Computer Graphics
- Transformations in OpenGL

Junjie Cao @ DLUT

Spring 2017

http://jjcao.github.io/ComputerGraphics/

Camera Analogy

- OpenGL coordinate system has different origin (lower-left corner) from the window system (upper-left corner)

- The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera

- The steps with a camera (or a computer) might be the following:
  - Arrange the scene to be photographed into the desired composition (modelling transformation)
  - Set up your tripod and pointing the camera at the scene (viewing transformation).
  - Choose a camera lens or adjust the zoom (projection transformation)
  - Determine how large you want the final photograph to be (viewport transformation)

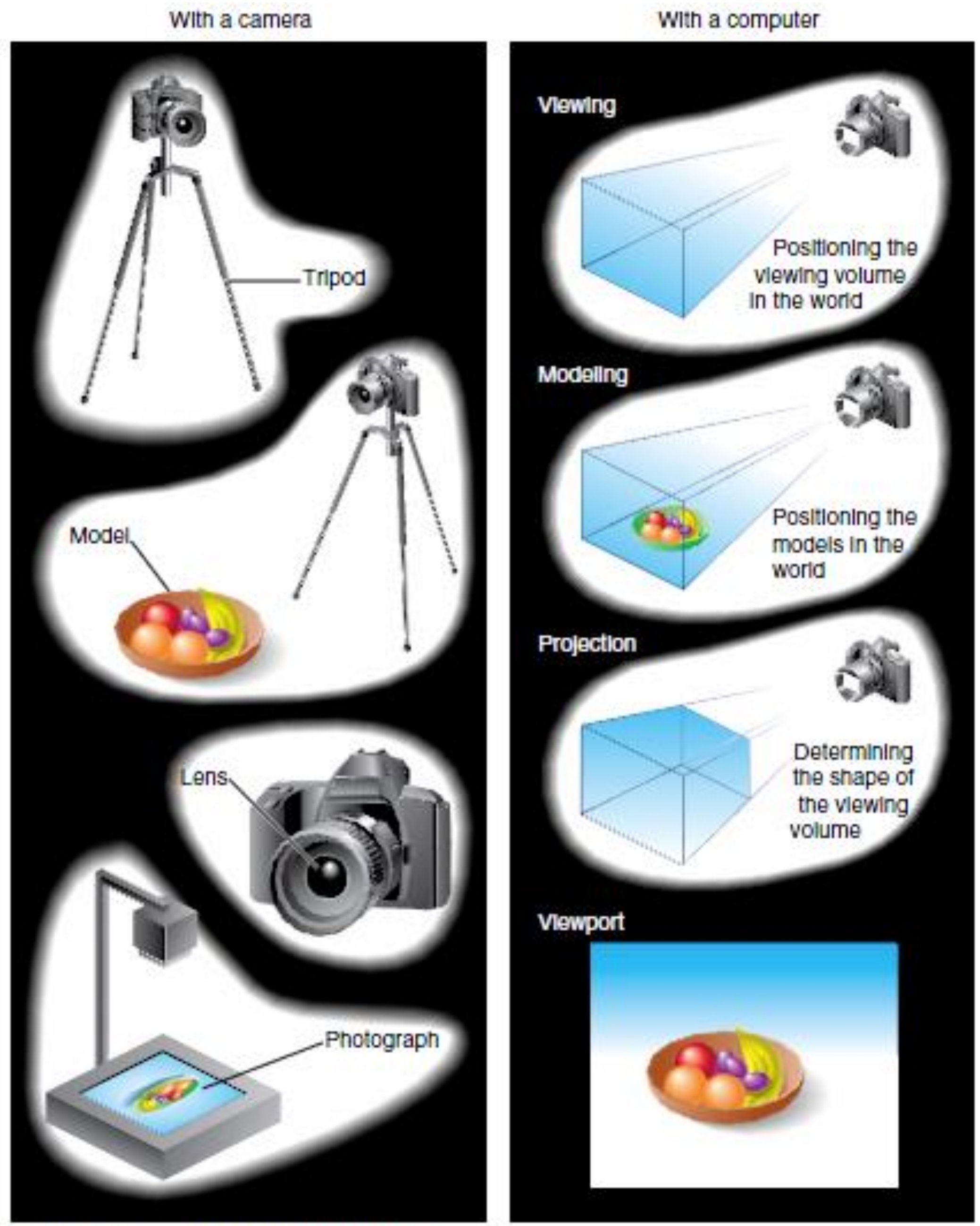
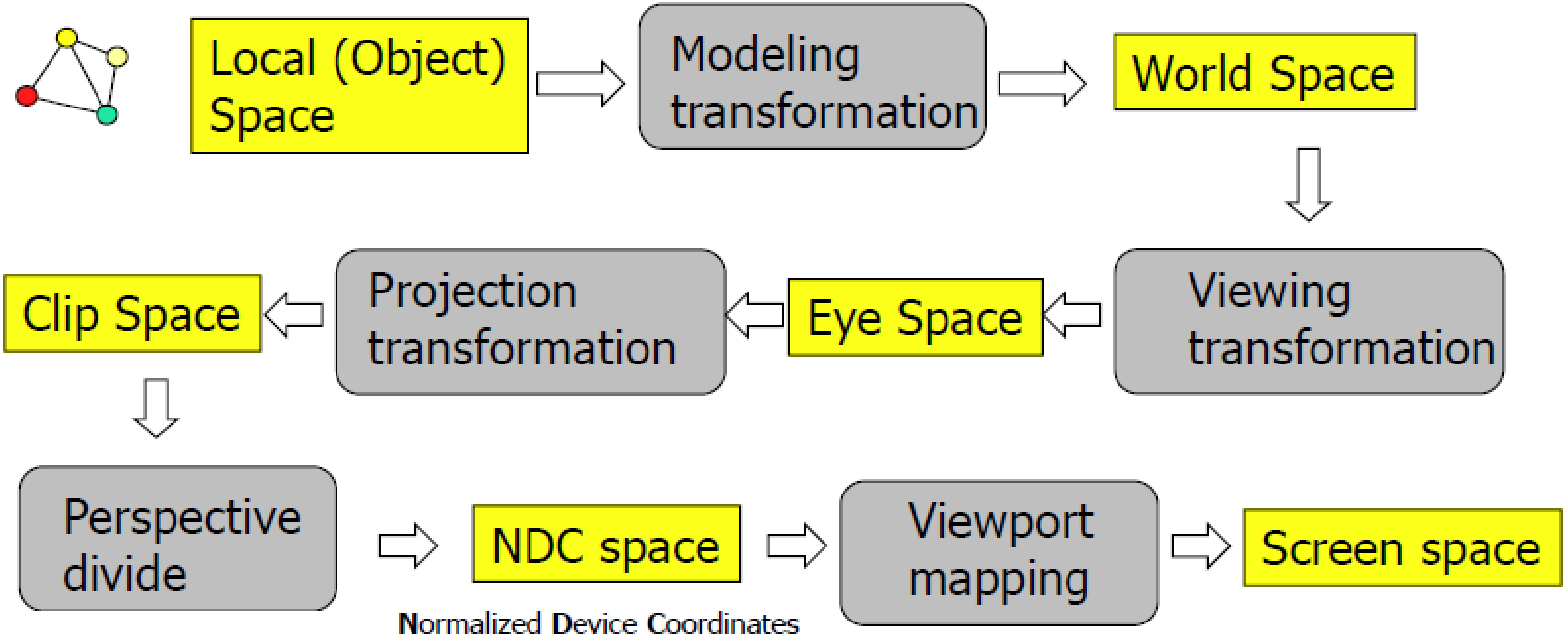


Figure 3-1 The Camera Analogy

Transformation Pipeline



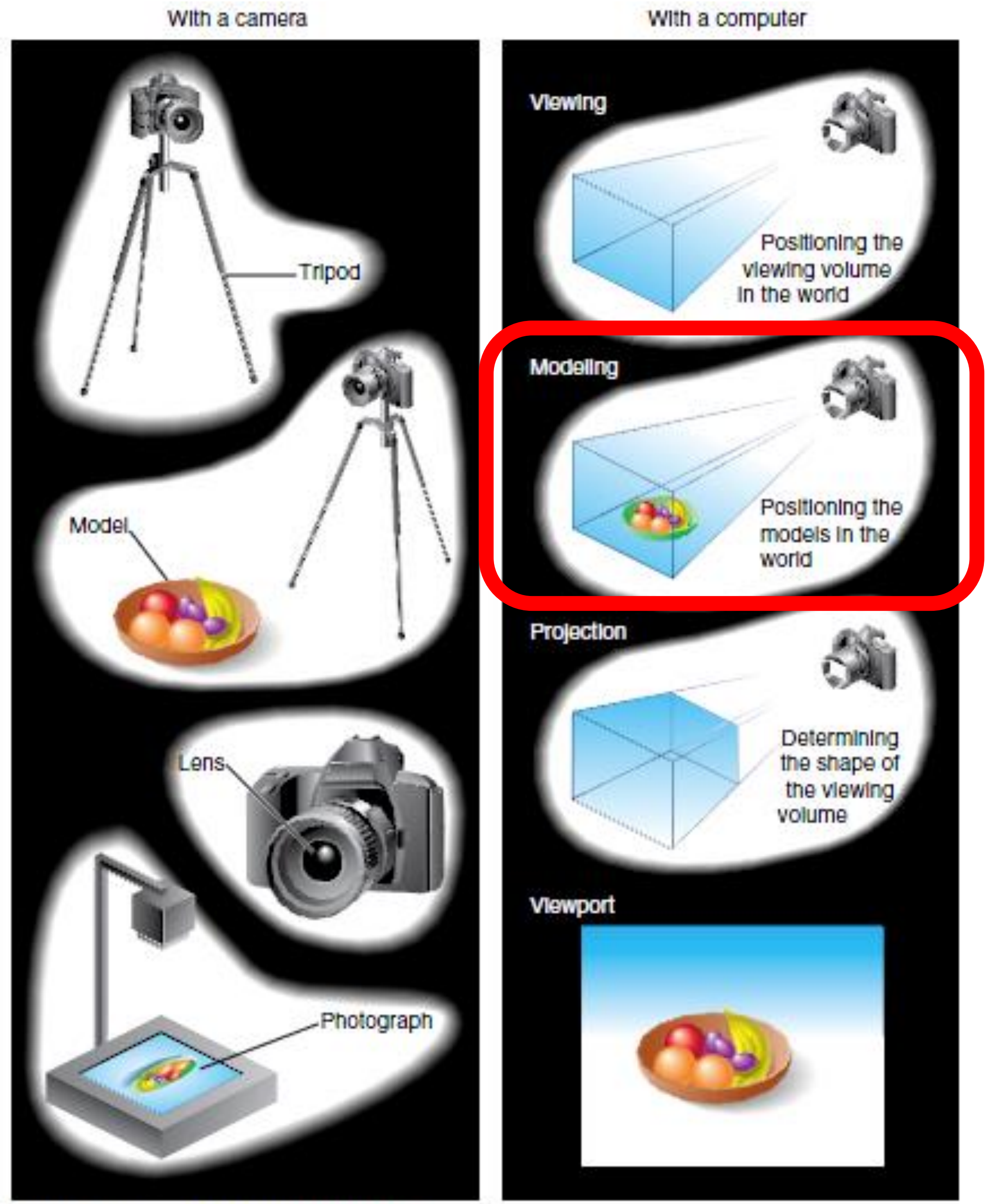
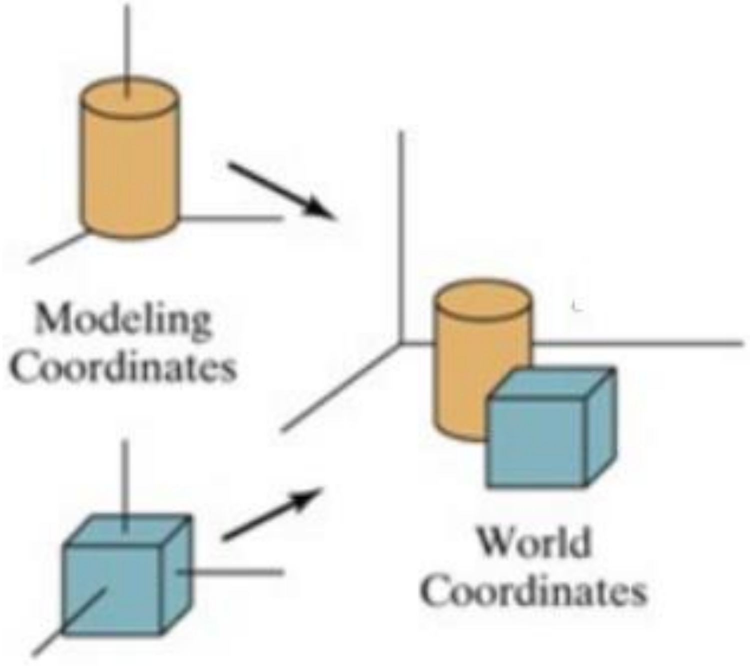


Figure 3-1 The Camera Analogy

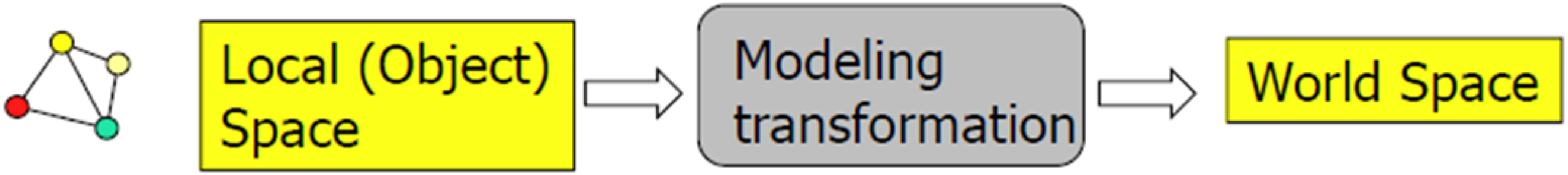
Local Coordinate System

- When you load a file containing a 3d object, its vertices stores coordinates in local CS.
- Assuming obj1, obj2 & obj3 are loaded.
  - Normally, their centers are the origins if they are actually created by code or hand.
  - Sometimes, their centers are not the origins of their local CS respectively if they are results of 3D scanning, etc.
  - Anyway, they are treated as local CS



World Coordinate System

- When the obj is just loaded, its local CS is used as WCS.
- To place multiple objs in your WCS, you need specify position, size, orientation of them
- Transformations need to be performed to position the object in WCS



- A modeling transformation is a sequence of translations, rotations, scalings (in arbitrary order) matrices multiplied together

$$\begin{aligned} \mathbf{x}' &= \mathbf{m_{11}x} + \mathbf{m_{12}y} + \mathbf{m_{13}z} \\ \mathbf{y}' &= \mathbf{m_{21}x} + \mathbf{m_{22}y} + \mathbf{m_{23}z} \\ \mathbf{z}' &= \mathbf{m_{31}x} + \mathbf{m_{32}y} + \mathbf{m_{33}z} \end{aligned}$$

or

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Modeling transformation matrix

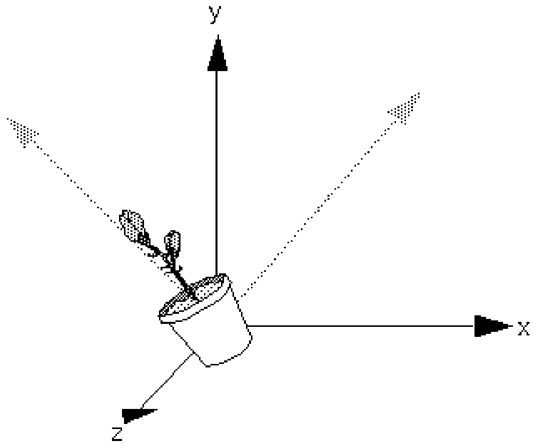
Modeling Transformations

- The three OpenGL routines for modeling transformations are:
  - glTranslate*(),
  - glScale*()
  - void **glRotate**{fd}(TYPE *angle*, TYPE *x*, TYPE *y*, TYPE *z*);
  - **glRotatef(**45.0, 0.0, 0.0, 1.0**)**

  deprecated

- These routines **transform an object (or coordinate system**, if you're thinking of it that way) by moving, rotating, stretching, shrinking, or reflecting it

- All three commands are **equivalent** to producing an appropriate translation, rotation, or scaling **matrix**, and then calling **glMultMatrix***() with that matrix as the argument



- OpenGL **automatically** computes the matrices for you

Modeling Transformations

- Each of these postmultiplies the *current matrix*
  - E.g., if current matrix is **C**, then **C=CS**

- The current matrix is either the **modelview** matrix or the projection matrix (also a texture matrix, won't discuss)
  - Set these with glMatrixMode(), e.g.:

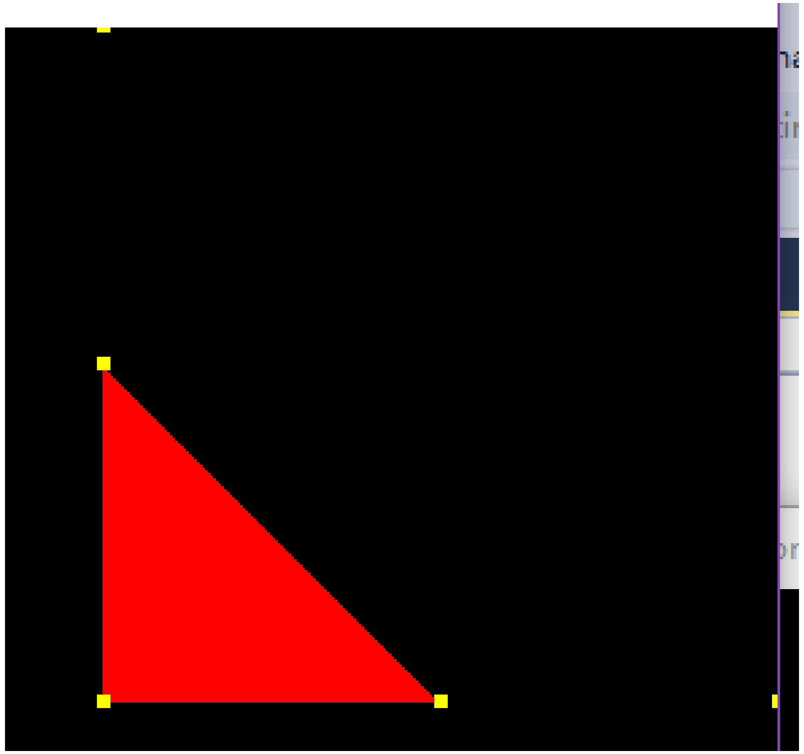
    glMatrixMode(GL_MODELVIEW);

    glMatrixMode(GL_PROJECTION);

- **WARNING: common mistake ahead!**
  - Be sure that you are in **GL_MODELVIEW** mode before making modeling or viewing calls!
  - Ugly mistake because it can appear to work, at least for a while…, see https://sjbaker.org/steve/omniv/projection_abuse.html

Example for Modeling Transformation 1

```
void display() {
glClear(GL_COLOR_BUFFER_BIT);
glColor4f(1,1,0,1); //glColor* have been deprecated in OpenGL 3

// draw triangle 1
glBegin(GL_TRIANGLES);
glColor4f(1.0,0.0,0.0,1.0);glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0); glVertex3f(0.0, 1.0, -10.0);
glEnd();
…
```



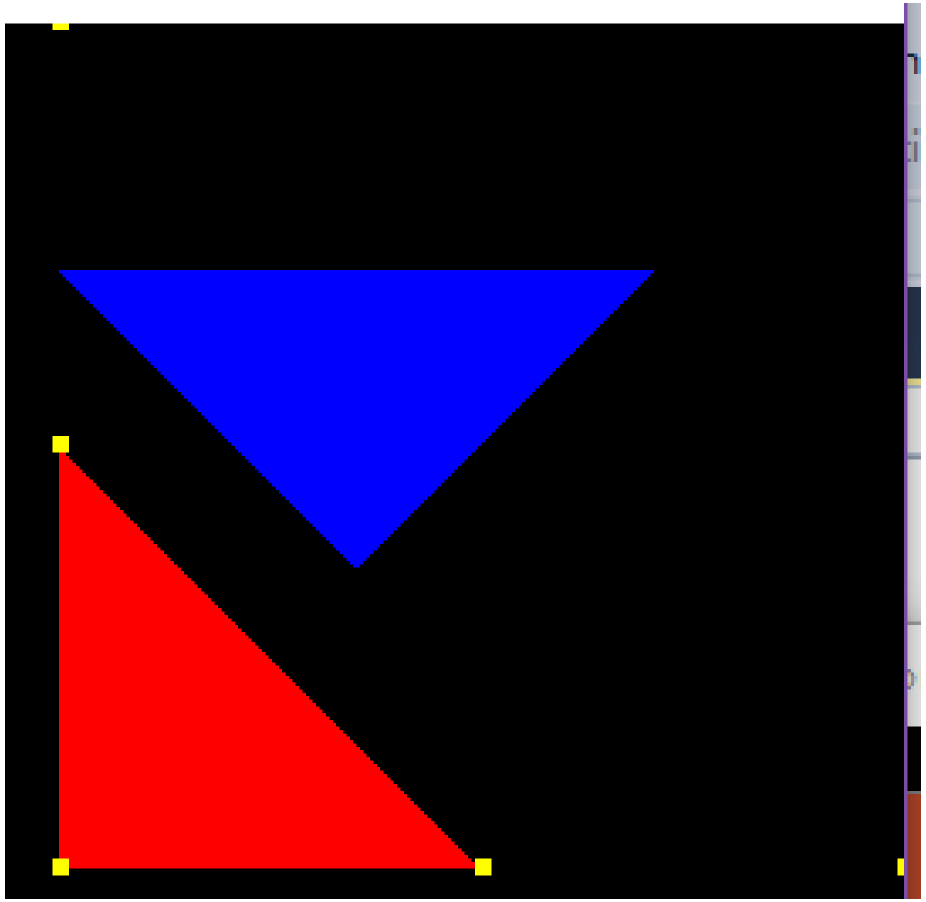
Example for Modeling Transformation 2

```
// draw triangle 3
glMatrixMode(GL_MODELVIEW);
glPushMatrix(); glLoadIdentity(); //More details will be explained
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glBegin(GL_TRIANGLES);
glColor4f(0.0, 1.0, 0.0, 1.0);glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0);glVertex3f(0.0, 1.0, -10.0);
glEnd();
glPopMatrix();

glutSwapBuffers();
}
```

Could you draw the two triangles on some paper?

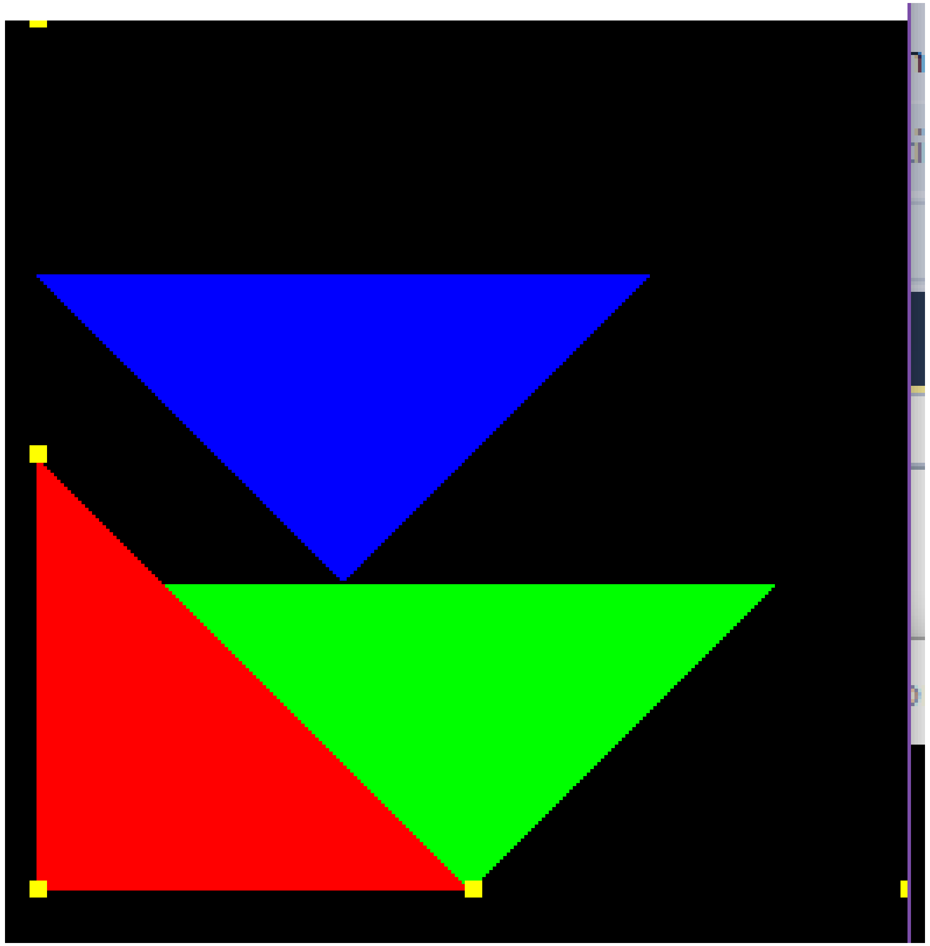
Example for Modeling Transformation 2

```
// draw triangle 3
glMatrixMode(GL_MODELVIEW);
glPushMatrix(); glLoadIdentity();
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glBegin(GL_TRIANGLES);
glColor4f(0.0, 1.0, 0.0, 1.0);
glVertex3f(0.0, 0.0, -10.0);
glVertex3f(1.0, 0.0, -10.0);
glVertex3f(0.0, 1.0, -10.0);
glEnd();
glPopMatrix();
glutSwapBuffers();
}
```

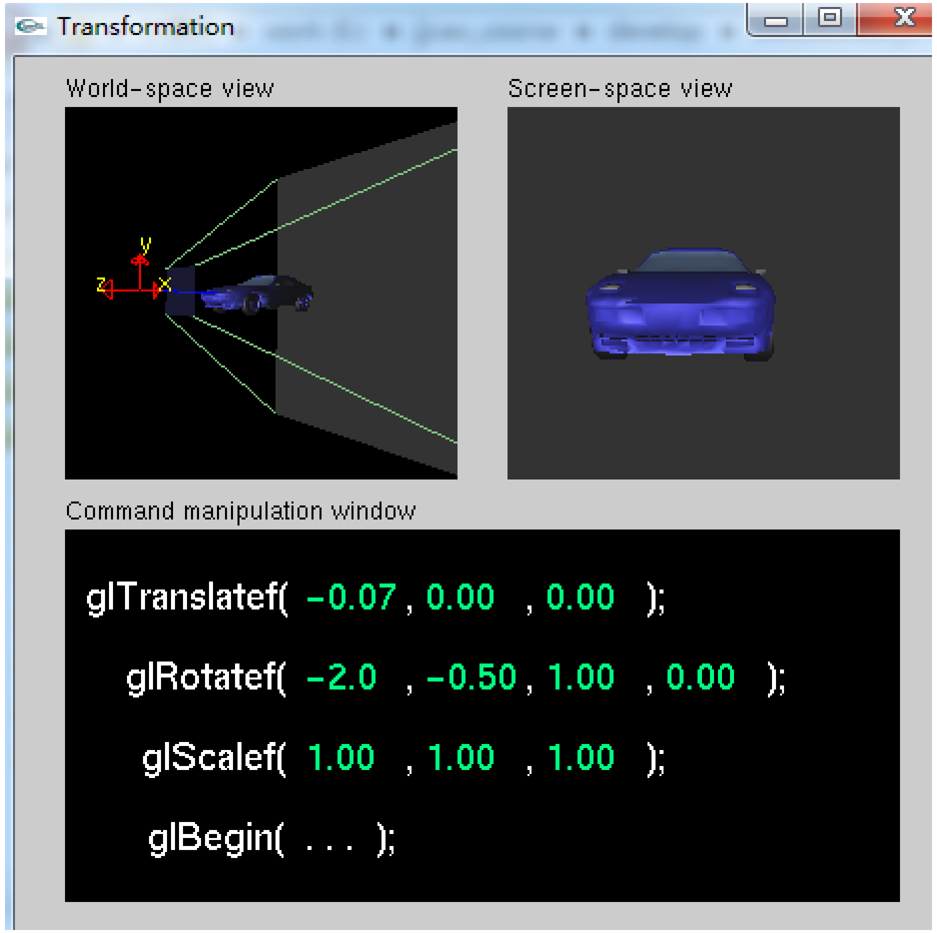


Example for Modeling Transformation 2

```
// draw triangle 2
glPushMatrix(); glLoadIdentity();
glTranslatef(1, 0, 0);
glRotatef(45, 0, 0, 1);
glColor4f(0.0, 1.0, 0.0, 1.0);
glBegin(GL_TRIANGLES); … glEnd();
glPopMatrix();

// draw triangle 3
glPushMatrix(); glLoadIdentity();
glRotatef(45, 0, 0, 1);
glTranslatef(1, 0, 0);
glColor4f(0.0, 1.0, 0.0, 1.0);
glBegin(GL_TRIANGLES);  … glEnd();
glPopMatrix();
```

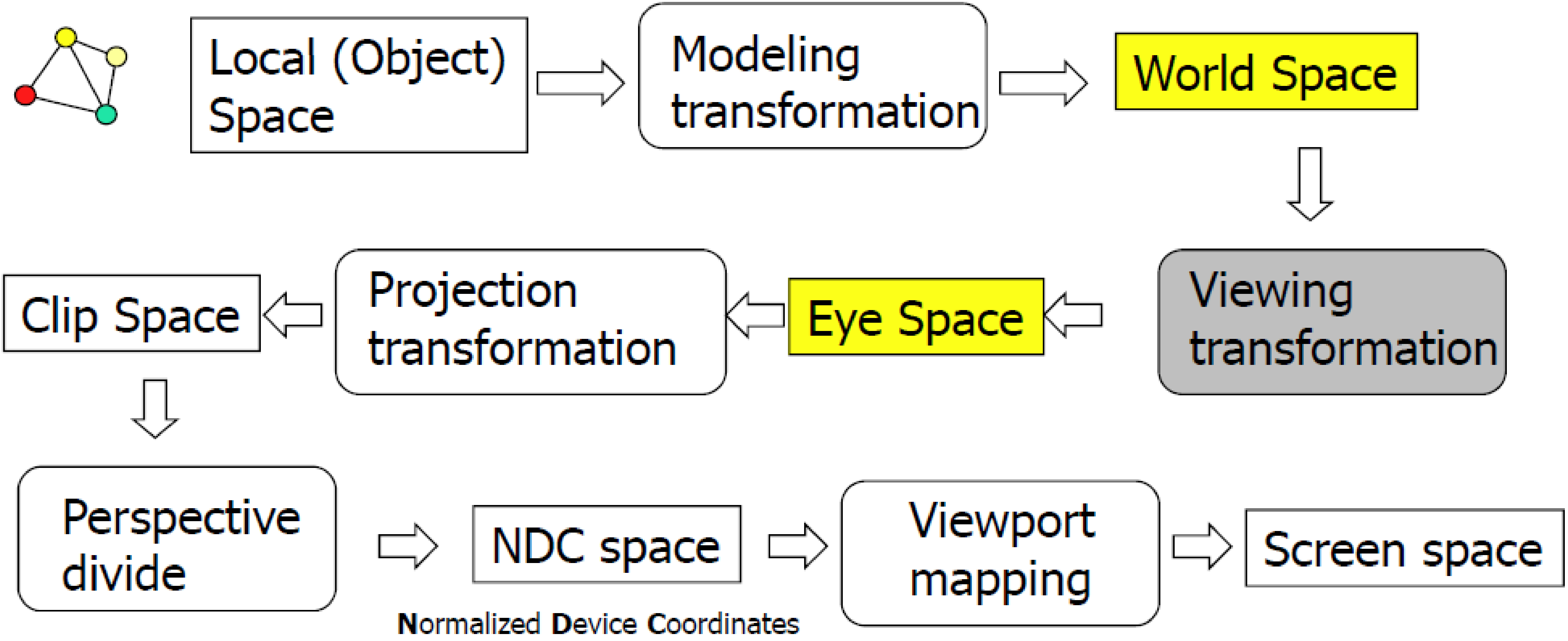


Modeling Transformations (cont)



Nate_Robins_tutorials: Transformation

Viewing transformation



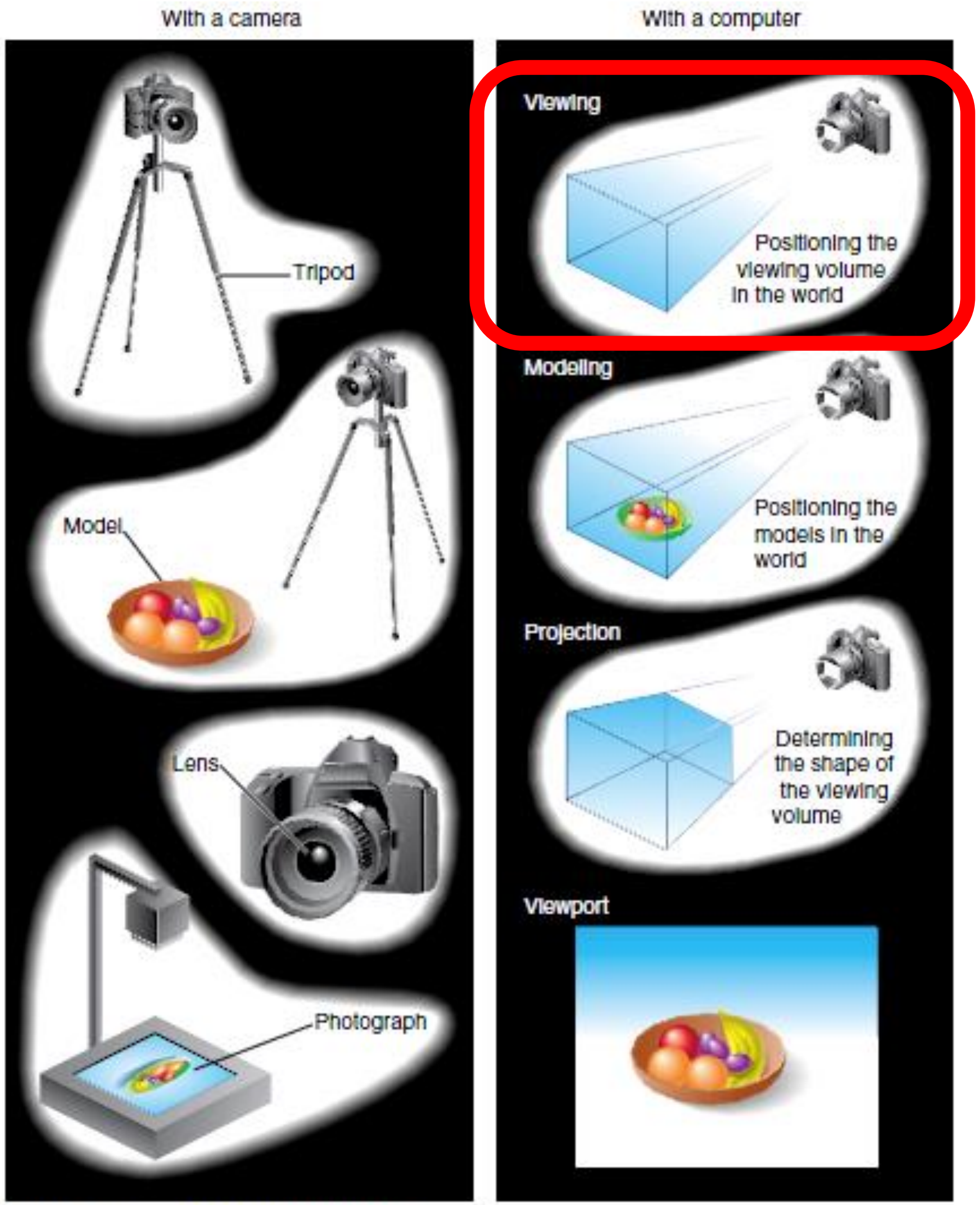


Figure 3-1 The Camera Analogy

Viewing Transformation

- Convert from WCS to the camera (eye) coordinate sys
- The camera position is the origin initially.
- The objs are also in the origin mostly. Or have been placed well in WCS
- Anyway, we need move the camera to see what we wish to see (may see nothing using default camera/viewing transformation)

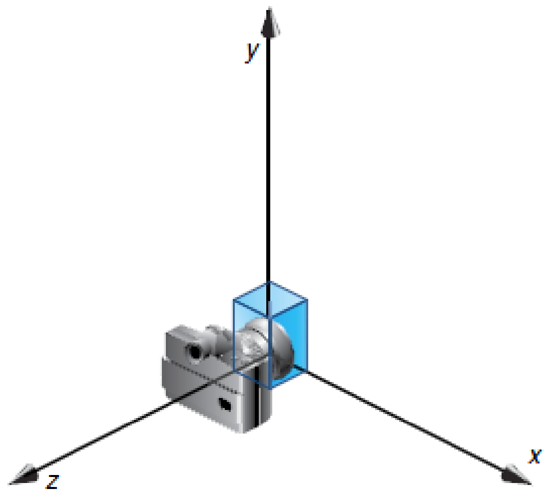
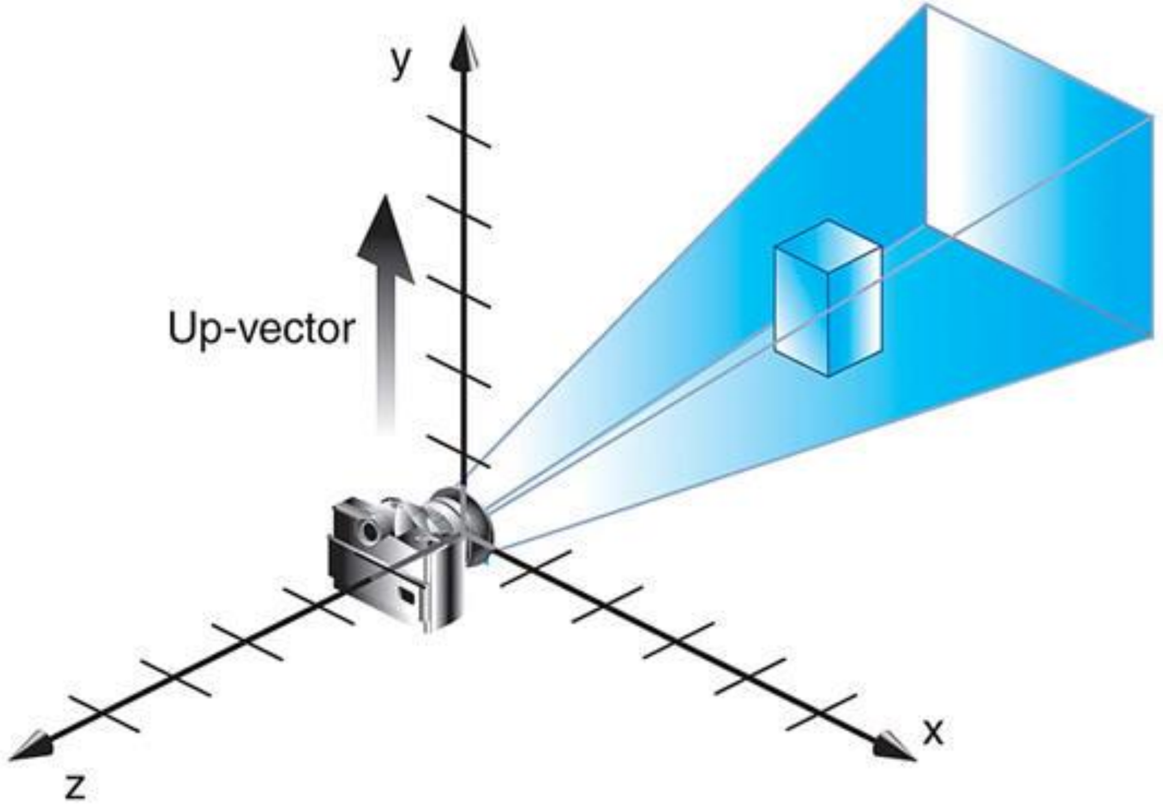
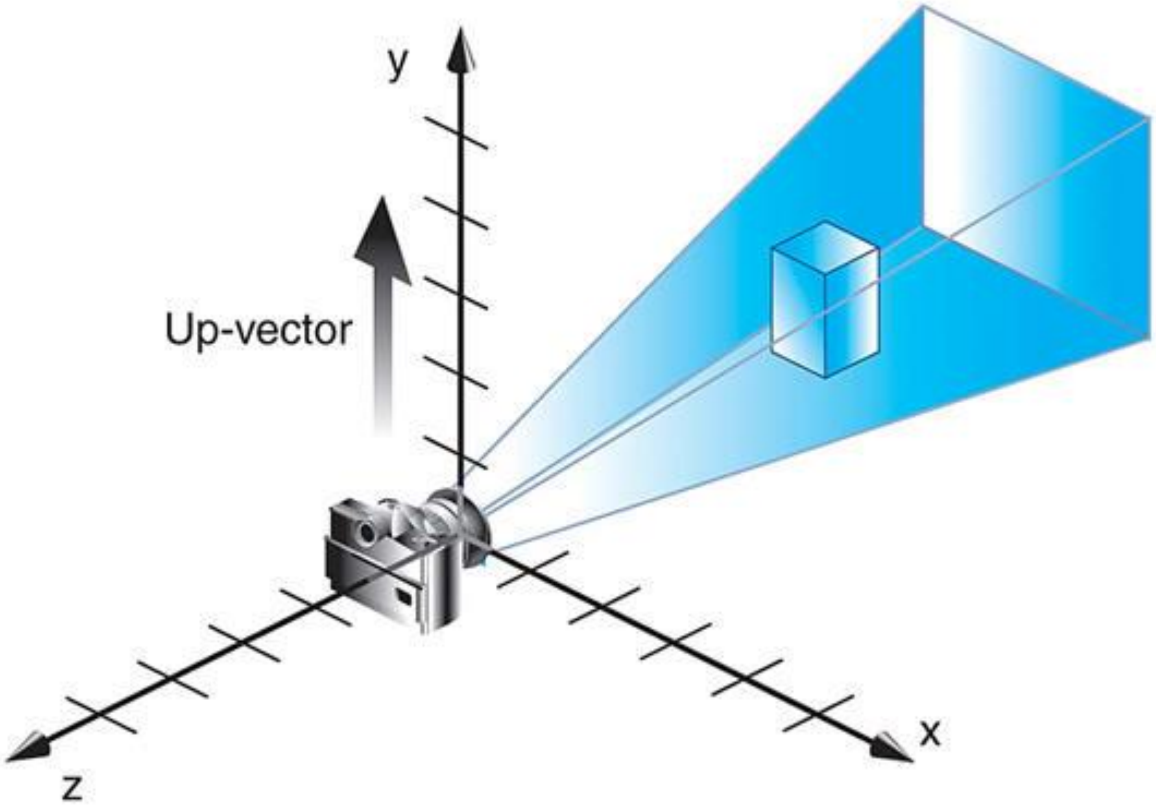


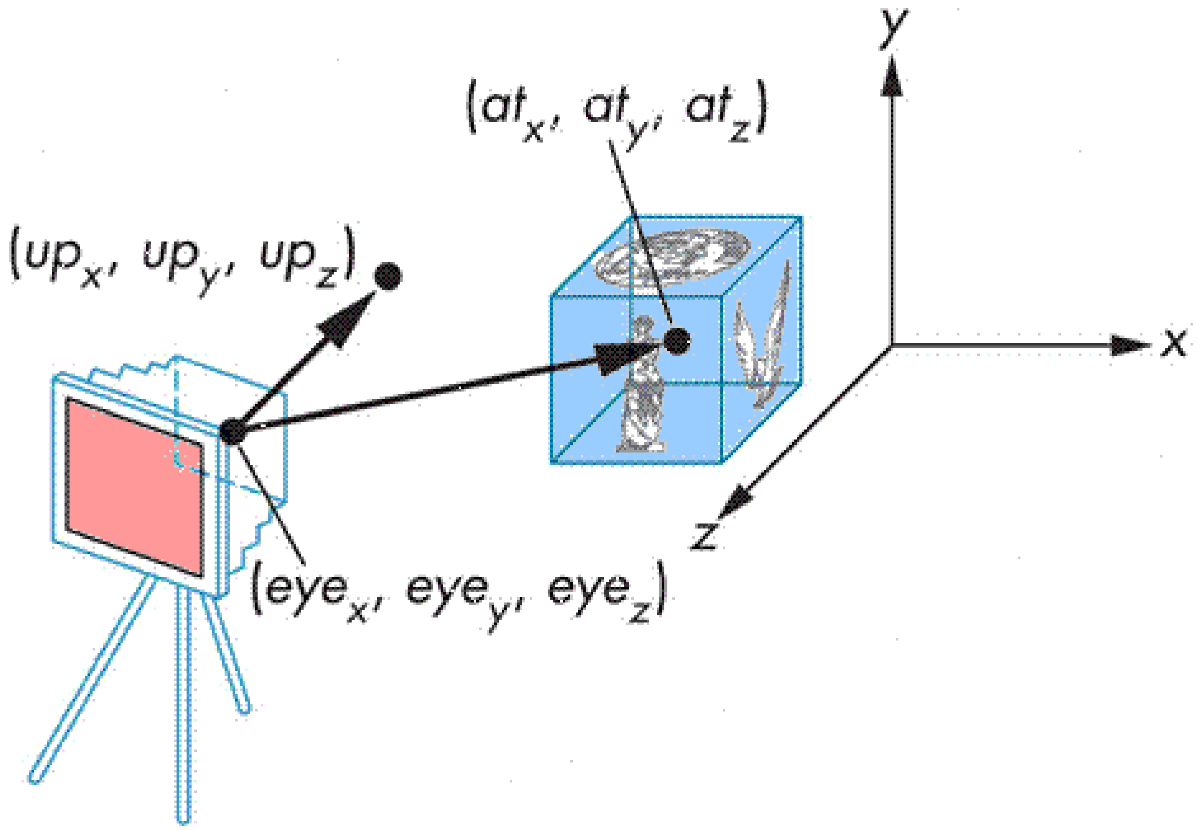
Figure 3-9 Object and Viewpoint at the Origin



Viewing Transformation

- void **gluLookAt**(*eyeX*, *eyeY*, *eyeZ*, *centerX*, *centerY*, *centerZ*, *upX*, *upY*, *upZ*);





Example: modeling + viewing transformation

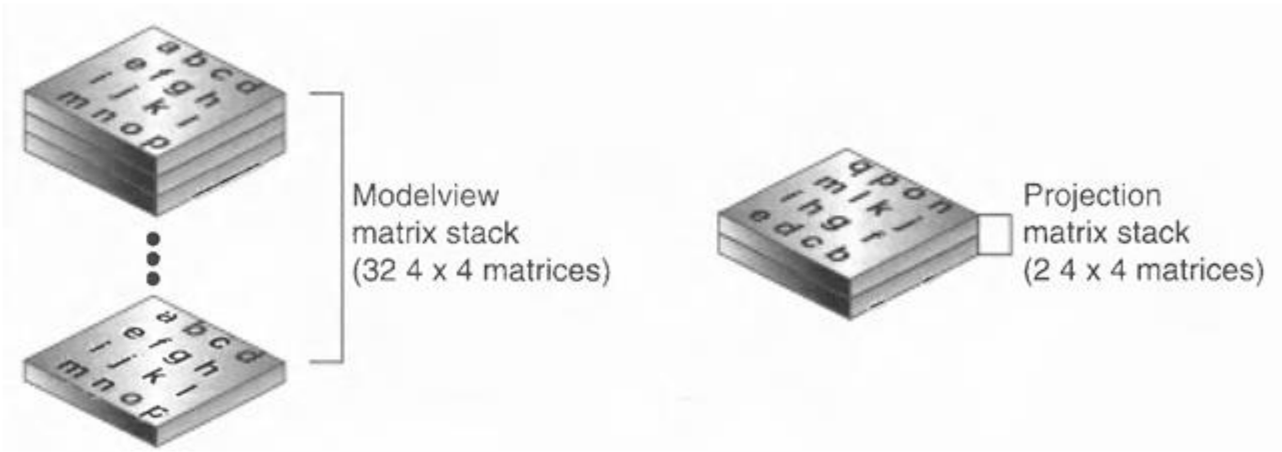
- With all this, we can give an outline for a typical display routine for drawing an image of a 3D scene with OpenGL 1.1:

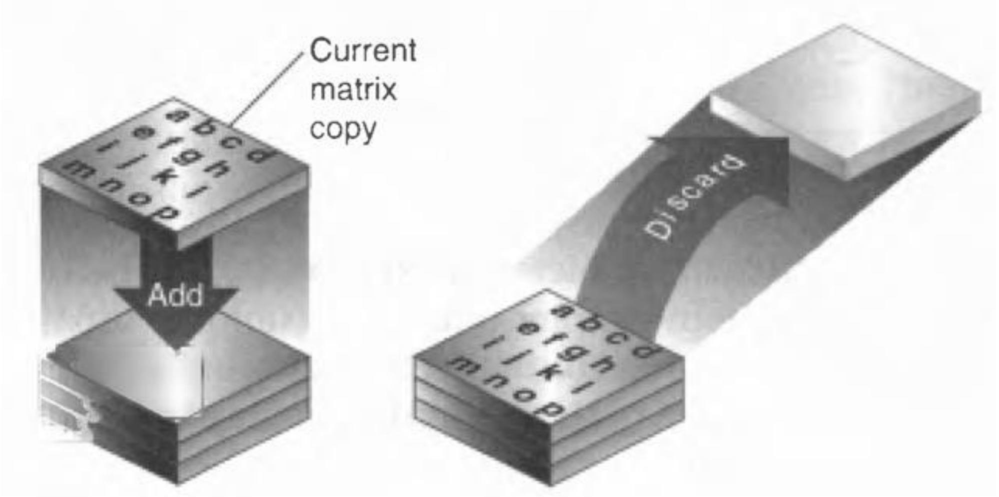
```
// possibly set clear color here, if not set elsewhere
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

// possibly set up the projection here, if not done elsewhere
glMatrixMode( GL_MODELVIEW ); glLoadIdentity();
gluLookAt( eyeX,eyeY,eyeZ, refX,refY,refZ, upX,upY,upZ ); // Viewing transform

glPushMatrix();
. .. // apply modeling transform and draw an object
glPopMatrix();

glPushMatrix();
. .. // apply another modeling transform and draw another object
glPopMatrix()
…
```

pushes the current matrix stack down by one, duplicating the current matrix: **glPushMatrix()**

pops the current matrix stack, replacing the current matrix with the one below it on the stack: **glPopMatrix()**

Manipulating the Matrix Stacks





void glPushMatrix(void);

Pushes all matrices in the current stack down one level. The current stack is determined by **glMatrixMode()**. The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the-top matrix. If too many matrices are pushed, an error is generated.

void glPopMatrix(void);

Pops the top matrix off the stack, destroying the contents of the popped matrix. What was the second-from-the-top matrix becomes the top matrix. The current stack is determined by **glMatrixMode()**. If the stack contains a single matrix, calling **glPopMatrix()** generates an error.

Assuming you are drawing a car with four wheels:
Draw the car body.

- Remember where you are, and translate to the right front wheel.
- Draw the wheel and throw away the last translation so your current position is back at the origin of the car body.
- Remember where you are, and translate to the left front wheel…

glPushMatrix() means “remember where you are” and **glPopMatrix()** means “go back to where you were.”

Current matrix and matrix stack

- glLoadIdentity — replace the current matrix with the identity matrix

  It is semantically equivalent to calling glLoadMatrix with the identity matrix

- glLoadMatrix — replace the current matrix with the specified matrix

Column major

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

- glMultMatrix
  - The current matrix is postmultiplied by the matrix

```
glLoadIdentity();
glMultMatrixf (M1);
glMultMatrixf (M2);
```

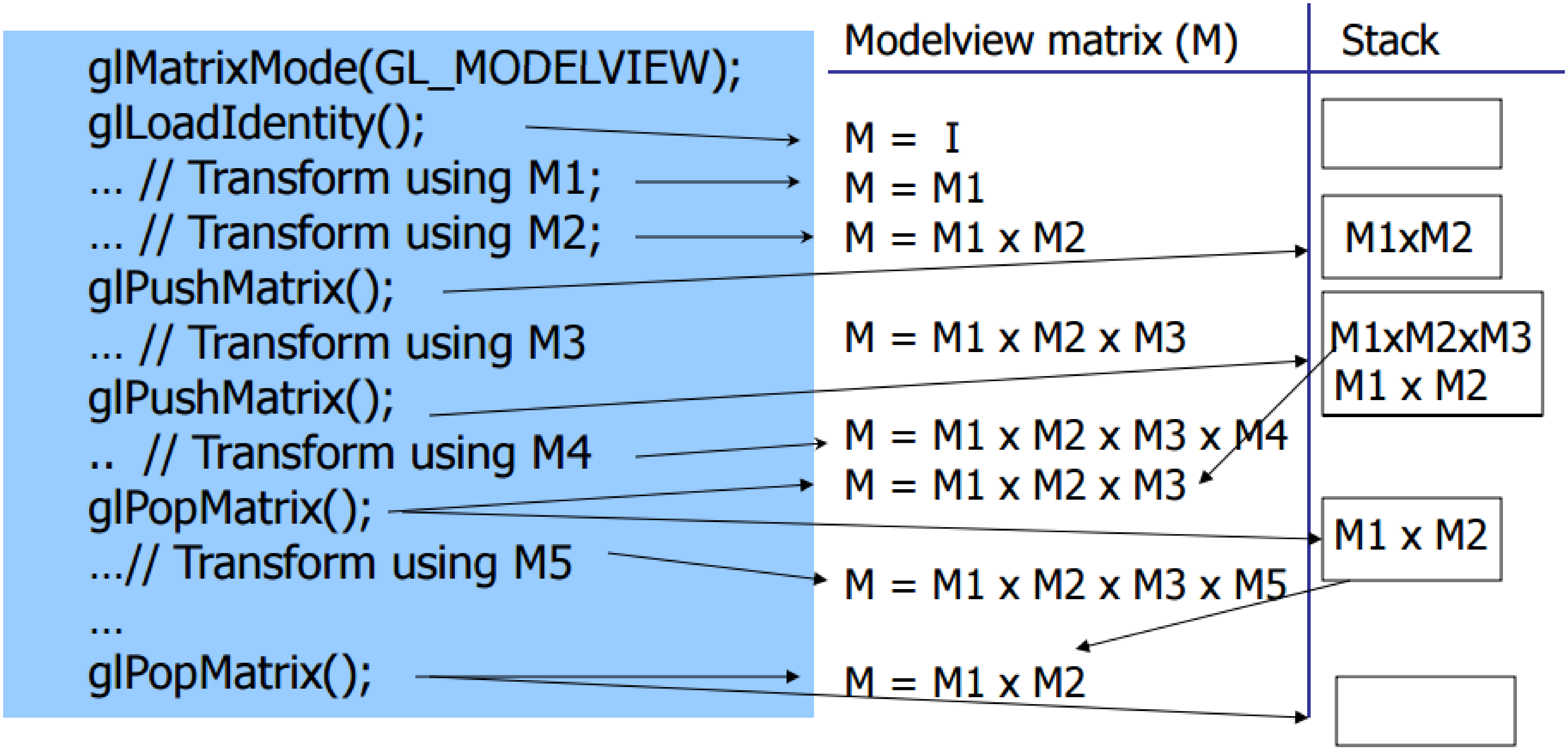
$M = M1 \cdot M2$

- matrix stack
  - glPushMatrix pushes the current matrix stack down by one, duplicating the current matrix. That is, after a glPushMatrix call, the matrix on top of the stack is identical to the one below it.
  - glPopMatrix pops the current matrix stack, replacing the current matrix with the one below it on the stack.
  - Initially, each of the stacks contains one matrix, an identity matrix.

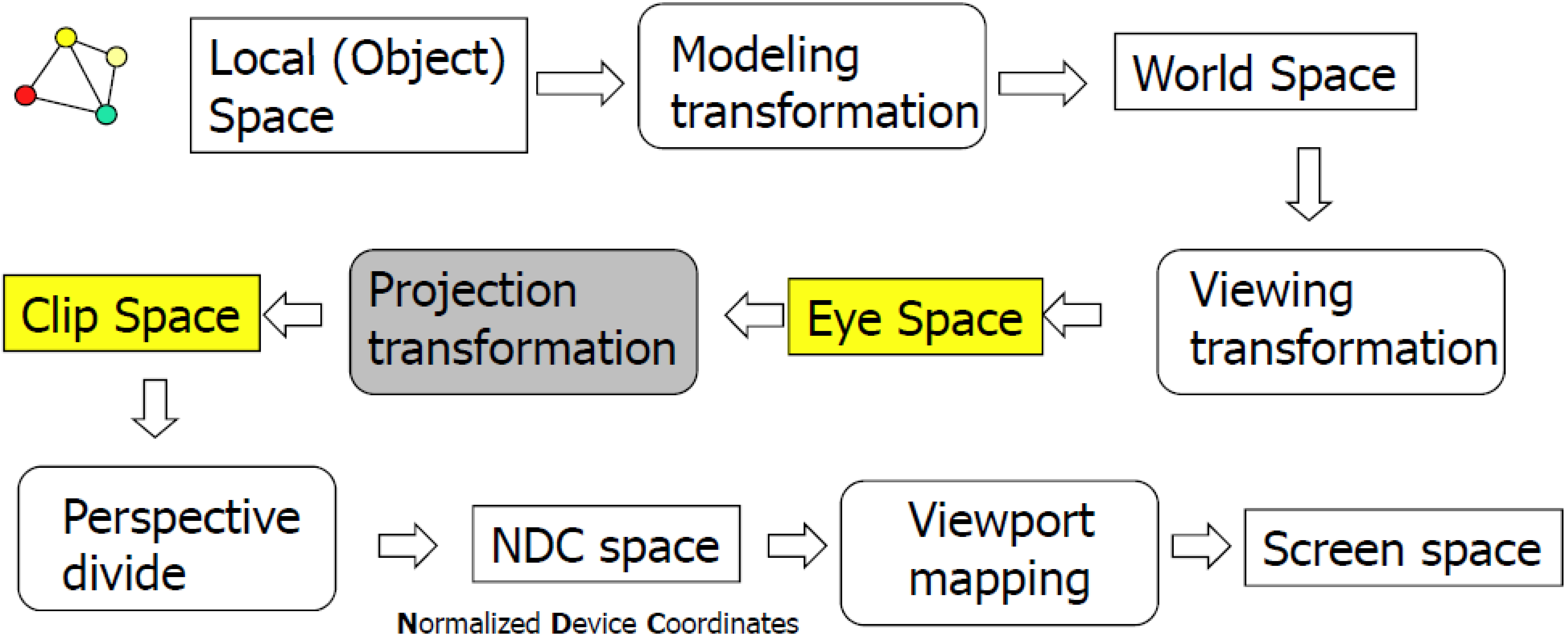
- Stack query

```
float mat[16];// get the modelview matrix
glGetFloatv(GL_MODELVIEW_MATRIX, mat);

Int depth;

glGetIntegerv( GL_MODELVIEW_STACK_DEPTH, &depth);
```

Push and Pop Matrix Stack



Transformation Pipeline



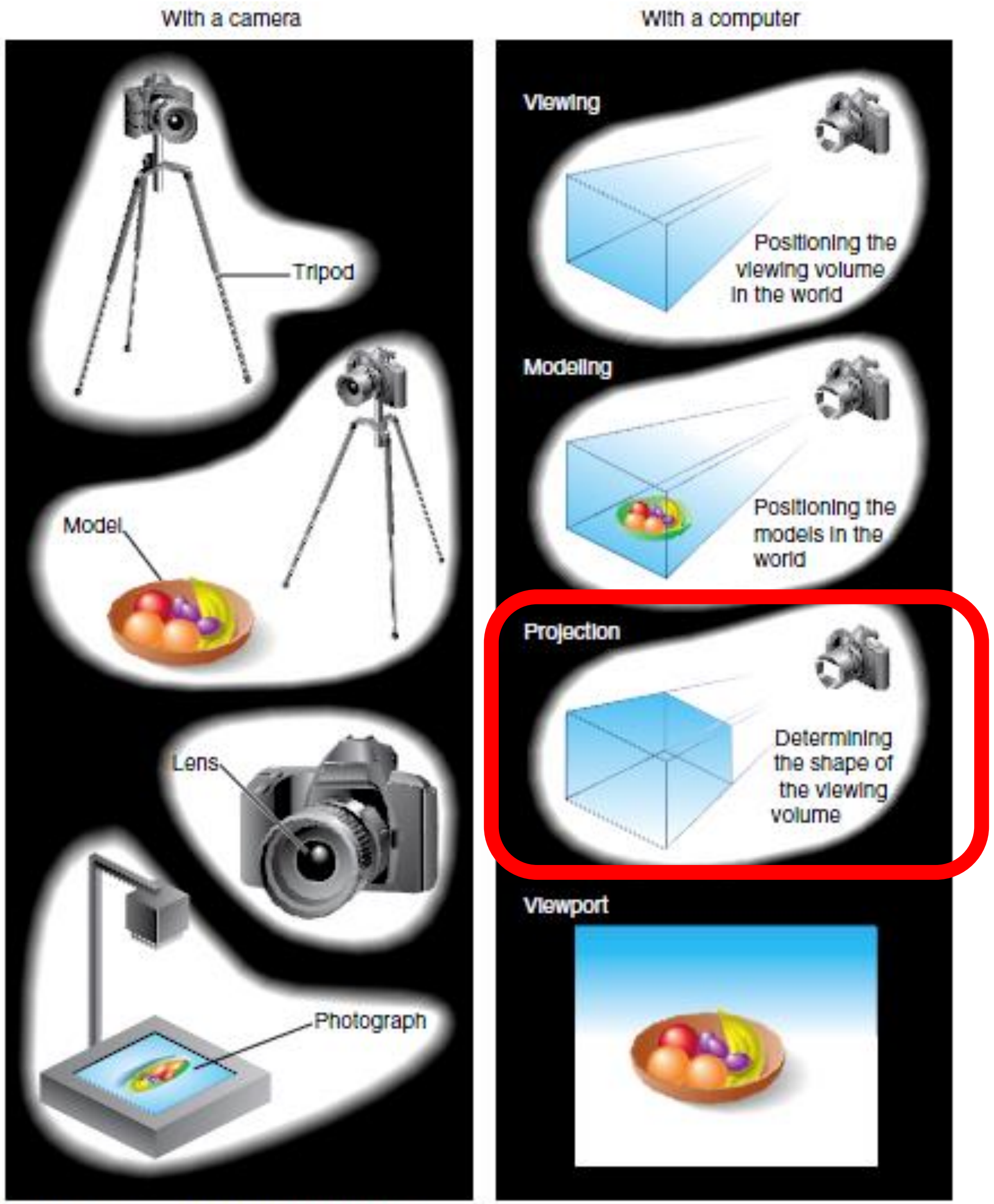
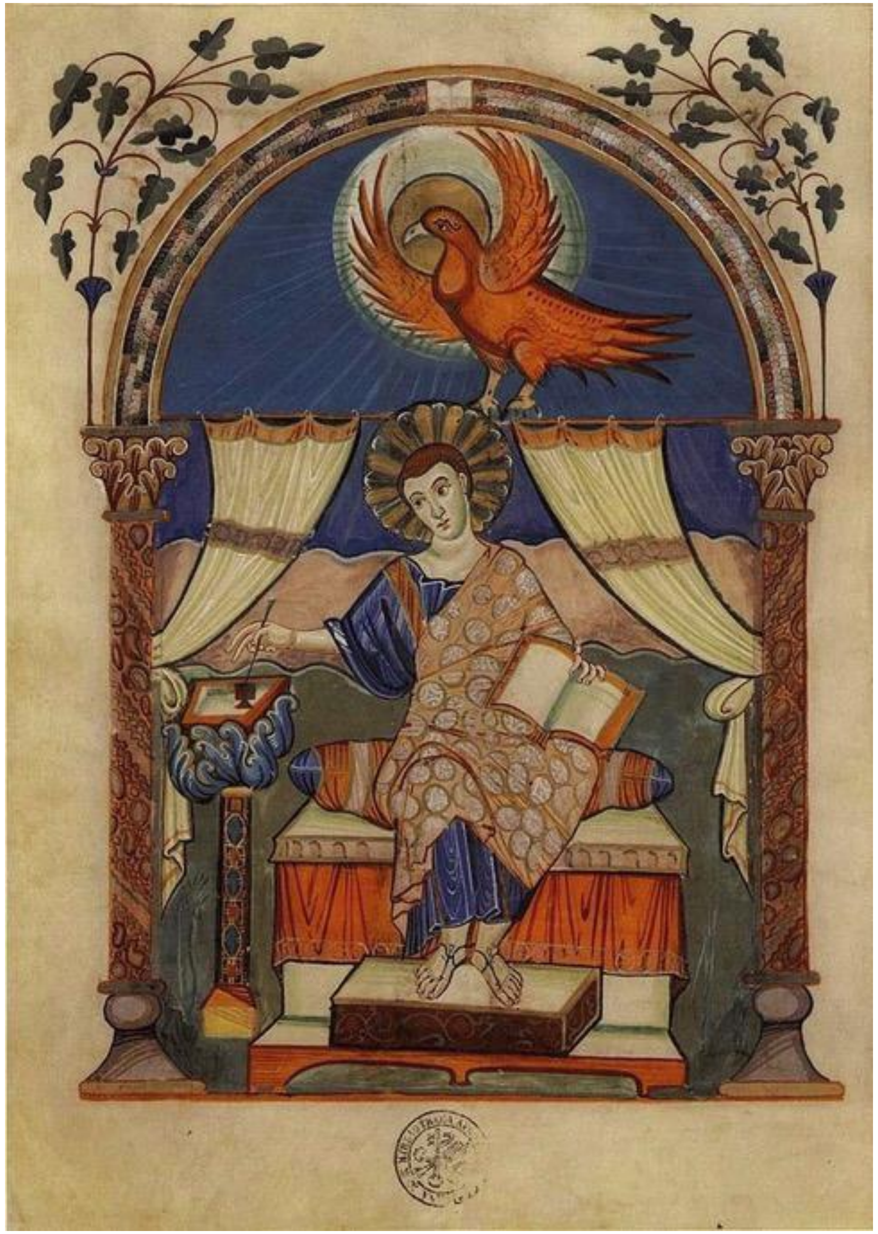


Figure 3-1 The Camera Analogy

Perspective projection

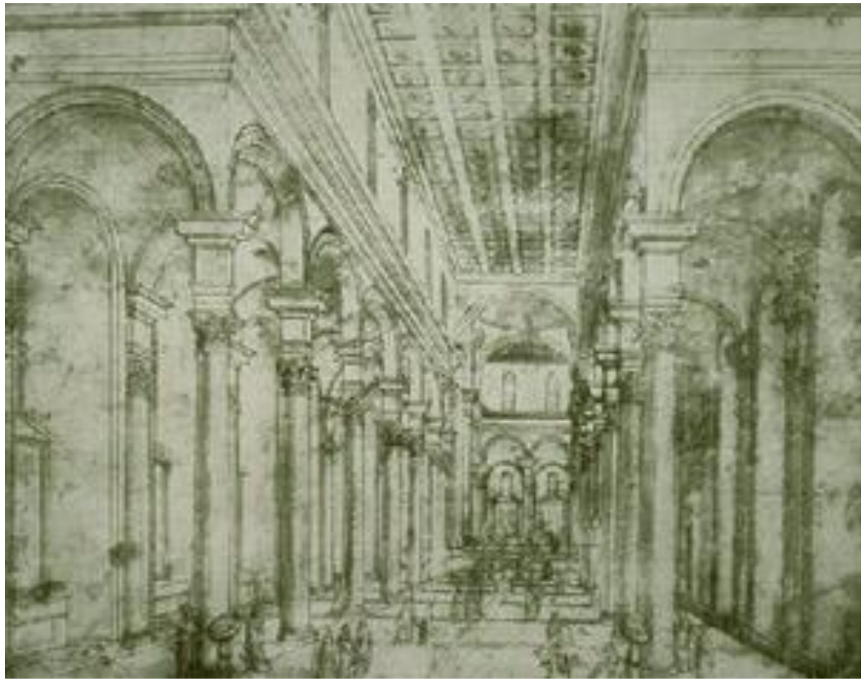
Early painting: incorrect perspective



8-9th century painting

Geometrically correct perspective in art

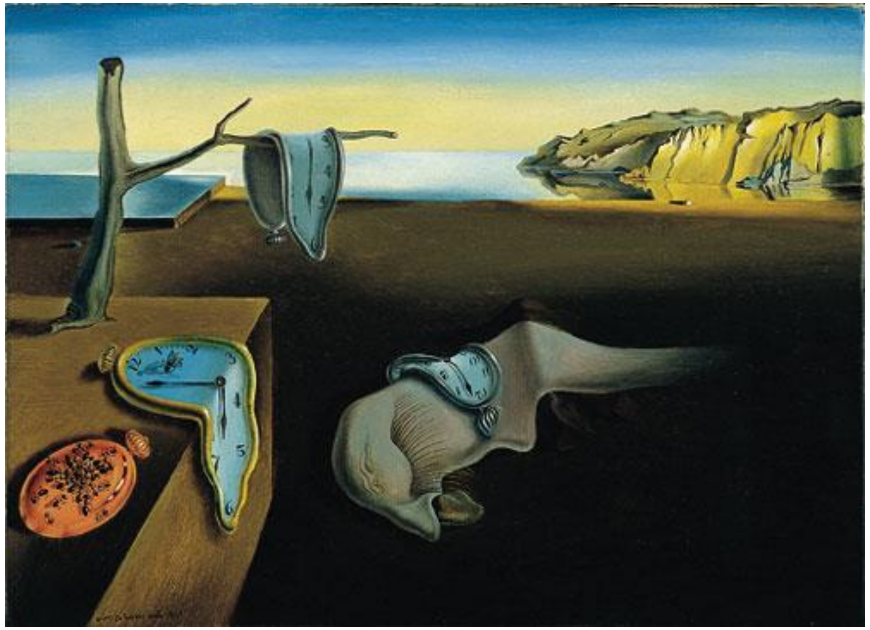
Ambrogio Lorenzetti
Annunciation, 1344

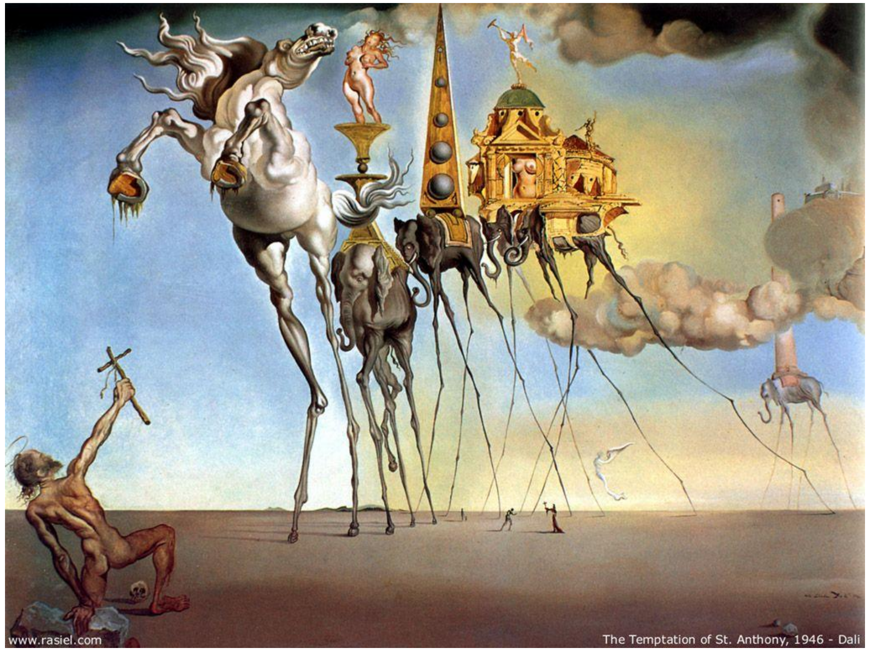


Brunelleschi, elevation of Santo Spirito,1434-83, Florence

Masaccio – The Tribute Money
c.1426-27
Fresco, The Brancacci Chapel, Florence

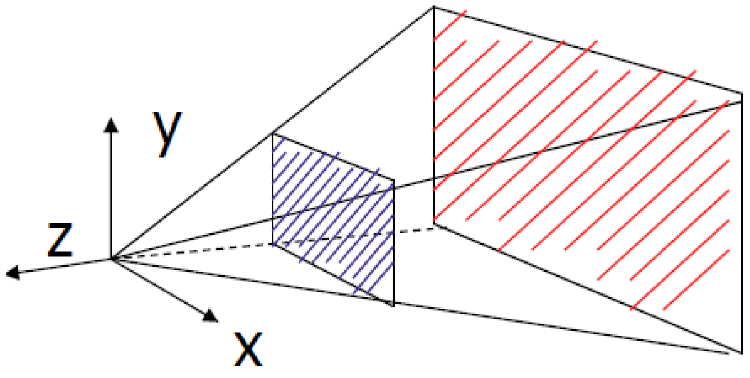
Later… rejection of proper perspective projection



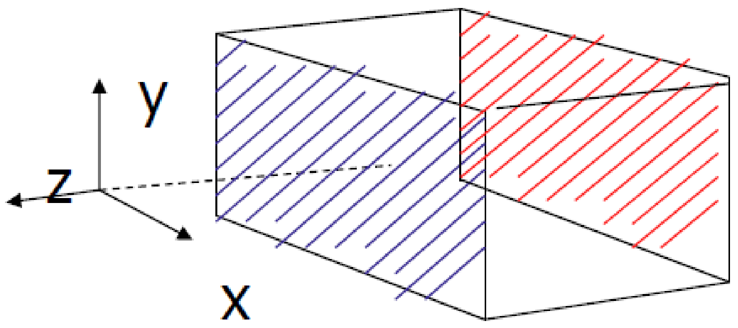


Projection Transformation

- Specifying PT is like choosing a **lens for a camera**
- The purpose of PT is to define a **viewing volume**, which is used in two ways.
  - The viewing volume determines **how an object is projected** onto the screen (that is, by using a perspective or an orthographic projection), and
  - Defines **which objects or portions of objects are clipped out** of the final image
- Need to establish the appropriate mode for constructing the viewing transformation, or in other words select the projection mode
  - **glMatrixMode(GL_PROJECTION);**
- This designates the projection matrix as the current matrix, which is originally set to the identity matrix



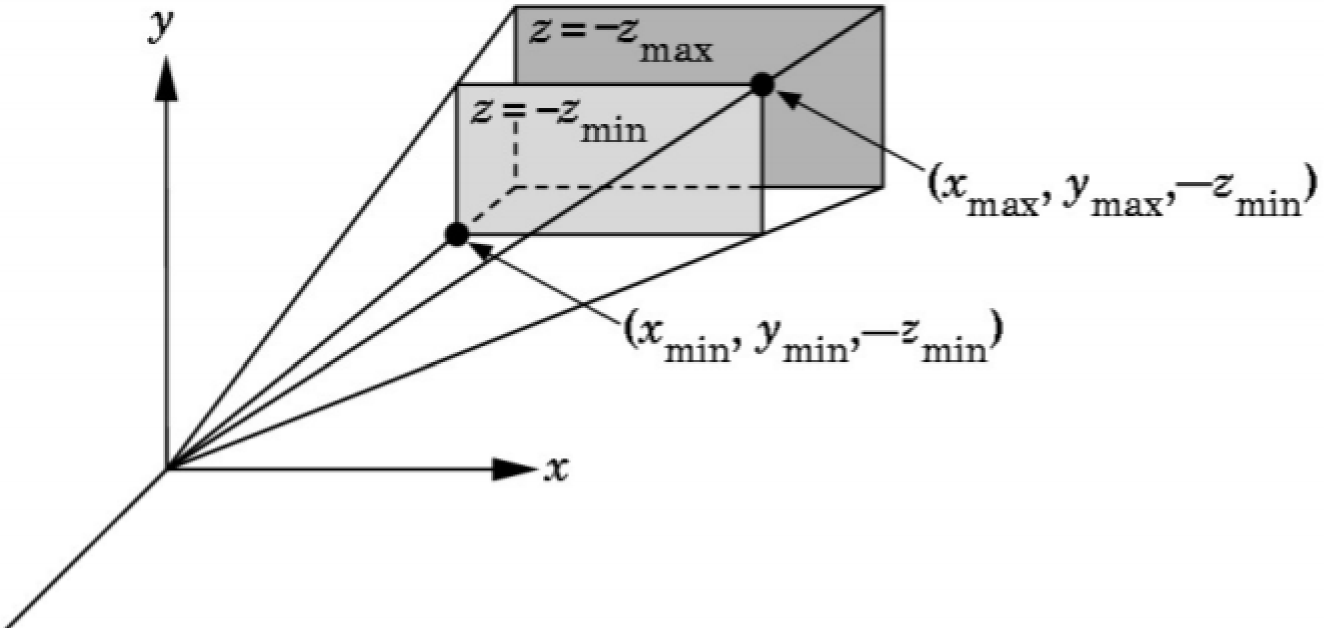
Perspective: **gluPerspective()**



Parallel: **glOrtho()**

Perspective Projection

glFrustum(xmin, xmax, ymin, ymax, N, F); N = near plane, F = far plane

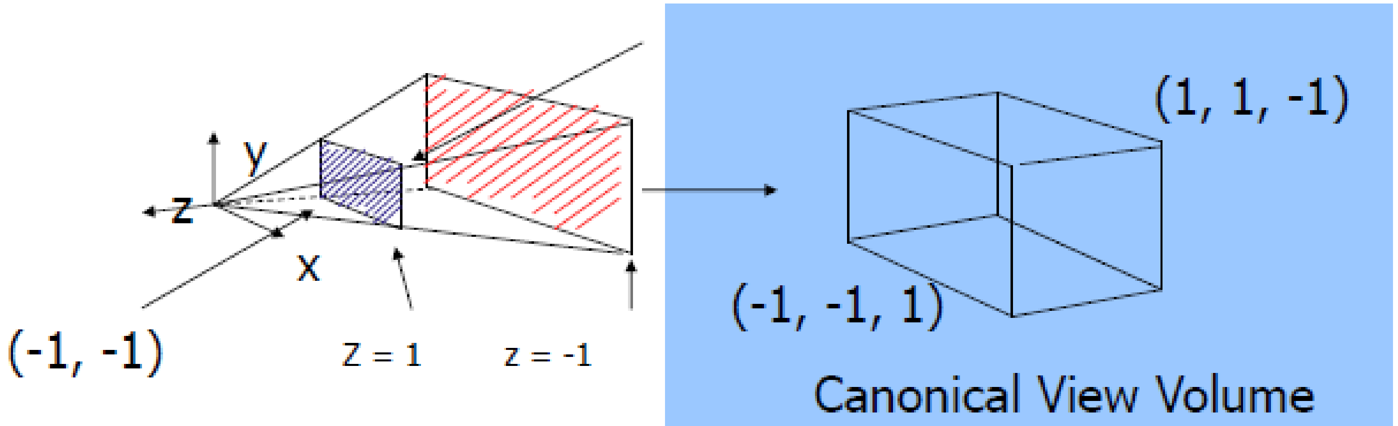


Projection Matrix

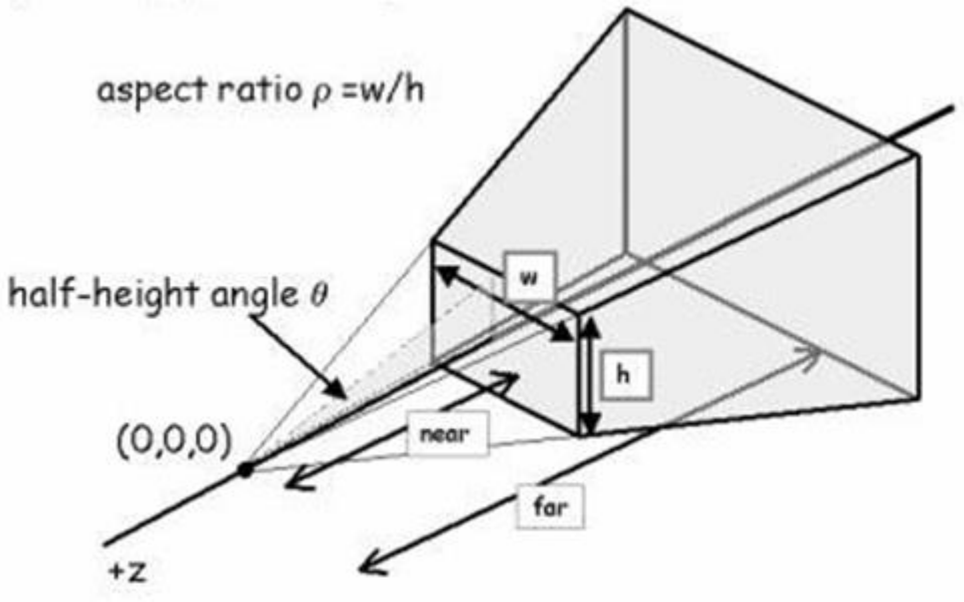
$$
\begin{array}{c} x' \\ y' \\ z' \\ w' \end{array}
=
\left|\begin{array}{cccc}
2N/(xmax-xmin) & 0 & (xmax+xmin)/(xmax-xmin) & 0 \\
0 & 2N/(ymax-ymin) & (ymax+ymin)/(ymax-ymin) & 0 \\
0 & 0 & -(F + N)/(F-N) & -2F*N/(F-N) \\
0 & 0 & -1 & 0
\end{array}\right|
\begin{array}{c} x \\ y \\ z \\ 1 \end{array}
$$

gluPerspective

- glFrustum() isn't intuitive to use so can use **gluPerspective** to specify
  - Fovy: the angle of the field of view in the y direction
  - Aspect: the aspect ratio of the width to height (x/y)
  - Near & far: distance between the viewpoint and the near and far clipping planes
- Note that gluPerspective() is limited to creating frustums that are symmetric in both the x- and y-axes along the line of sight



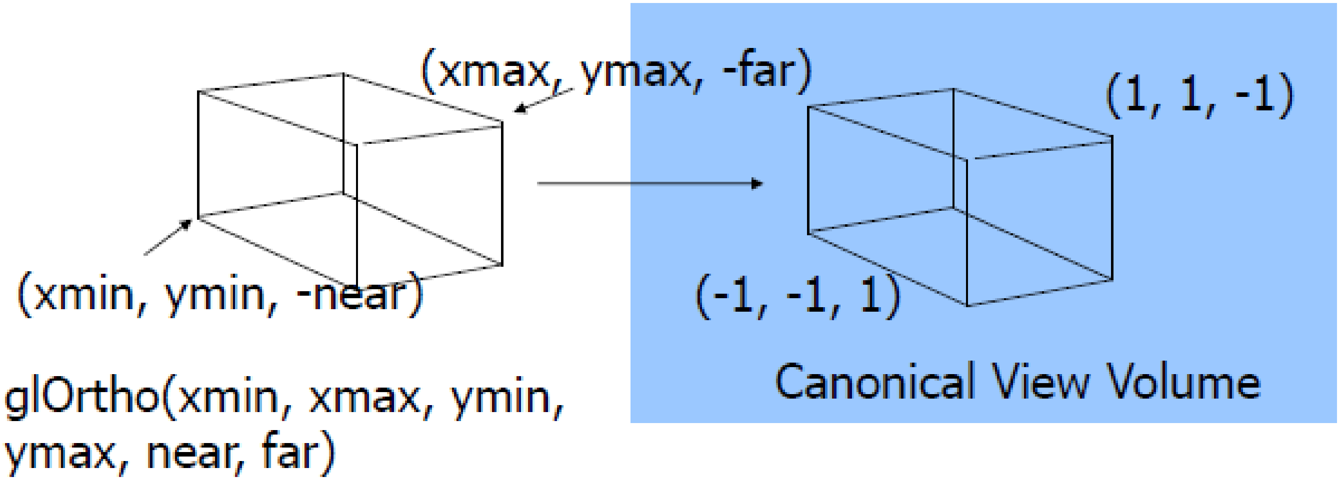
gluPerspective(fovy, aspect, near, far)

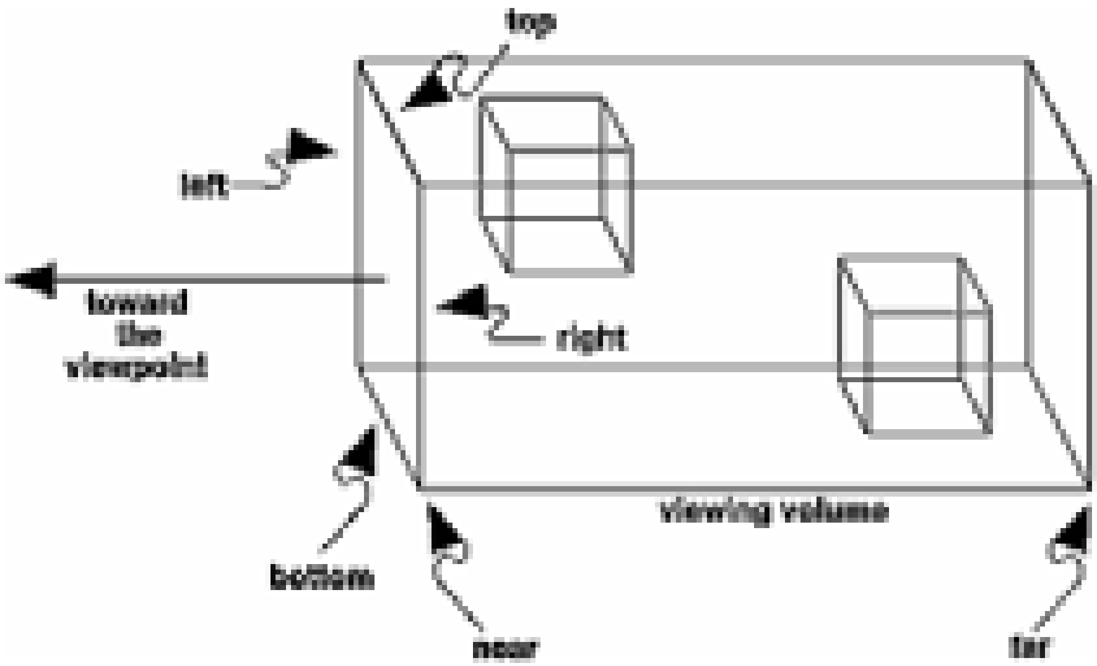


Maps (projects) everything in the visible volume into a canonical view volume

Orthographic Projection

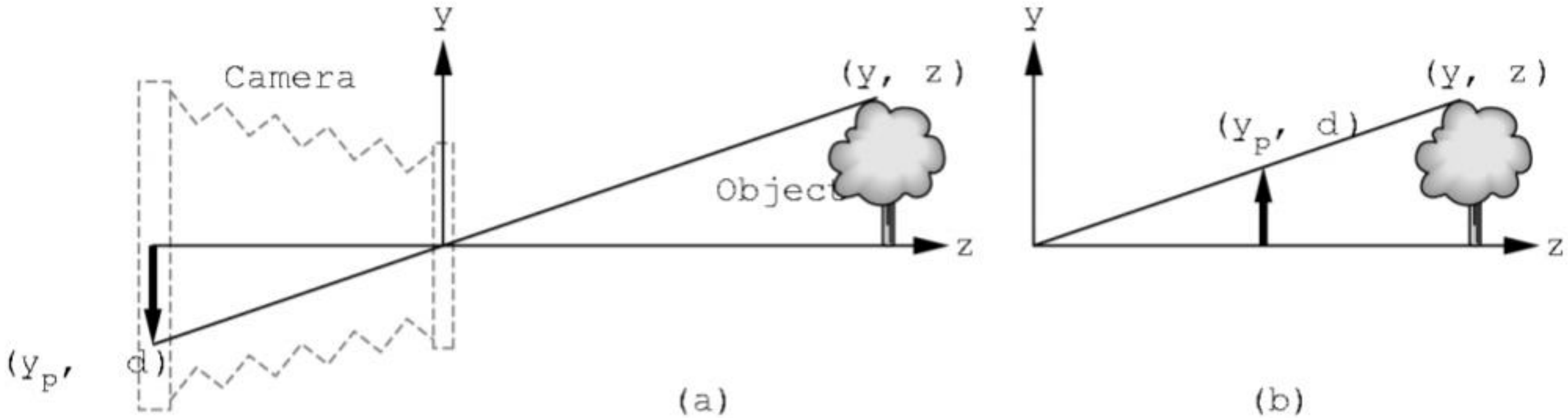
- Ortographic projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them
  - void **gluOrtho2D** (left, right, bottom, top);
  - void **glOrtho** (left, right, bottom, top, near, far);





Maps (projects) everything in the visible volume into a canonical view volume

Perspective Viewing Mathematically



- d = focal length
- $y/z = y_p/d$ so $y_p = y/(z/d) = yd/z$
- Note that y_p is non-linear in the depth z!

homogeneous coordinates

Perspective projection is not affine:

$$M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$ has no solution for M

Idea: exploit homogeneous coordinates

$$p = w \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$ for arbitrary $w \neq 0$

Perspective Projection Matrix

- Use multiple of point

$$(z/d)\begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix}$$

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

- Solve

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \quad \text{with} \quad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

Projection Algorithm

- **Input**: 3D point $[x\ y\ z]^\top$ to project
- Form $[x\ y\ z\ 1]^\top$
- Multiply M with $[x\ y\ z\ 1]^\top$; obtaining $[X\ Y\ Z\ W]^\top$
- Perform perspective division:
  X/W , Y/W , Z/W
- **Output**: $[X/W, Y/W, Z/W]^\top$
- (last coordinate will be d)

$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} \quad \text{with} \quad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

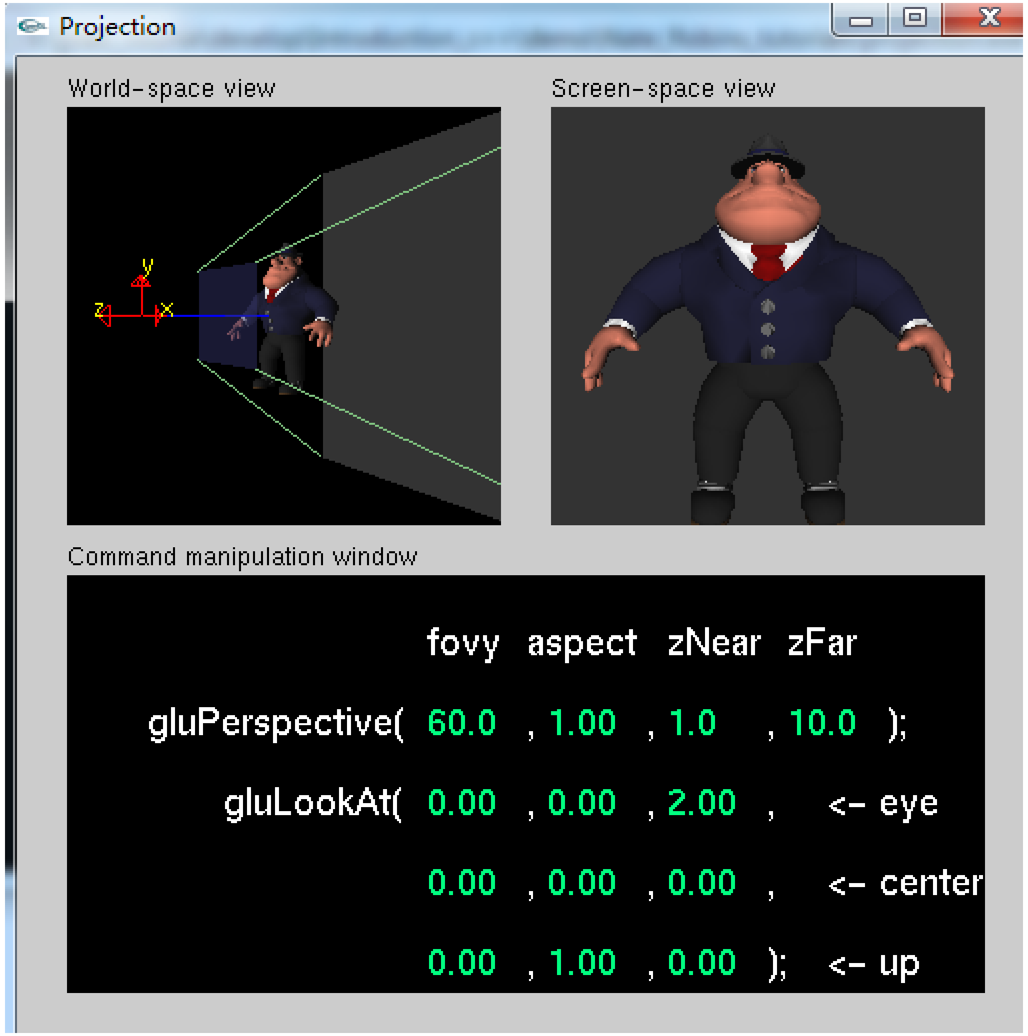
$$M\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

Perspective Division

- Normalize $[X\ Y\ Z\ W]^\top$ to $[X/W,\ Y/W,\ Z/W,\ 1]^\top$
- Perform perspective division after projection

- Projection in OpenGL is more complex (includes clipping)

Projection & Viewpoint (cont)

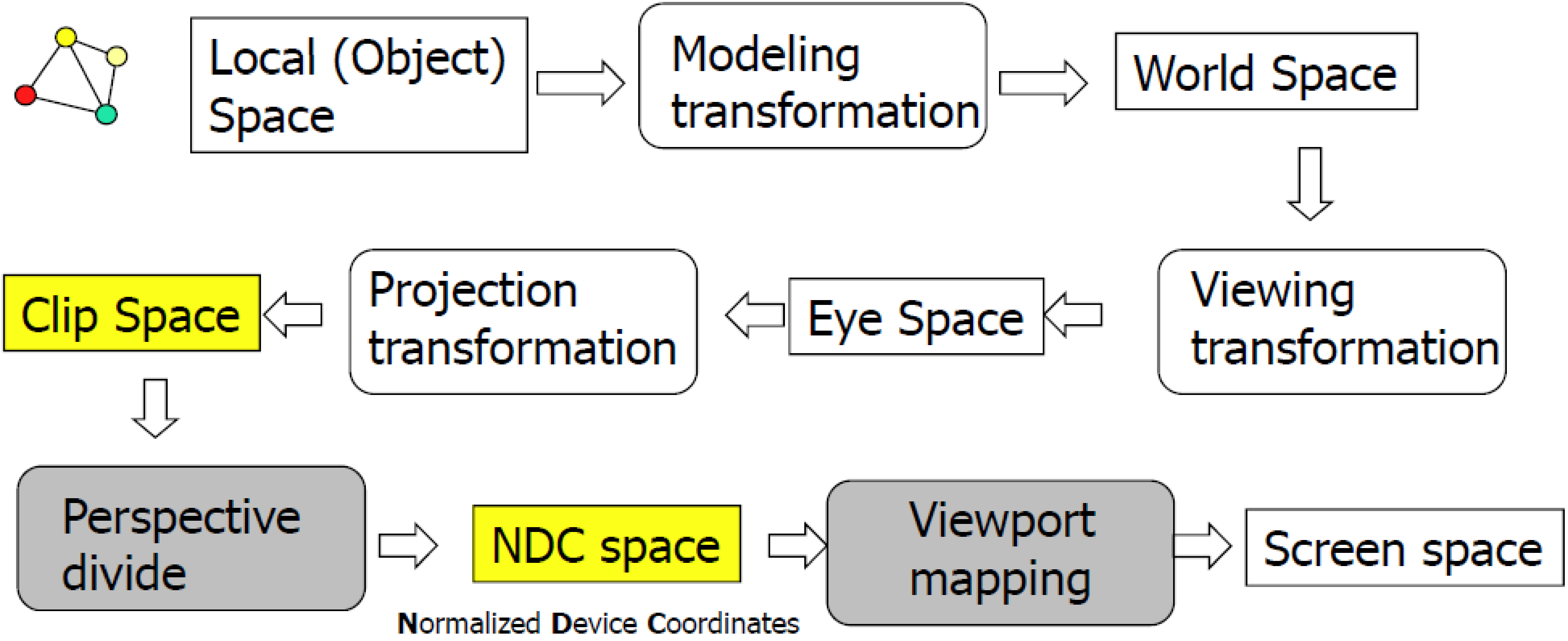


Nate_Robins_tutorials: Projection

The Golden Rule

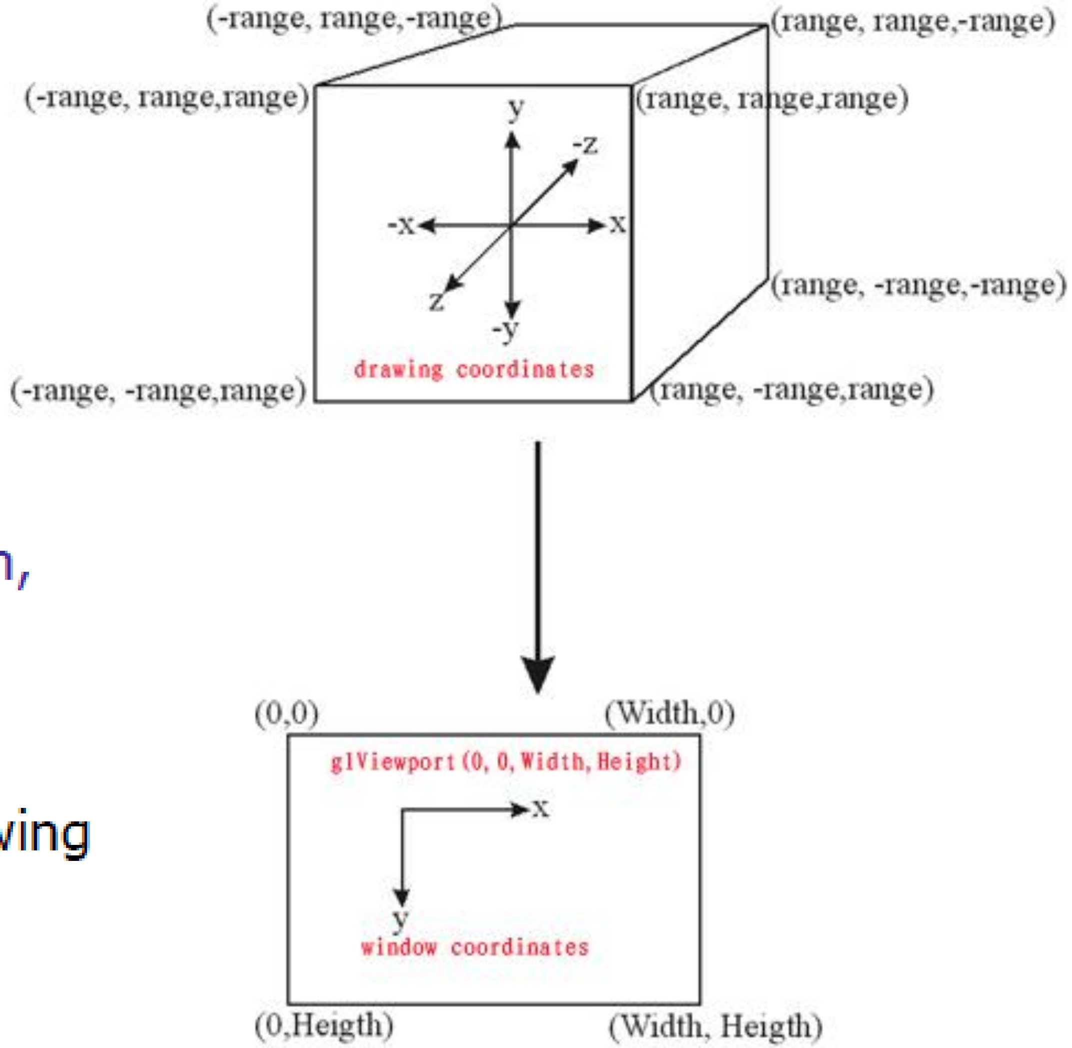
- Modeling transformation
  - glMatrixMode(GL_MODELVIEW); glRotate3f?
- Viewing transformation
  - glMatrixMode(GL_MODELVIEW); gluLookAt()
- Projection transformation
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity - to initialize current matrix.
  - **gluPerspective/glFrustum/glOrtho/gluOrtho2 - to set the appropriate projection onto the stack.**
  - You *could* use glLoadMatrix to set up your own projection matrix (if you understand the restrictions and consequences) - but I'm told that this can cause problems for some OpenGL implementations which rely on data passed to glFrustum, etc to determine the near and far clip planes.

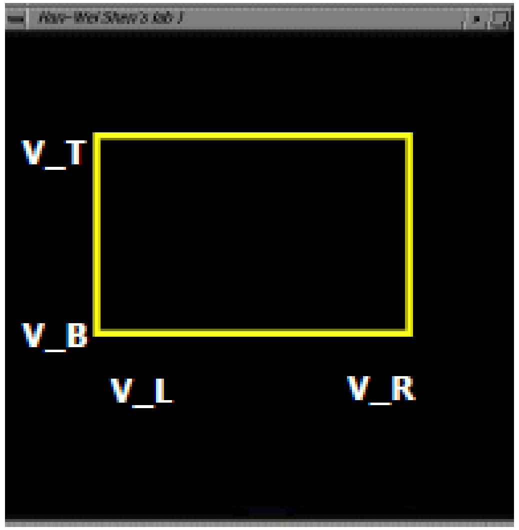
Transformation Pipeline



Viewpoint transformation

- The **viewport** maps the **drawing coordinates** to **window coordinates** and therefore defines the region of the scene, which can be seen. If the user resizes the window, we have to adjust the **viewport** and correct the aspect ratio.
-





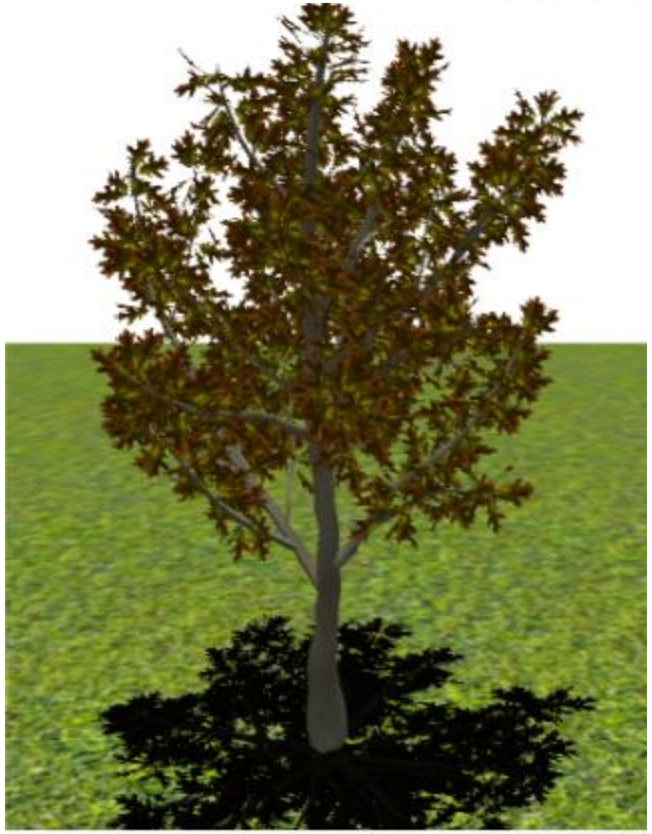
glViewport(int left, int bottom,
int (right-left),
int (top-bottom));

call this function before drawing
(calling glBegin() and
glEnd())

Hierarchical Models

- Many graphical objects are structured
- Structure often naturally hierarchical
  - Wheels of a car
  - Arms or legs of a figure
  - Chess pieces

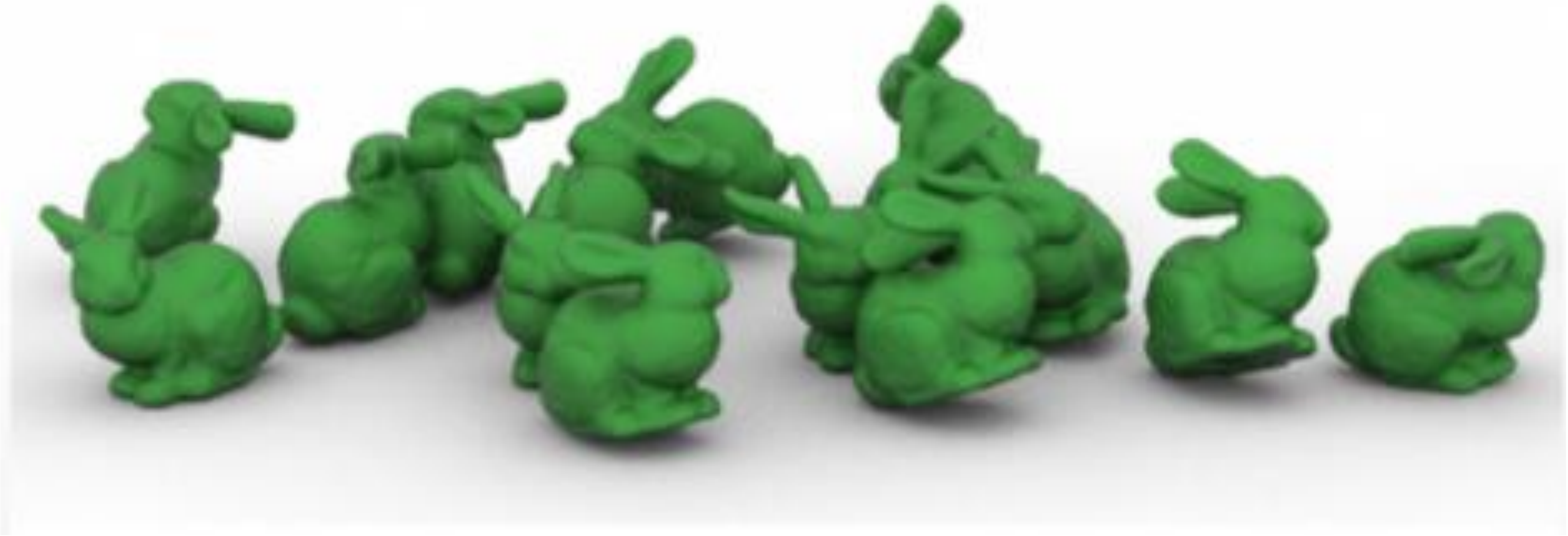
- Exploit structure for
  - Efficient rendering
    - Example: tree leaves
  - Concise specification of model parameters
    - Example: joint angles



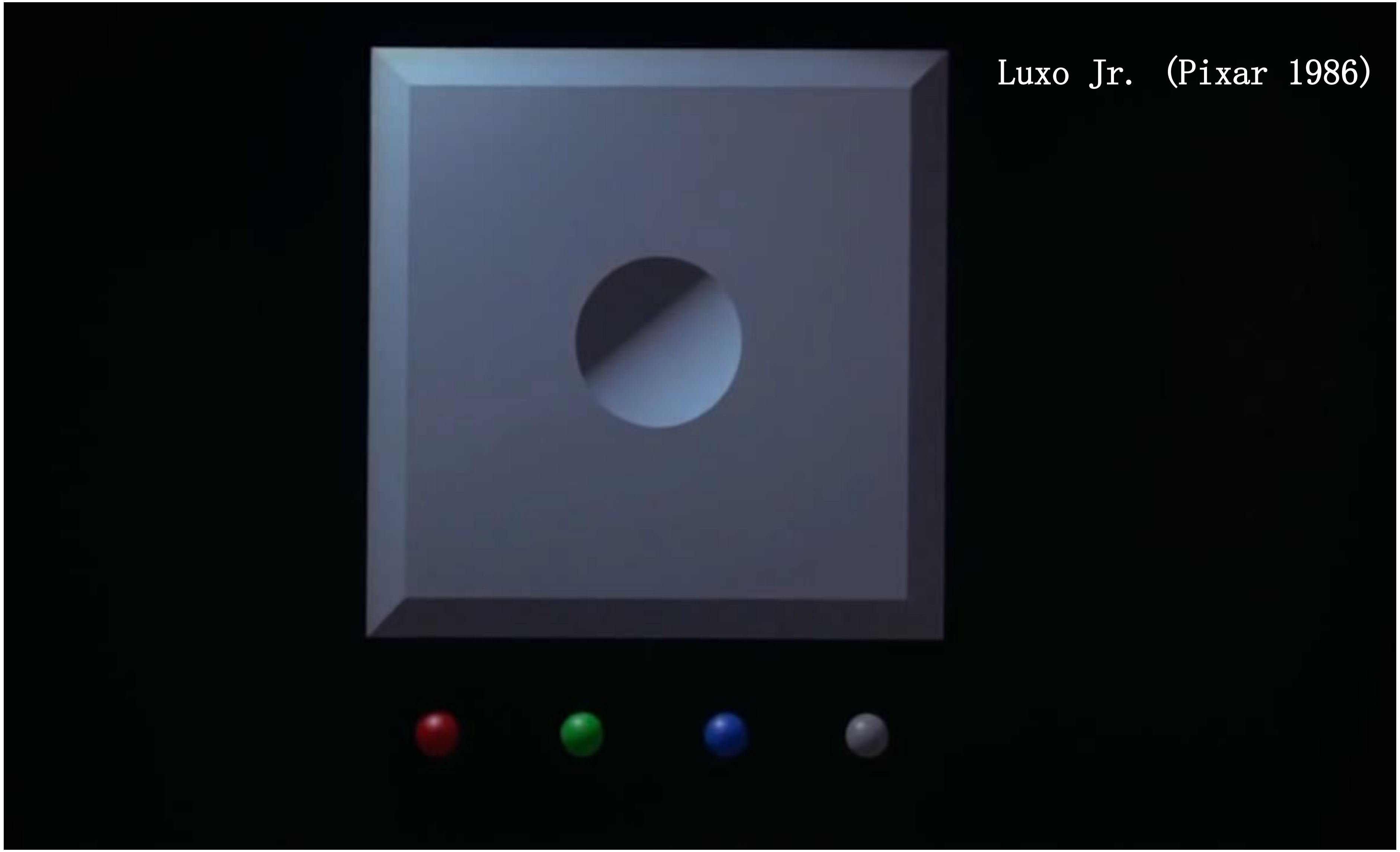
Instance Transformation

- Instances can be shared across space or time

- Write a function that renders the object in “standard” configuration
- Apply transformations to different instances
- Typical order: scaling, rotation, translation



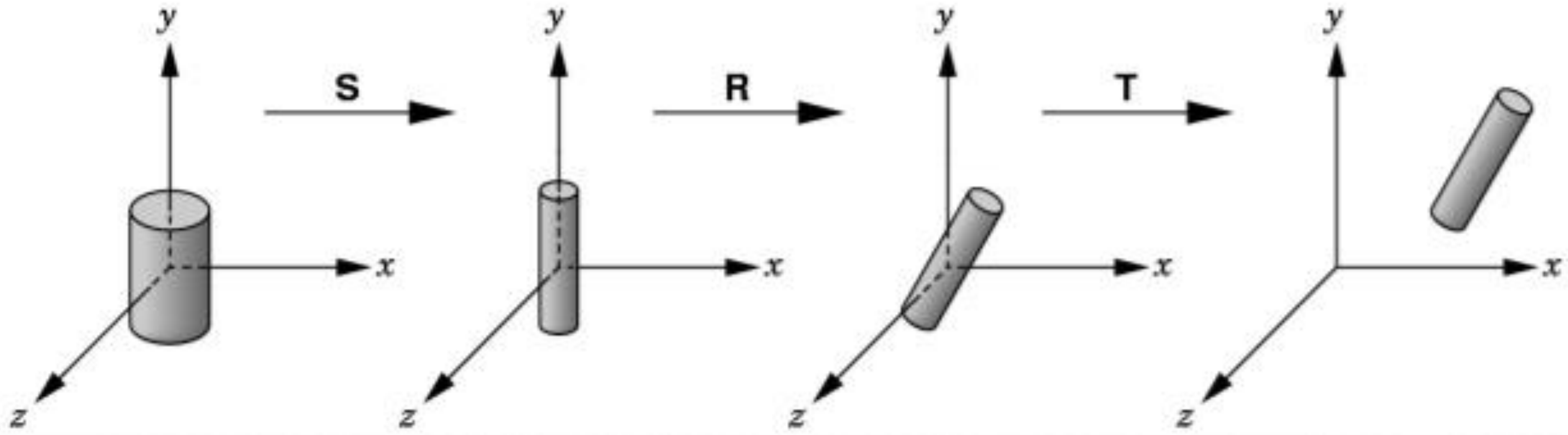
Animation: modeling motion



Pixar Shorts Collection Luxo Jr 1986

Sample Instance Transformation

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(...);
glRotatef(...);
glScalef(...);
gluCylinder(...);
```



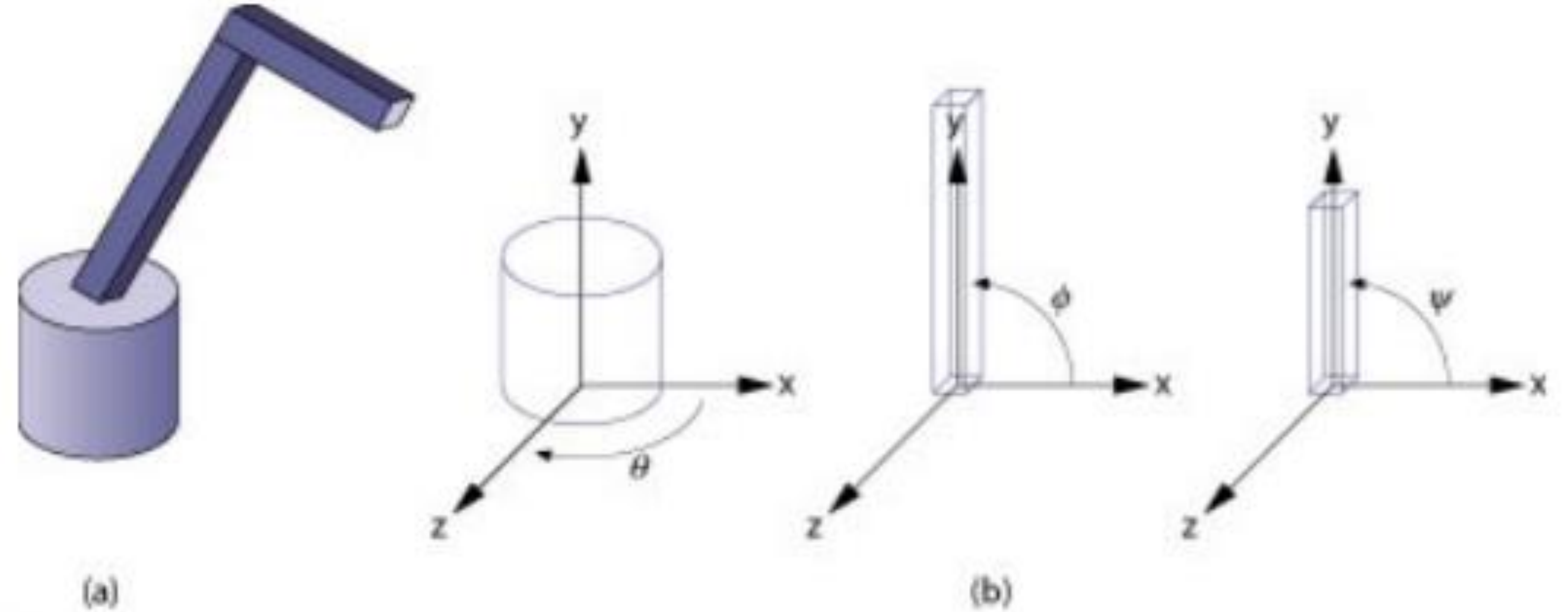
Display Lists

- Sharing display commands
- Display lists are stored on the GPU
- May contain drawing commands and transfns.
- Initialization:

```
GLuint torus = glGenLists(1);
glNewList(torus, GL_COMPILE);
  Torus(8, 25);
glEndList();
```

- Use: glCallList(torus);
- Can share both within each frame, and across different frames in time
- Can be hierarchical: a display list may call another

Drawing a Compound Object

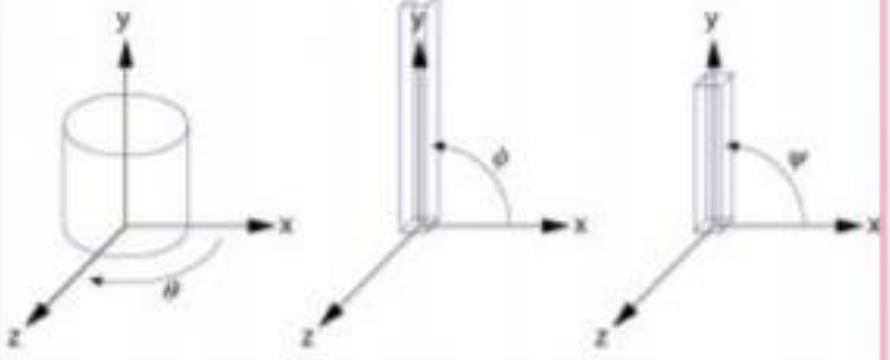


Base rotation θ, arm angle ϕ, joint angle ψ

Interleave Drawing & Transformation

h1 = height of base, h2 = length of lower arm

```
void drawRobot(GLfloat theta, GLfloat phi, GLfloat psi)
{
   glRotatef(theta, 0.0, 1.0, 0.0);
   drawBase();
   glTranslatef(0.0, h1, 0.0);
   glRotatef(phi, 0.0, 0.0, 1.0);
   drawLowerArm();
   glTranslatef(0.0, h2, 0.0);
   glRotatef(psi, 0.0, 0.0, 1.0);
   drawUpperArm();
}
```



Assessment of Interleaving

- Compact
- Correct “by construction”
- Efficient
- Inefficient alternative:

```
glPushMatrix();
glRotatef(theta, ...);
drawBase();
glPopMatrix();
```

```
glPushMatrix();
glRotatef(theta, ...);
glTranslatef(...);
glRotatef(phi, ...);
drawLowerArm();
glPopMatrix();
```

```
...etc...
```

Hierarchical Objects and Animation

- Drawing functions are time-invariant
  - drawBase(); drawLowerArm(); drawUpperArm();

- Can be easily stored in display list

- Change parameters of model with time

- Redraw when idle callback is invoked

A Bug to Watch

- GLfloat theta = 0.0; ...; /* update in idle callback */
- GLfloat phi = 0.0; ...; /* update in idle callback */
- GLuint arm = glGenLists(1);

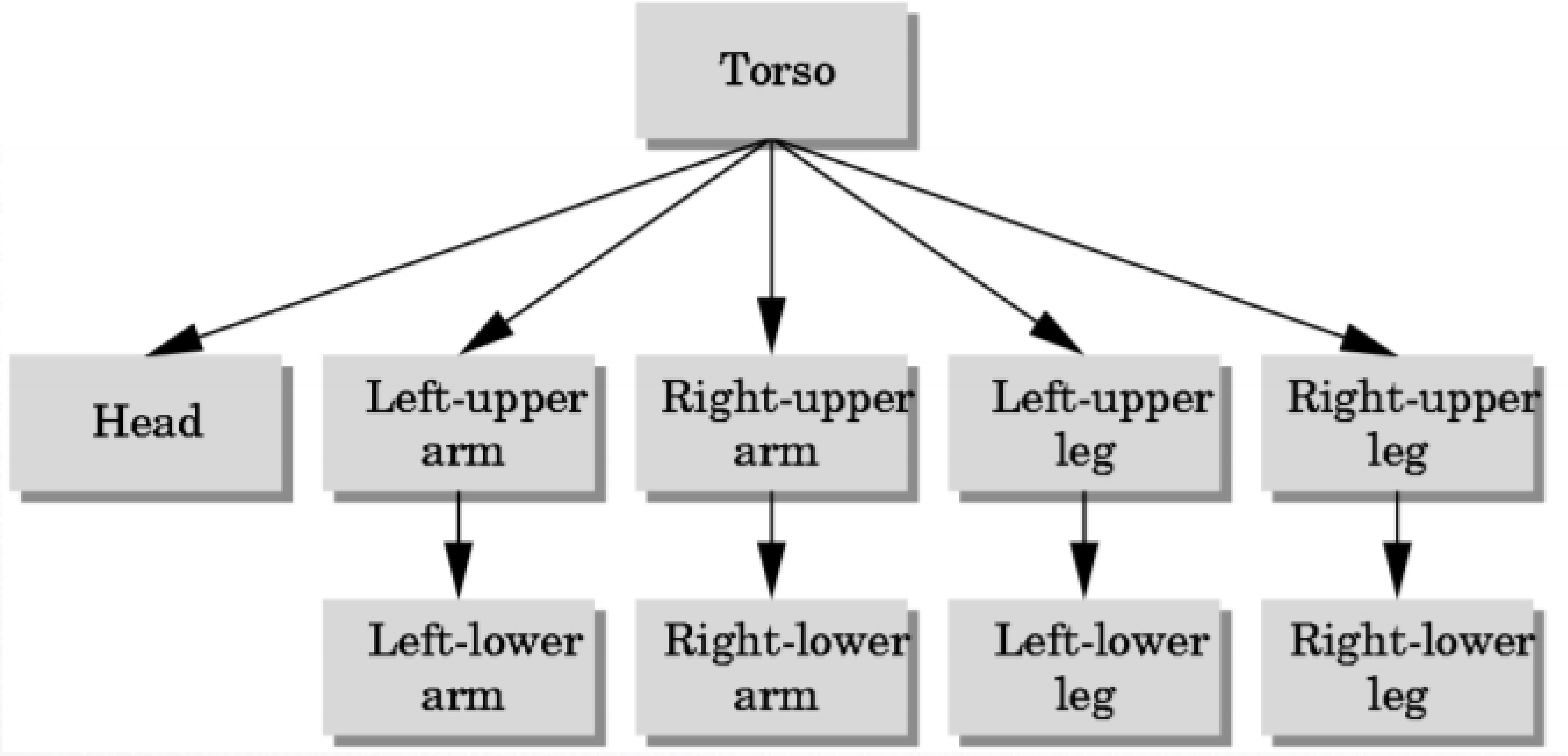
- /* in init function */
- glNewList(arm, GL_COMPILE);
- glRotatef(theta, 0.0, 1.0, 0.0);
- drawBase();
- ...
- drawUpperArm();
- glEndList();

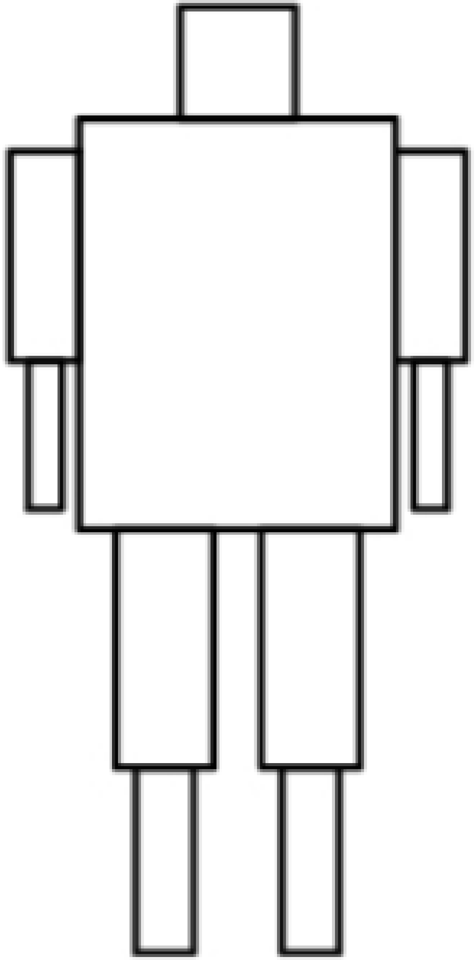
- /* in display callback */
- glCallList(arm);

What is wrong?

More Complex Objects

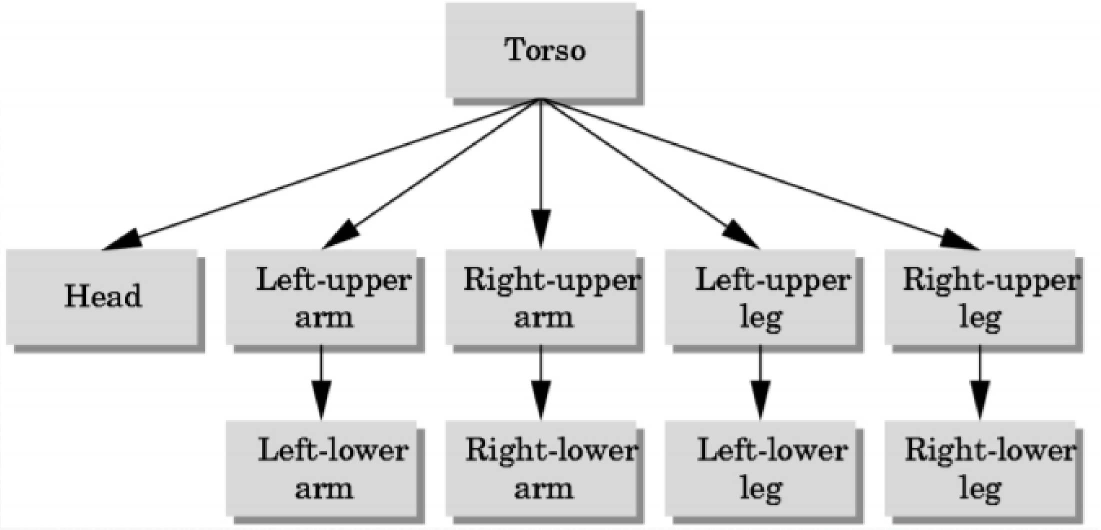
- Tree rather than linear structure
- Interleave along each branch
- Use push and pop to save state





Hierarchical Tree Traversal

- Order not necessarily fixed
- Example:




```
void drawFigure()
{
  glPushMatrix();  /* save */
  drawTorso();
  glTranslatef(...); /* move head */
  glRotatef(...);    /* rotate head */
  drawHead();
  glPopMatrix();   /* restore */

  glPushMatrix();
  glTranslatef(...);
  glRotatef(...);
  drawLeftUpperArm();
  glTranslatef(...)
  glRotatef(...)
  drawLeftLowerArm();
  glPopMatrix();
  ... }
```

Using Tree Data Structures

- Can make tree form explicit in data structure

```
typedef struct treenode
{
  GLfloat m[16];
  void (*f) ( );
  struct treenode *sibling;
  struct treenode *child;
} treenode;
```

Initializing Tree Data Structure

- Initializing transformation matrix for node

```
treenode torso, head, ...;
/* in init function */
glLoadIdentity();
glRotatef(...);
glGetFloatv(GL_MODELVIEW_MATRIX, torso.m);
```

- Initializing pointers

```
torso.f = drawTorso;
torso.sibling = NULL;
torso.child = &head;
```

Generic Traversal

- Recursive definition

```
void traverse (treenode *root)
{
  if (root == NULL) return;
  glPushMatrix();
  glMultMatrixf(root->m);
  root->f();
  if (root->child != NULL) traverse(root->child);
  glPopMatrix();
  if (root->sibling != NULL) traverse(root->sibling);
}
```

See demo code