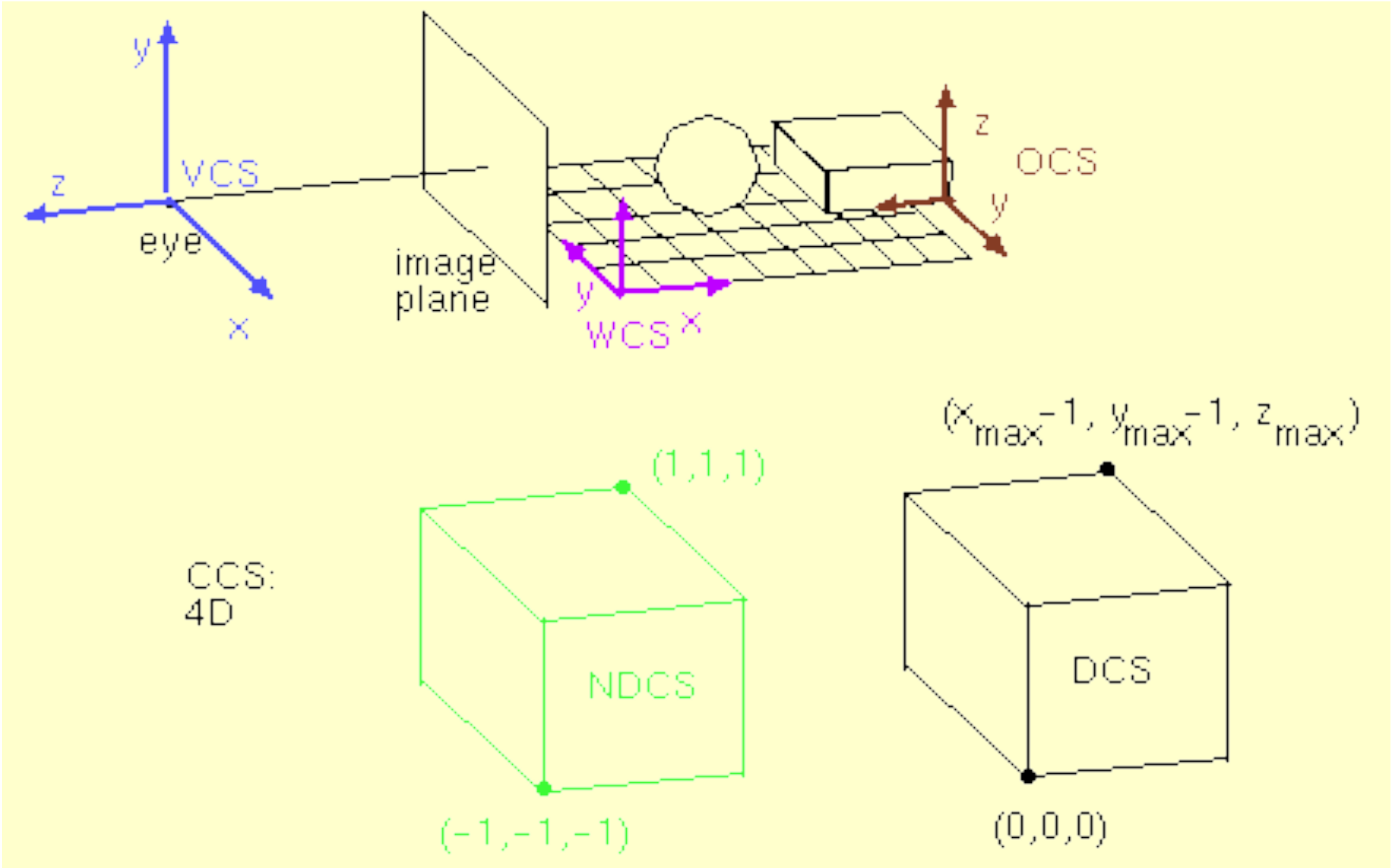
OpenGL Transformations
Chapter 4.10

Stages of Vertex Transformations



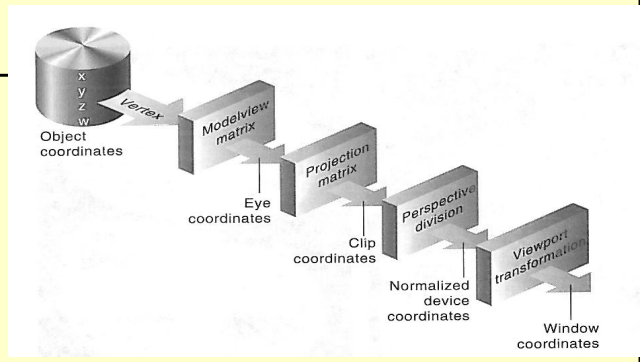
OCS - object coordinate system
WCS - world coordinate system
VCS - viewing coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device coordinate system

Stages of Vertex Transformations



OpenGL functions for setting up transformations

modelling transformation *modelview matrix*	glTranslatef() glRotatef() glScalef()
viewing transformation *modelview matrix*	gluLookAt()
projection transformation *projection matrix*	glFrustum() gluPerspective() glOrtho() gluOrtho2D()
viewing transformation	glViewport()



Objectives

- Introduce **OpenGL matrix modes**
  - Model-View
  - Projection
- Learn how to carry out **transformations in OpenGL**
  - Rotation
  - Translation
  - Scaling

Objectives

Introduce **OpenGL matrix modes**

- Model-view
- Projection

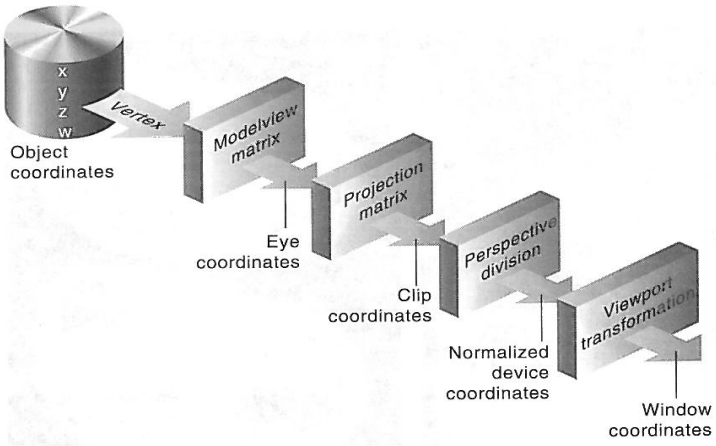
Learn how to carry out **transformations in OpenGL**

- Rotation
- Translation
- Scaling

OpenGL Matrices

- In **OpenGL matrices are part of the State**
- Multiple types
  - Model-View (`GL_MODELVIEW`)
  - Projection (`GL_PROJECTION`)
  - Texture (`GL_TEXTURE`) (ignore for now)
  - Color(`GL_COLOR`) (ignore for now)
- Single set of functions for manipulation
- Select which to manipulated by

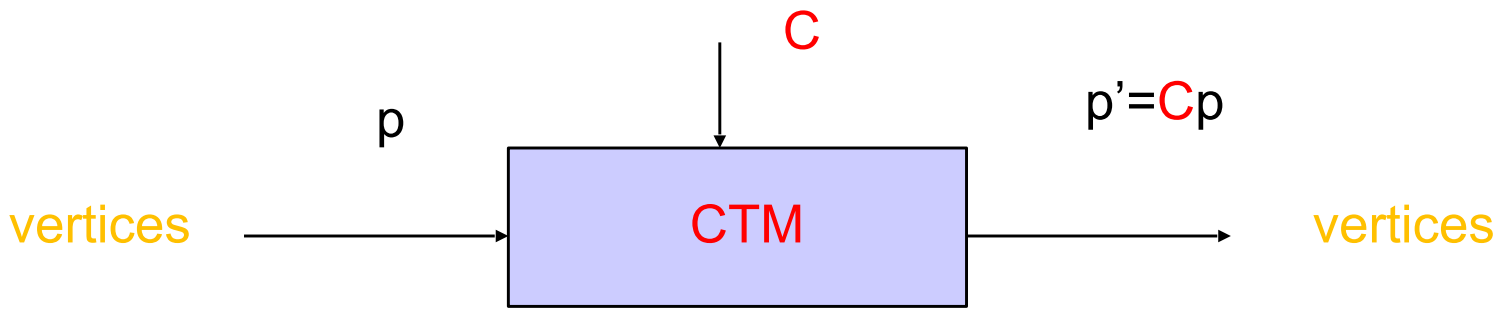
```
glMatrixMode(GL_MODELVIEW);
glMatrixMode(GL_PROJECTION);
```

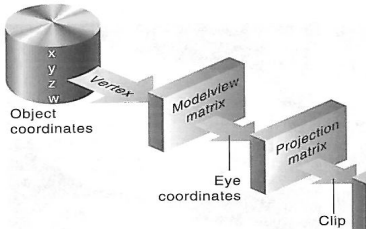


Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 **homogeneous coordinate** matrix, the ***Current Transformation Matrix*** (**CTM**) that is **part of the state** and is **applied to all vertices that pass down the pipeline**

- The **CTM** is defined in the user program and loaded into a transformation unit





CTM operations

The **C** can be altered either by **loading a new CTM** or by **postmutiplication**

Load an identity matrix: C ← I
Load an arbitrary matrix: C ← M

Load a translation matrix: C ← T
Load a rotation matrix: C ← R
Load a scaling matrix: C ← S

Postmultiply by an arbitrary matrix: C ← CM
Postmultiply by a translation matrix: C ← CT
Postmultiply by a rotation matrix: C ← C R
Postmultiply by a scaling matrix: C ← C S

Rotation about a Fixed Point

Start with identity matrix: $C \leftarrow I$

Move fixed point to origin: $C \leftarrow CT$

Rotate: $C \leftarrow CR$

Move fixed point back: $C \leftarrow CT^{-1}$

Result: $C = TRT^{-1}$ which is **backwards**.

This result is a consequence of doing **postmultiplications.**
Let's try again.

Reversing the Order

We want $\mathbf{C} = \mathbf{T}^{-1}\,\mathbf{R}\,\mathbf{T}$

so we must do the operations in the following order

$\mathbf{C} \leftarrow \mathbf{I}$

$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}^{-1}$

$\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$

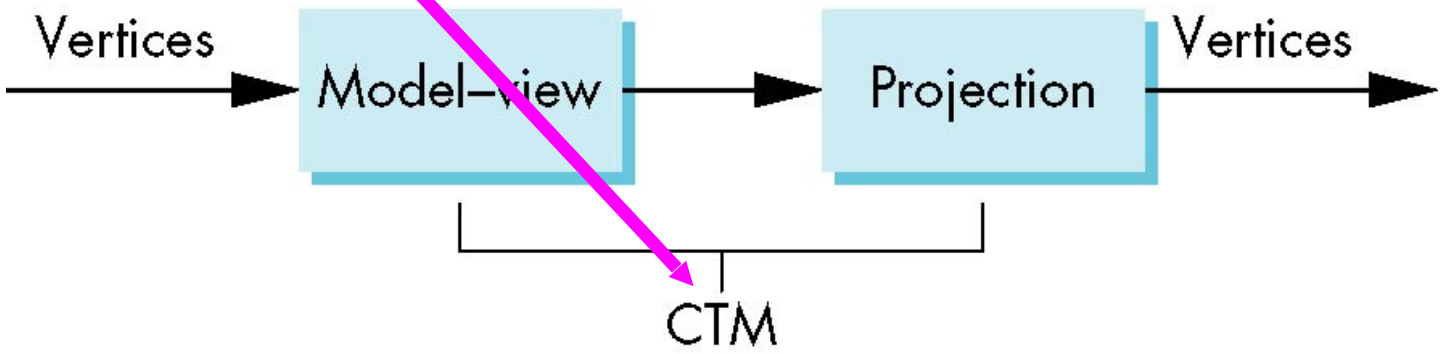
$\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$

Each operation corresponds to one function call in the program

Note that the last operation specified is the first executed in the program

CTM in OpenGL

- OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM
- Can manipulate each by first setting the correct matrix mode



Objectives

Introduce **OpenGL matrix modes**

- Model-view
- Projection

Learn how to carry out **transformations in OpenGL**

- Rotation
- Translation
- Scaling

Rotation, Translation, Scaling

Load an identity matrix:

```
glLoadIdentity()
```

Multiply on right:

```
glRotatef(theta, vx, vy, vz)
```

`theta` in degrees, (`vx, vy, vz`) define axis of rotation

```
glTranslatef(dx, dy, dz)
```

```
glScalef( sx, sy, sz)
```

Each has a float (f) and double (d) format (`glScaled`)

Example

- Rotation about z axis by 30 degrees with a **fixed point** of (1.0, 2.0, 3.0)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(1.0, 2.0, 3.0);
glRotatef(30.0, 0.0, 0.0, 1.0);
glTranslatef(-1.0, -2.0, -3.0);
```

??????

- Remember that last matrix specified in the program is the first applied

Arbitrary Matrices

- Can load and multiply by matrices defined in the application program

```
glLoadMatrixf(m)
glMultMatrixf(m)
```

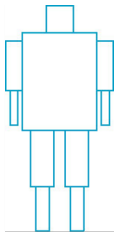
- The matrix **m** is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns

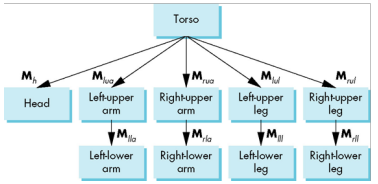
- In **glMultMatrixf**, **m** multiplies the existing matrix on the right

Matrix Stacks

- In many situations we want to save transformation matrices for use later
  - **Traversing hierarchical data structures** (Chapter 10)
  - **Avoiding state changes when executing display lists**
- OpenGL maintains **stacks** for each type of matrix
  - Access present type (as set by **`glMatrixMode()`** by

```
glPushMatrix()
glPopMatrix()
```





Reading Back Matrices

- Can also **access matrices** (and other parts of the **State**) by *query* functions

```
glGetIntegerv
 glGetFloatv
glGetBooleanv
 glGetDoublev
 glIsEnabled
```

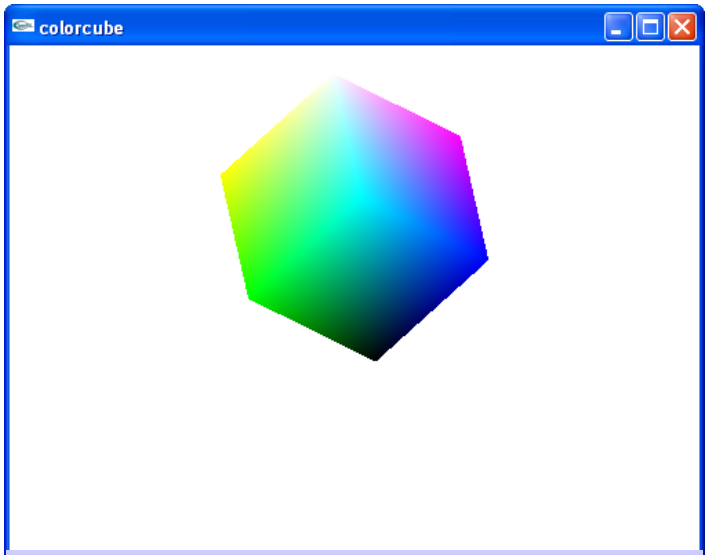
- For matrices, we use as

```
        double m[16];
glGetFloatv(GL_MODELVIEW, m);
```

Using Transformations (see Homework4)

- Example: use **idle** function to rotate a cube and **mouse** function to change direction of rotation

- Start with a program that draws a **cube** (`colorcube.c`) in a standard way
  - **Centered at origin**
  - **Sides aligned with axes**



colorcube.c
Lecture 4 Class Participation

main.c

```
void main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("colorcube");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutIdleFunc(spinCube);
    glutMouseFunc(mouse);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Idle and Mouse callbacks

```
void spinCube()
{
  theta[axis] += 2.0;
  if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
    glutPostRedisplay();
}
```

```
void mouse(int btn, int state, int x, int y)
                        {
  if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
                        axis = 0;
 if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
                        axis = 1;
  if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
                        axis = 2;
                        }
```

Display callback

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    colorcube();
    glutSwapBuffers();
}
```

Note that because of fixed form of callbacks, variables such as `theta` and `axis` must be defined as globals

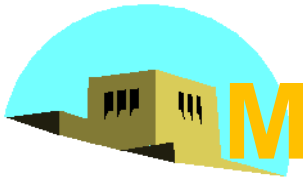
Camera information is in standard reshape callback

Using the Model-View Matrix

- In OpenGL the model-view matrix is used to
  Position the camera
  - Can be done by rotations and translations but is often easier to use `gluLookAt`

  Build models of objects

- The projection matrix is used to define the view volume and to select a camera lens



Model-view and Projection Matrices

Although both are manipulated by the same functions, we have to be careful because incremental changes are always made by postmultiplication

For example, rotating **model-view** and **projection** matrices by the same matrix are not equivalent operations.
Postmultiplication of the model-view matrix is equivalent to **premultiplication of the projection matrix**

Rotation

- We want to use transformations to **move** and **reorient an Object** Problem: find a sequence of **model-view** matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_n$ so that when they are applied successively to **one or more Objects** we see a transition

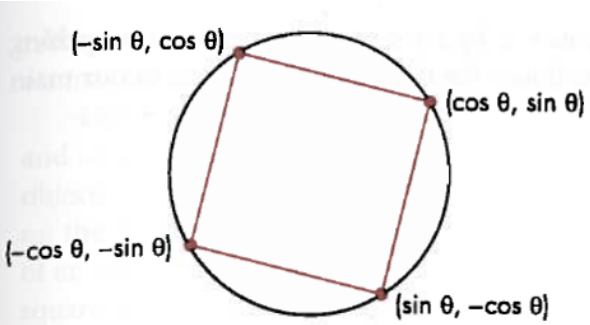
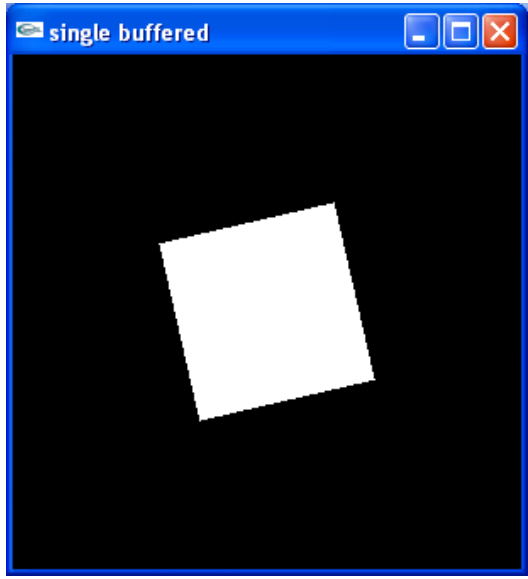


FIGURE 3.20 Square constructed from four points on a circle.

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    thetar = theta/(3.14159/180.0); /* convert degrees to radians */
    glVertex2f(cos(thetar), sin(thetar));
    glVertex2f(-sin(thetar), cos(thetar));
    glVertex2f(-cos(thetar), -sin(thetar));
    glVertex2f(sin(thetar), -cos(thetar));
  glEnd();
}
```



The University of New Mexico

Smooth Rotation

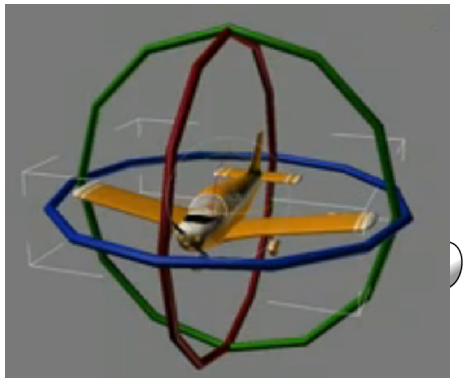
- From a practical standpoint, we are often want to use transformations to **move** and **reorient an object smoothly**

  Problem: find a sequence of **model-view** matrices $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_n$ so that when they are applied successively to **one or more objects** we see a smooth transition

- For **orientating an object**, we can use the fact that **every rotation corresponds to part of a great circle on a sphere**

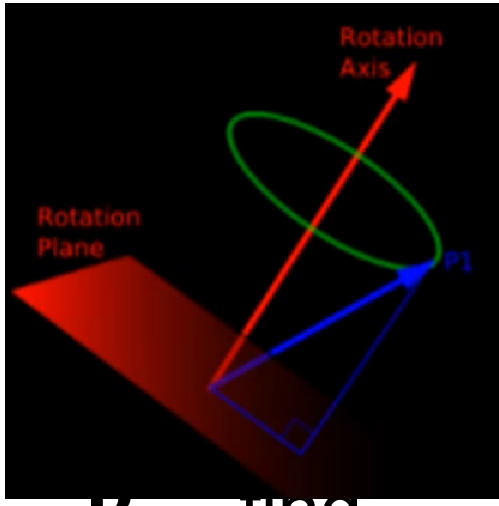
  Find the axis of rotation and angle

  Virtual trackball (see text)



Virtual trackball

Incremental Rotation



- Consider the two approaches

  For a sequence of rotation matrices $\mathbf{R_0}, \mathbf{R_1}, \ldots, \mathbf{R_n}$, find the Euler angles for each and use $\mathbf{R_i} = \mathbf{R_{iz}}\,\mathbf{R_{iy}}\,\mathbf{R_{ix}}$

  - Not very efficient

  Use the final positions to determine the axis and angle of rotation, then increment only that angle

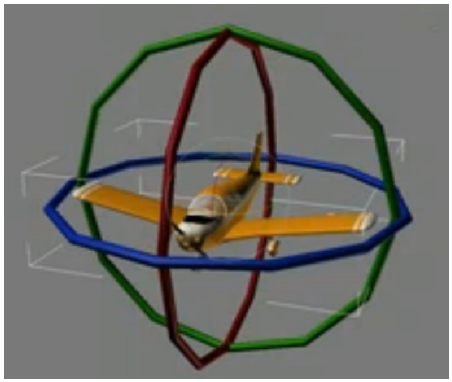
- **Quaternions** can be more efficient than either

Quaternions

- Extension of imaginary numbers **from two to three dimensions**
- Requires **one real and three imaginary components i, j, k**

$$q=q_0+q_1 i+q_2 j+q_3 k$$

- **Quaternions** can express rotations on sphere smoothly and efficiently. Process:

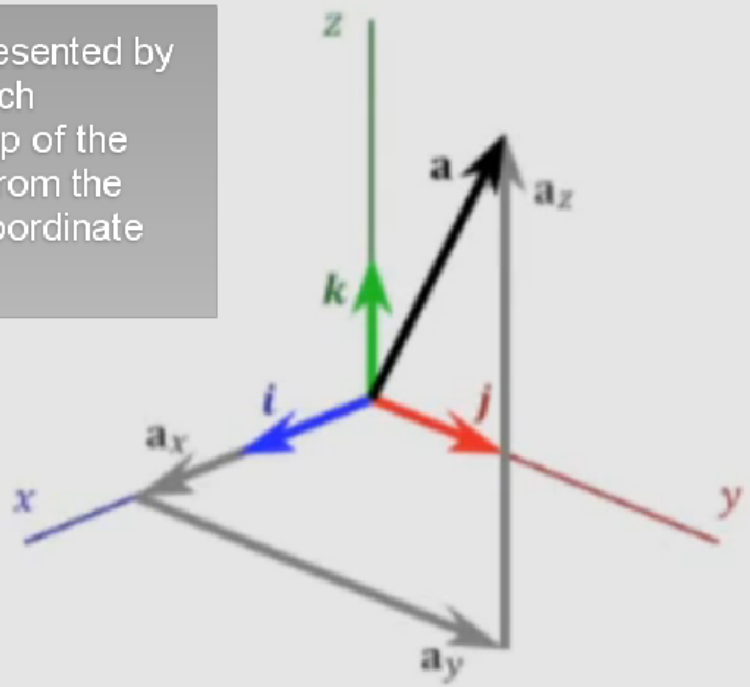


Quaternions (4D object)

The University of New Mexico

$$i^2 = j^2 = k^2 = ijk = -1$$

A vector is represented by an x,y,z set, which represents the tip of the line originating from the origin of your coordinate system.

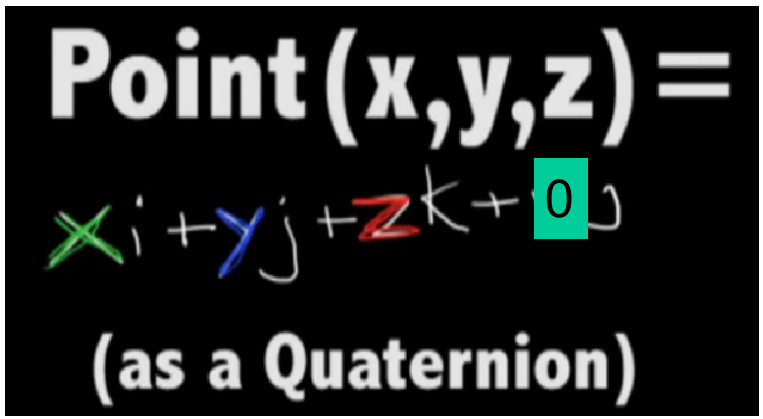


Quaternion Conjugate

The conjugate of a quaternion $a = a_1 + a_2\,i + a_3\,j + a_4\,k$ is defined by

$$\bar{a} = a_1 - a_2\,i - a_3\,j - a_4\,k.$$

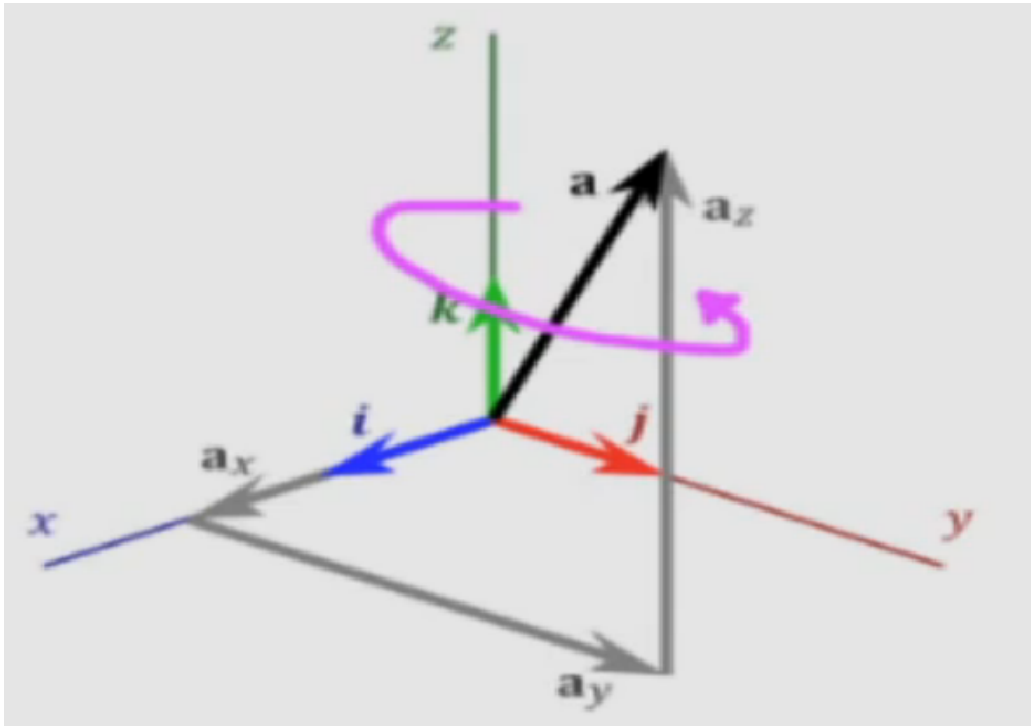
Quaternion Point * Quaternion Rotation Vector = Quaternion Point



The University of New Mexico

Quaternions

$$i^2 = j^2 = k^2 = ijk = -1$$



Point (x,y,z) =

xi + yj + zk + w

(as a Quaternion)

rotationQ = x1i+y1i+z1i+w

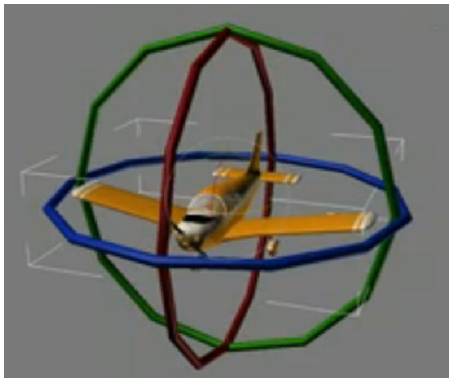
pointQ = x2i+y2i+z2i+w

rotationQ * pointQ = halfway!

halfway * conjugate = answer

Quaternions

- **Quaternions** can express rotations on sphere smoothly and efficiently.



- Process:
  - Model-view matrix → **quaternion**
  - Carry out operations with **quaternions**
  - **Quaternion** → Model-view matriz

(come up with the M components for rotation from the **Quaternion** components!)

WOW!

Building Models
Chapter 4.5

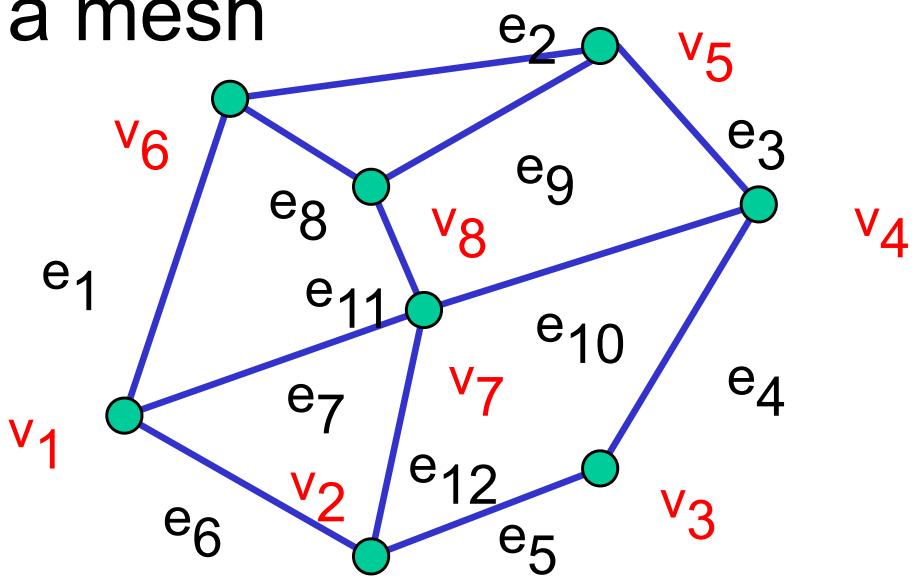
Objectives

- Simple data structures for building polygonal models
  - Vertex lists
  - Edge lists

- OpenGL vertex arrays

Representing a Mesh

- Consider a mesh



- There are 8 nodes and 12 edges
  - 5 interior polygons
  - 6 interior (shared) edges
- **Each vertex** has a location $v_i = (x_i\ y_i\ z_i)$

Simple Representation

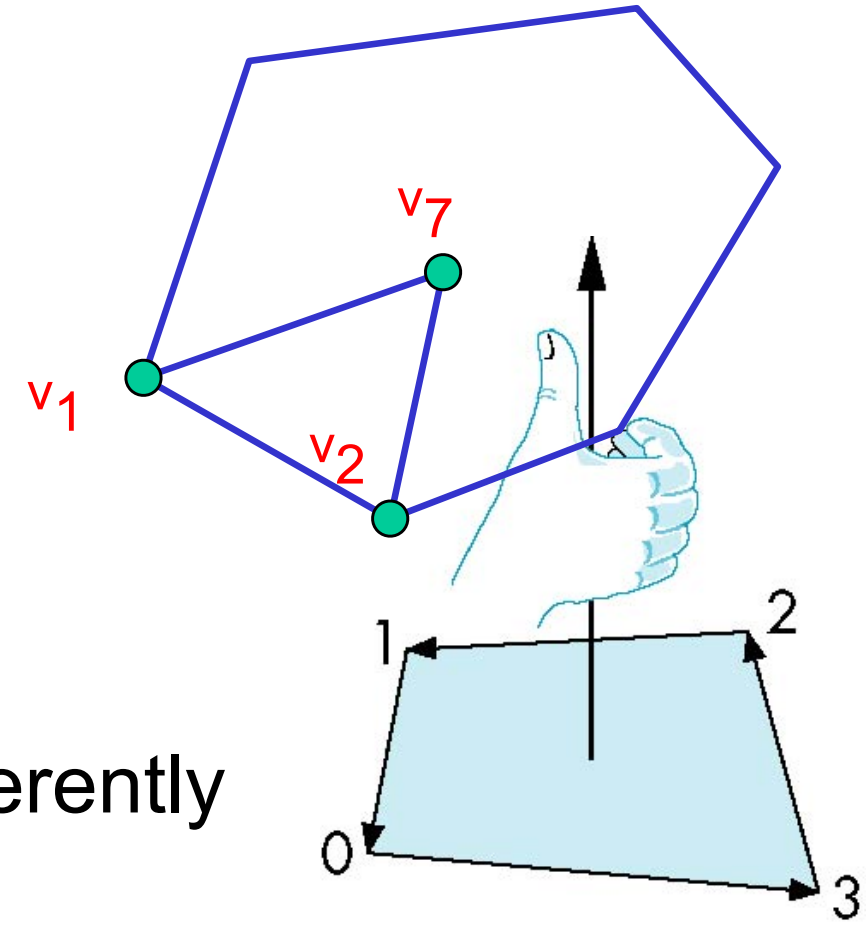
- Define each polygon by the **geometric locations** of its vertices
- Leads to OpenGL code such as

```
glBegin(GL_POLYGON);
 glVertex3f(x1, y1, z1);
 glVertex3f(x6, y6, z6);
 glVertex3f(x7, y7, z7);
      glEnd();
```

- Inefficient and unstructured
  - **Consider moving a vertex to a new location**
  - **Must search for all occurrences**

Inward and Outward Facing Polygons

- The $\{v_1, v_2, v_7\}$ and $\{v_1, v_7, v_2\}$ are equivalent in that the same polygon will be rendered by **OpenGL** but the order $\{v_1, v_7, v_2\}$ is different
- The first describes ***outwardly facing* polygons**
- Use the *right-hand rule* = counter-clockwise encirclement of outward-pointing normal
- **OpenGL** can treat inward and
- ***outwardly facing* polygons** differently



Geometry vs Topology

Generally it is a good idea to look for data structures that separate the geometry from the topology

Geometry: locations of the **vertices**

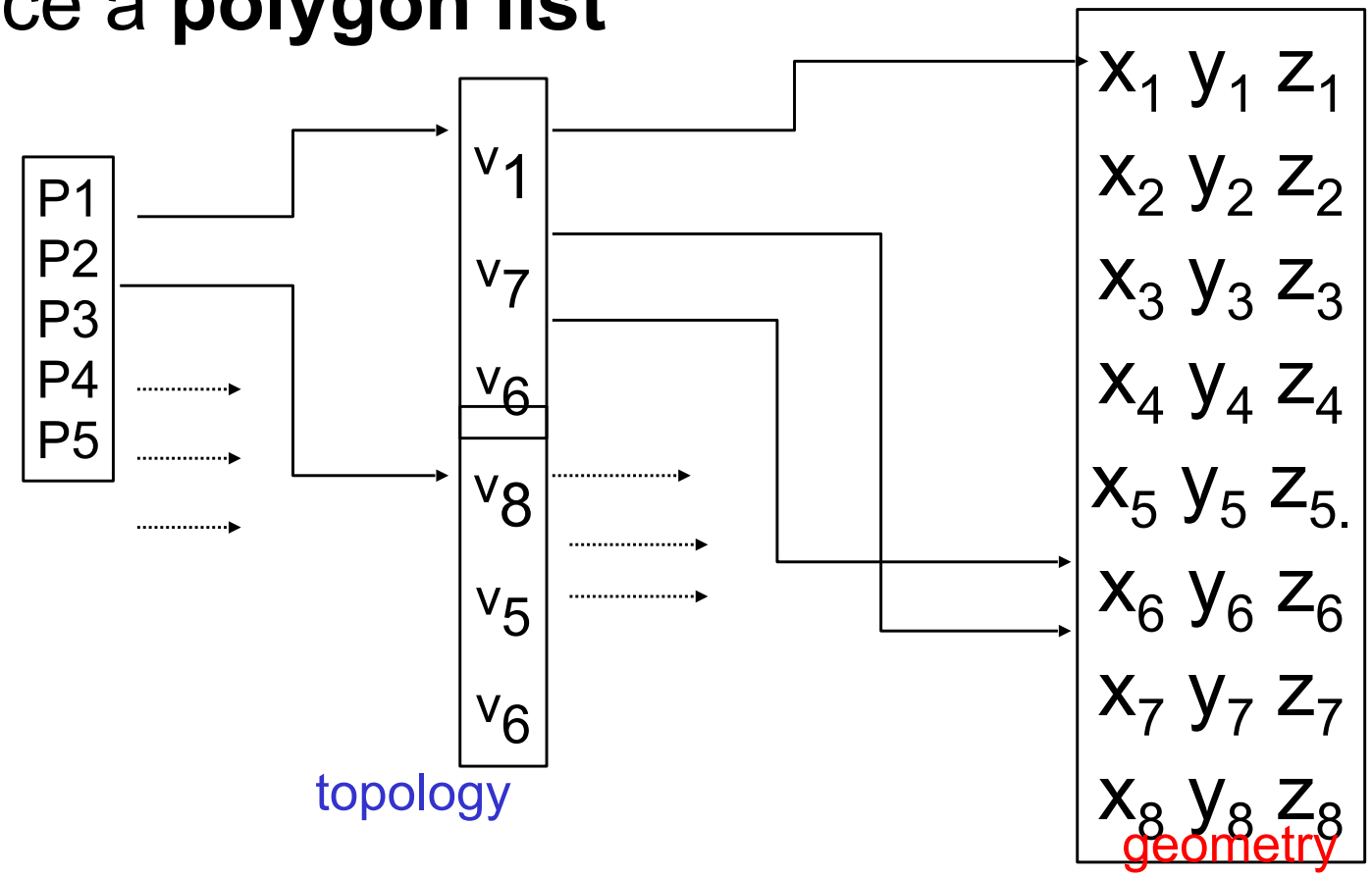
Topology: organization of the **vertices** and edges

Example: a polygon is an ordered list of **vertices** with an edge connecting successive **pairs of vertices** and the last to the first

Topology holds even if geometry changes

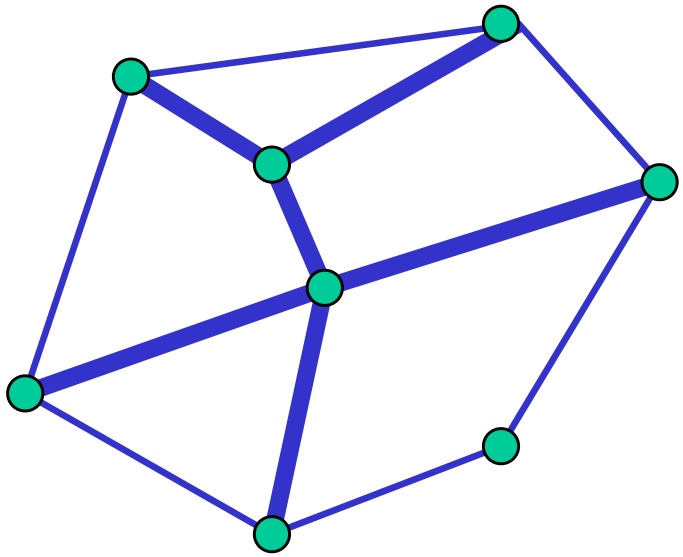
Vertex Lists

- Put the geometry in an array
- Use pointers from the vertices into this array
- Introduce a **polygon list**

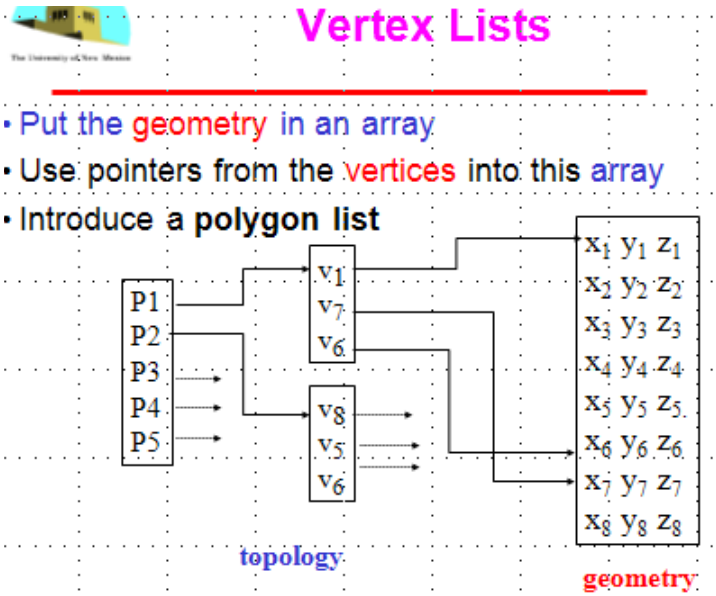


Shared Edges

Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, **shared edges** are drawn twice

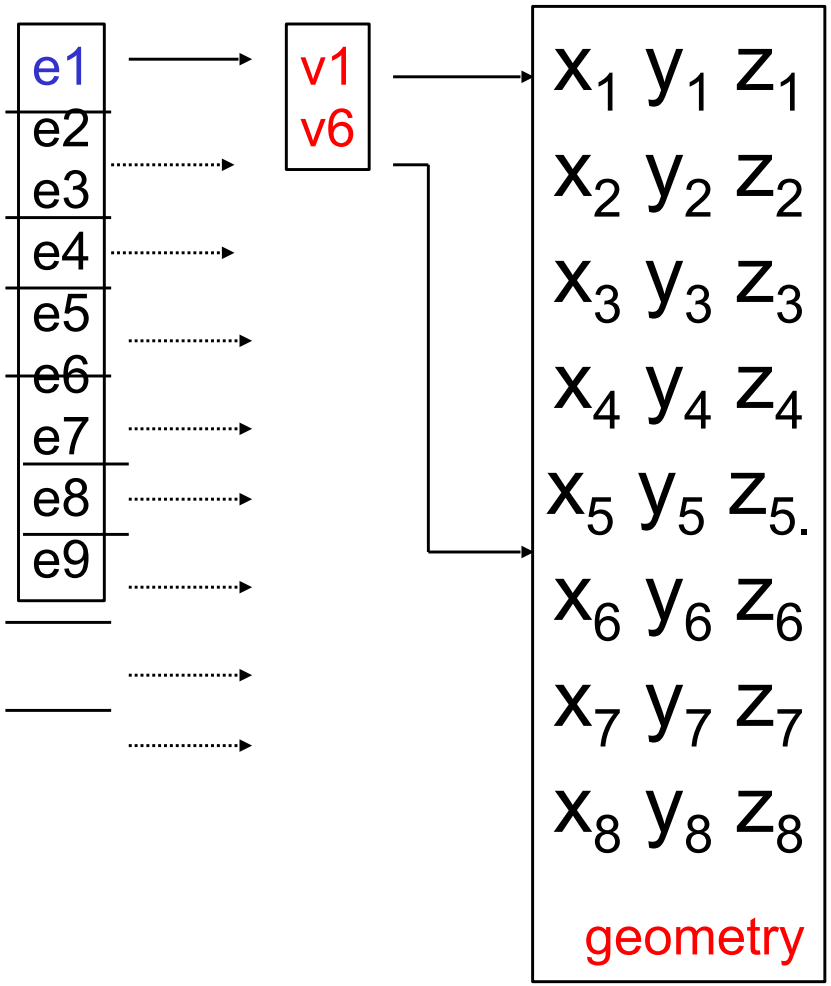


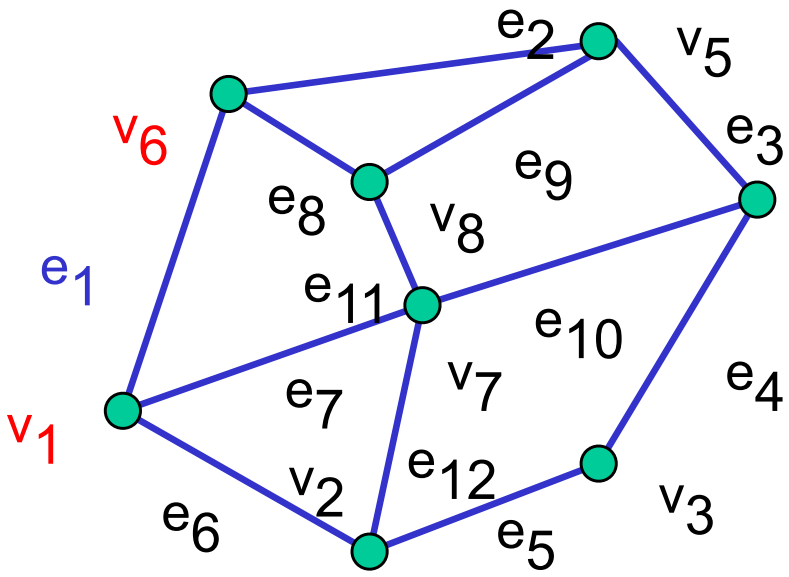
Can store mesh by ***edge list***



The University of New Mexico

Edge List





Note polygons are not represented

Modeling a Cube

Model a color cube for rotating cube program

Define global arrays for vertices and colors

```
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};
```

```
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
   {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
   {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

Drawing a polygon from a list of indices

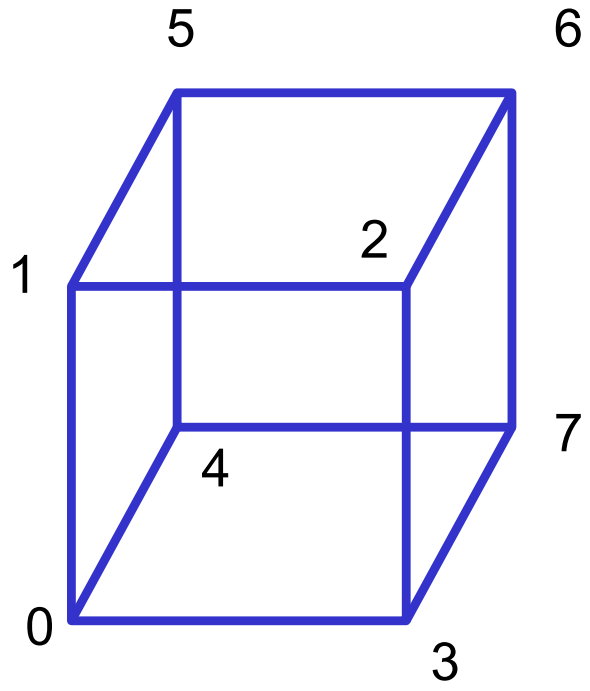
Draw a quadrilateral from a list of indices into the array `vertices` and use color corresponding to first index

```
void polygon(int a, int b, int c , int d)
                    {
                  glBegin(GL_POLYGON);
                   glColor3fv(colors[a]);
                 glVertex3fv(vertices[a]);
                 glVertex3fv(vertices[b]);
                 glVertex3fv(vertices[c]);
                 glVertex3fv(vertices[d]);
                        glEnd();
                          }
```

Draw cube from faces

```
void polygon(int a, int b, int c , int d)
{
   glBegin(GL_POLYGON);
      glColor3fv(colors[a]);
      glVertex3fv(vertices[a]);
      glVertex3fv(vertices[b]);
      glVertex3fv(vertices[c]);
      glVertex3fv(vertices[d]);
    glEnd();
 }
```

```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```



Note that vertices are ordered so that

we obtain correct outwardly facing polygons (normals)

Efficiency

- The weakness of our approach is that we are building the **model** in the application and must do many **function calls** to draw the cube

- **Drawing a cube** by its faces in the most straight forward way requires
  - 6 `glBegin`, 6 `glEnd`
  - 6 `glColor`
  - 24 `glVertex`
  - **More** if we use **texture** and **lighting**

Vertex Arrays

- **OpenGL** provides a facility called ***vertex arrays*** that allows us to store **array** data in the implementation
- Six types of **arrays** supported
  - **Vertices**
  - **Colors**
  - Color indices
  - Normals
  - Texture coordinates
  - Edge flags
- We will need only **Colors** and **Vertices**

Initialization

Using the same color and vertex data, first we enable

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

Identify location of arrays

```
glVertexPointer(3, GL_FLOAT, 0, vertices);
```

3d arrays

stored as floats

data contiguous

data array

```
glColorPointer(3, GL_FLOAT, 0, colors);
```

Mapping indices to faces

```
void colorcube( )
{
    polygon(0,3,2,1);
    polygon(2,3,7,6);
    polygon(0,4,7,3);
    polygon(1,2,6,5);
    polygon(4,5,6,7);
    polygon(0,1,5,4);
}
```

Form an array of face indices

```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6
    0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

Each successive **four indices** describe a **face** of the **cube**

Draw through **glDrawElements** which replaces all **glVertex** and **glColor** calls in the display callback


```
GLubyte cubeIndices[24] = {0,3,2,1,2,3,7,6
0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

Drawing the cube

- Method 1:

what to draw

number of indices

```
for(i=0; i<6; i++)
  glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE,&cubeIndices[4*i])
```

format of index data

start of index data

- Method 2:

```
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
```

Draws cube with 1 function call!!

SUMMARY and NOTES

In this chapter, we have presented two different but ultimately complementary points of view regarding the mathematics of computer graphics. One is that **mathematical abstraction of the objects with which we work in computer graphics is necessary** if we are to understand the operations that we carry out in our programs. The other is that **transformations and the techniques for carrying them out, such as the use of homogeneous coordinates are the basis for implementations of graphics systems**.

Our mathematical tools come from the study of **vector analysis** and **linear algebra**.

We pursued a **coordinate- free approach** for two reasons. First, we wanted to show that all the basic concepts of geometric objects and of transformations are independent of the ways the latter are represented. Second, as object- oriented languages become more prevalent, application programmers will work directly with the objects, instead of with those objects representations.

Homogeneous coordinates provided a wonderful example of the power of mathematical abstraction. By going to an abstract mathematical space the affine space we were able to find a tool that led directly to efficient software and hardware methods.

Finally, we provided the **set of affine transformations supported in OpenGL** and discussed ways that we could concatenate them to provide all affine transformations.

THE END