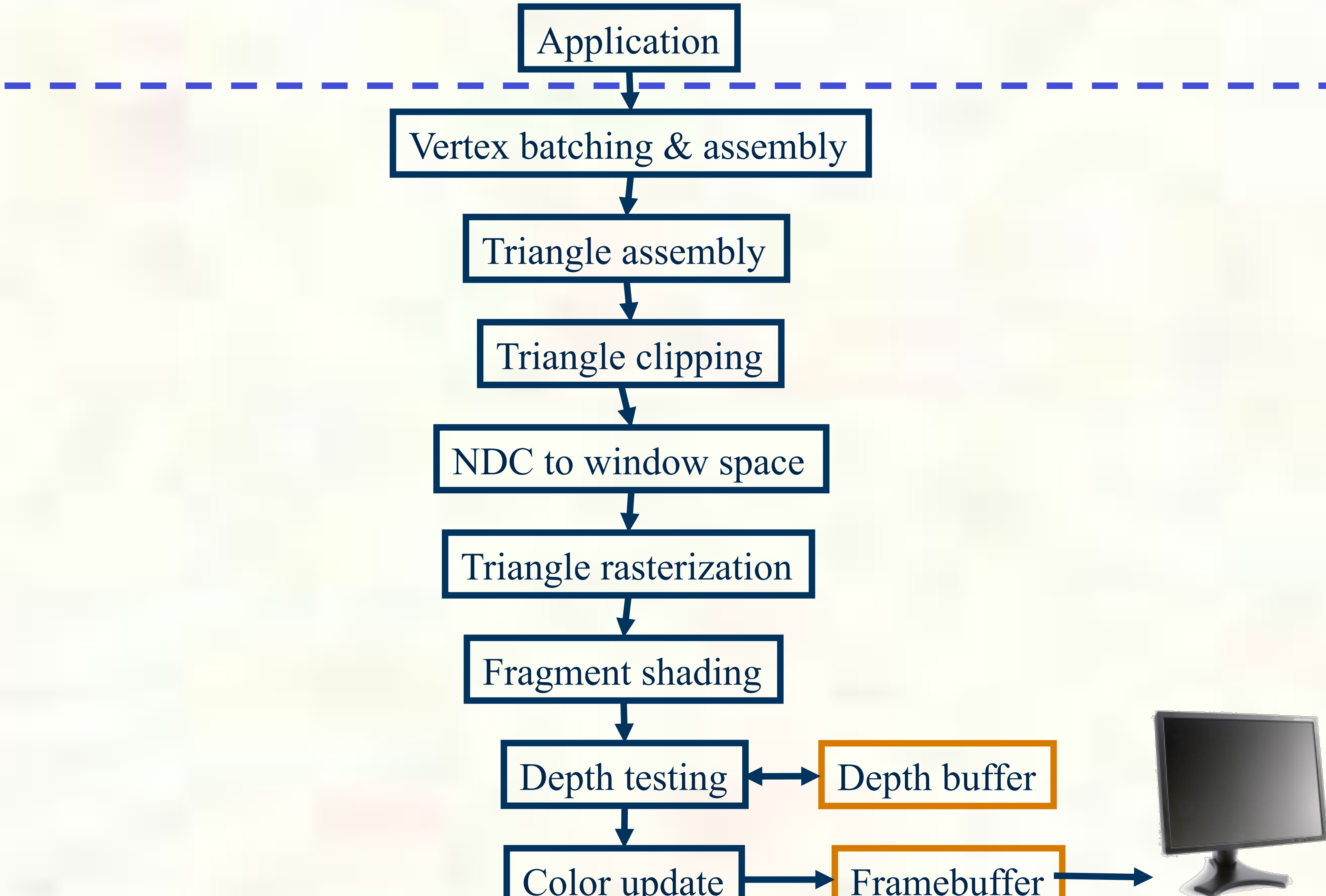
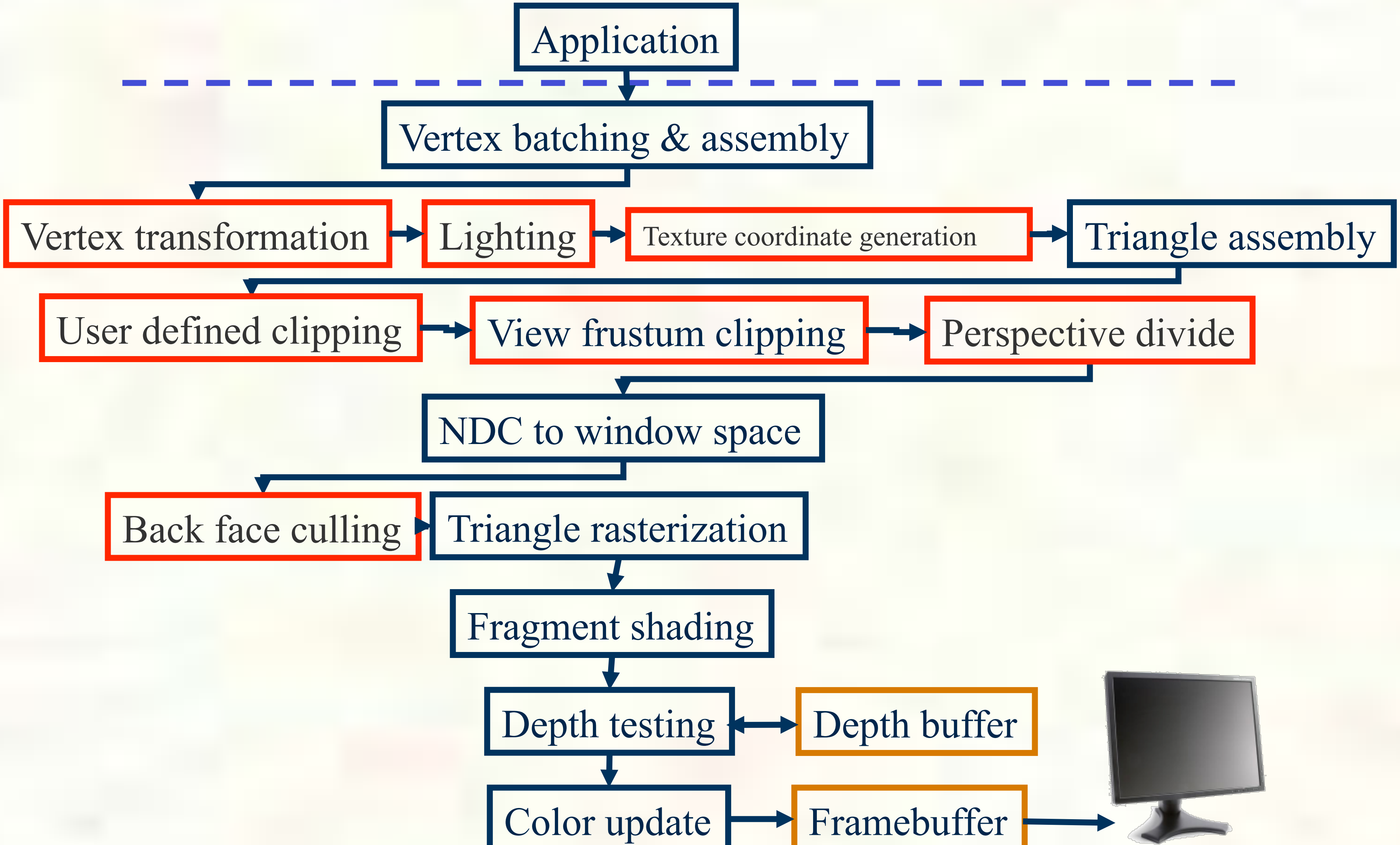
Viewing and Modeling

Don Fussell
Computer Science Department
The University of Texas at Austin

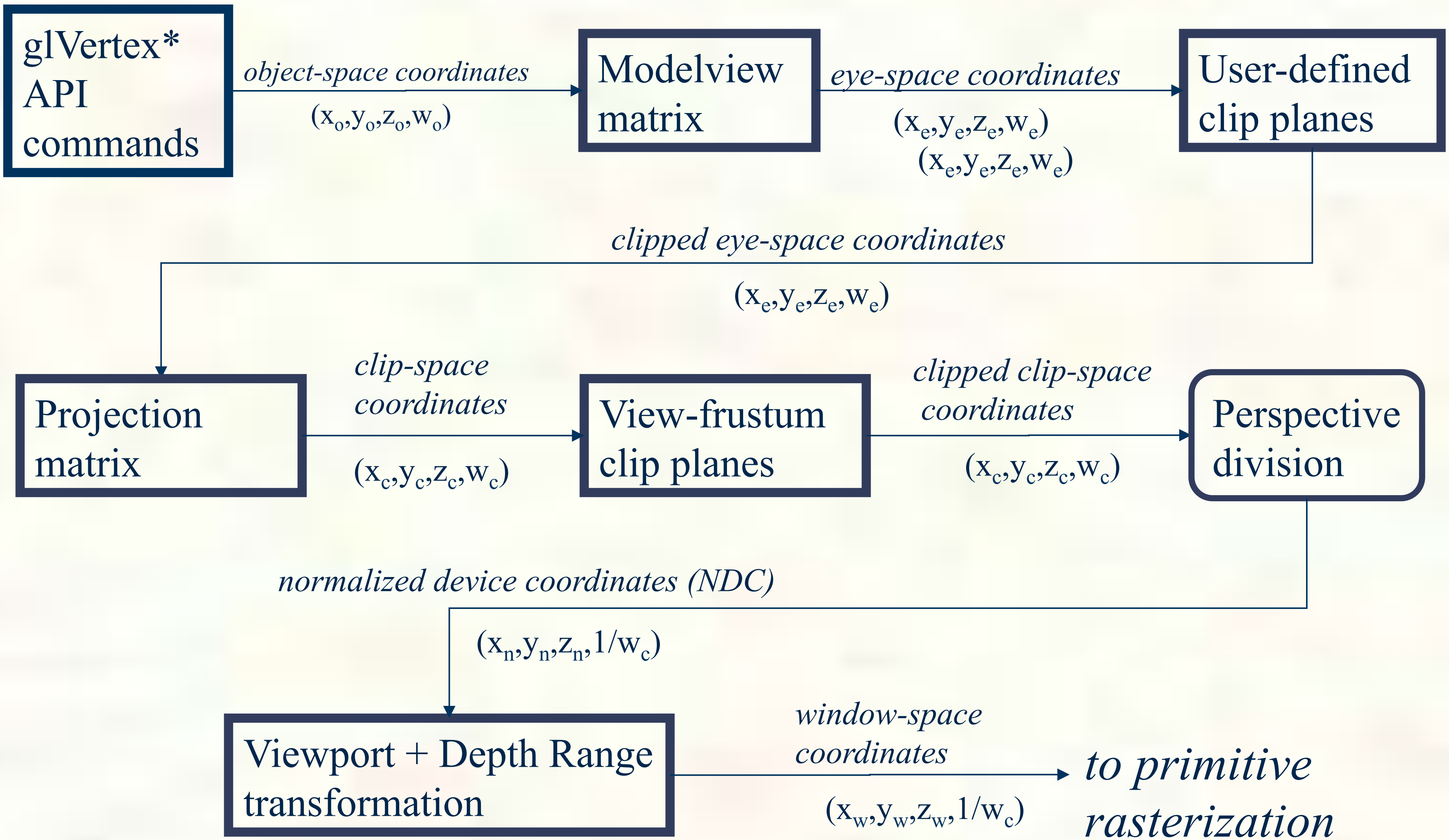
A Simplified Graphics Pipeline



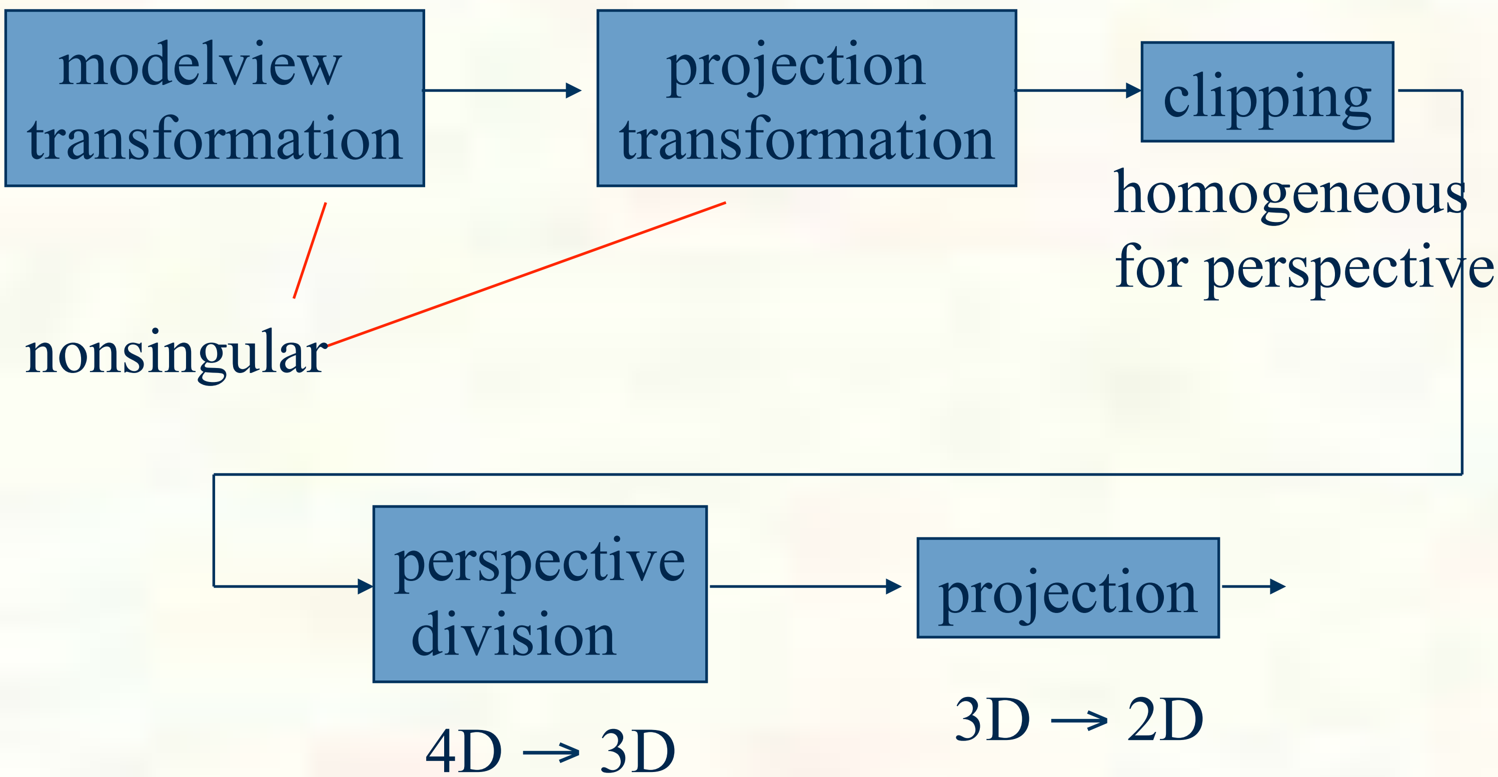
A few more steps expanded



Conceptual Vertex Transformation



Pipeline View



Computer Viewing

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume

The World and Camera Frames

- When we work with representations, we work with n-tuples or arrays of scalars
- Changes in frame are then defined by 4 x 4 matrices
- In OpenGL, the base frame that we start with is the world frame
- Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix
- Initially these frames are the same (**M**=**I**)

Vertex Transformation

- Object-space vertex position transformed by a general linear projective transformation
  - Expressed as a 4x4 matrix

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

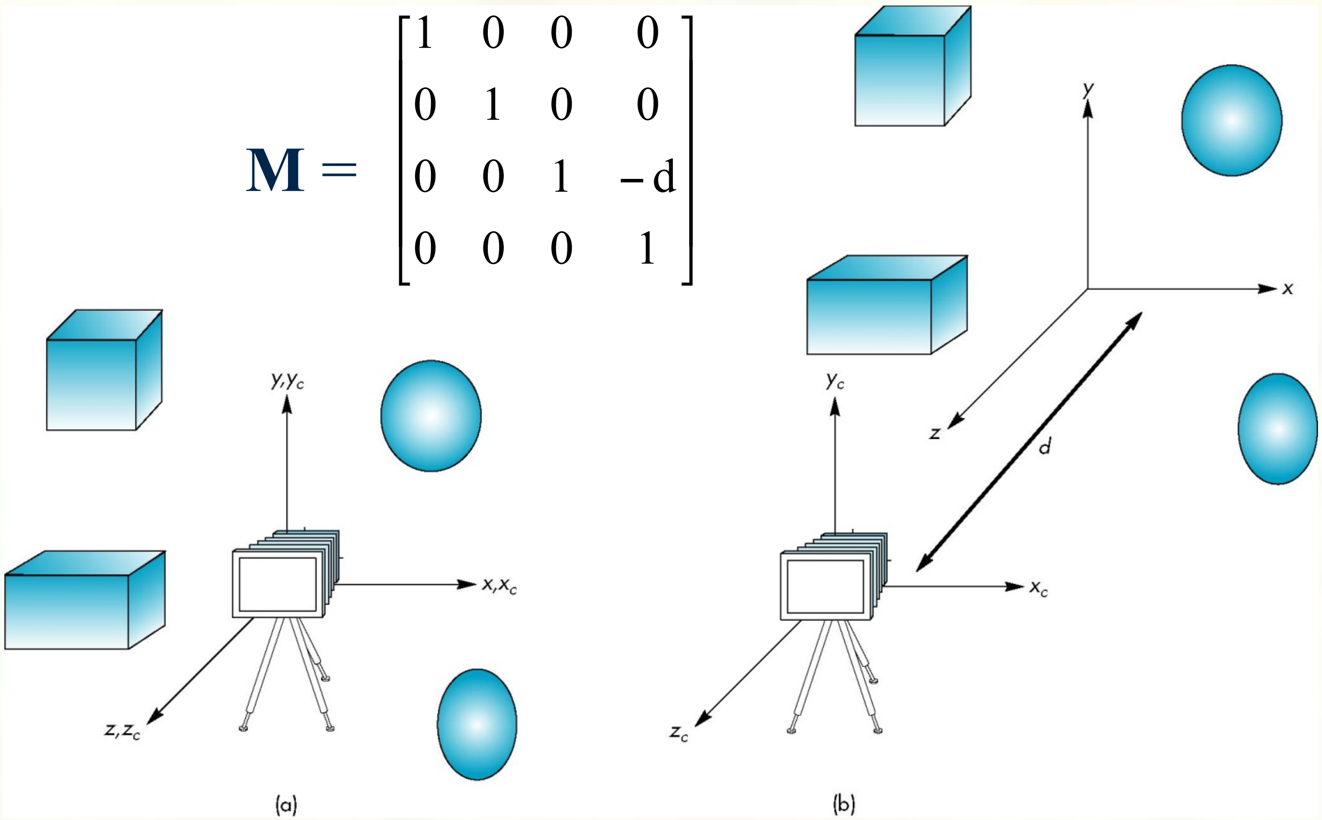
The OpenGL Camera

- In OpenGL, initially the object and camera frames are the same
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity

Moving the Camera

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

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



Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation (`glTranslatef(0.0,0.0,-d);`)
  - `d > 0`

Translate Transform

- Prototype
  - glTranslatef(GLfloat x, GLfloat y, GLfloat z)
- Post-concatenates this matrix

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

glTranslatef Matrix

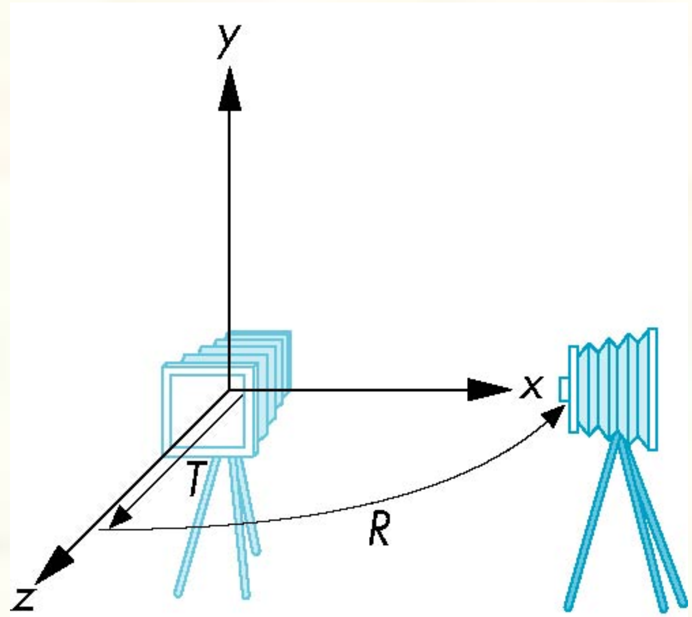
- Modelview specification
  - glLoadIdentity();
    glTranslatef(0,0,-14)
    - x translate=0, y translate=0, z translate=-14
    - Point at (0,0,0) would move to (0,0,-14)
      - Down the negative Z axis
- Matrix

the translation vector

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General Camera Motion

- We can position the camera anywhere by a sequence of rotations and translations
- Example: side view
  - Move camera to the origin
  - Rotate the camera
  - Model-view matrix C = RT



OpenGL code

- Remember that last transformation specified is first to be applied

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity();
glRotatef(90.0, 0.0, 1.0, 0.0);
glTranslatef(0.0, 0.0, -d);
```

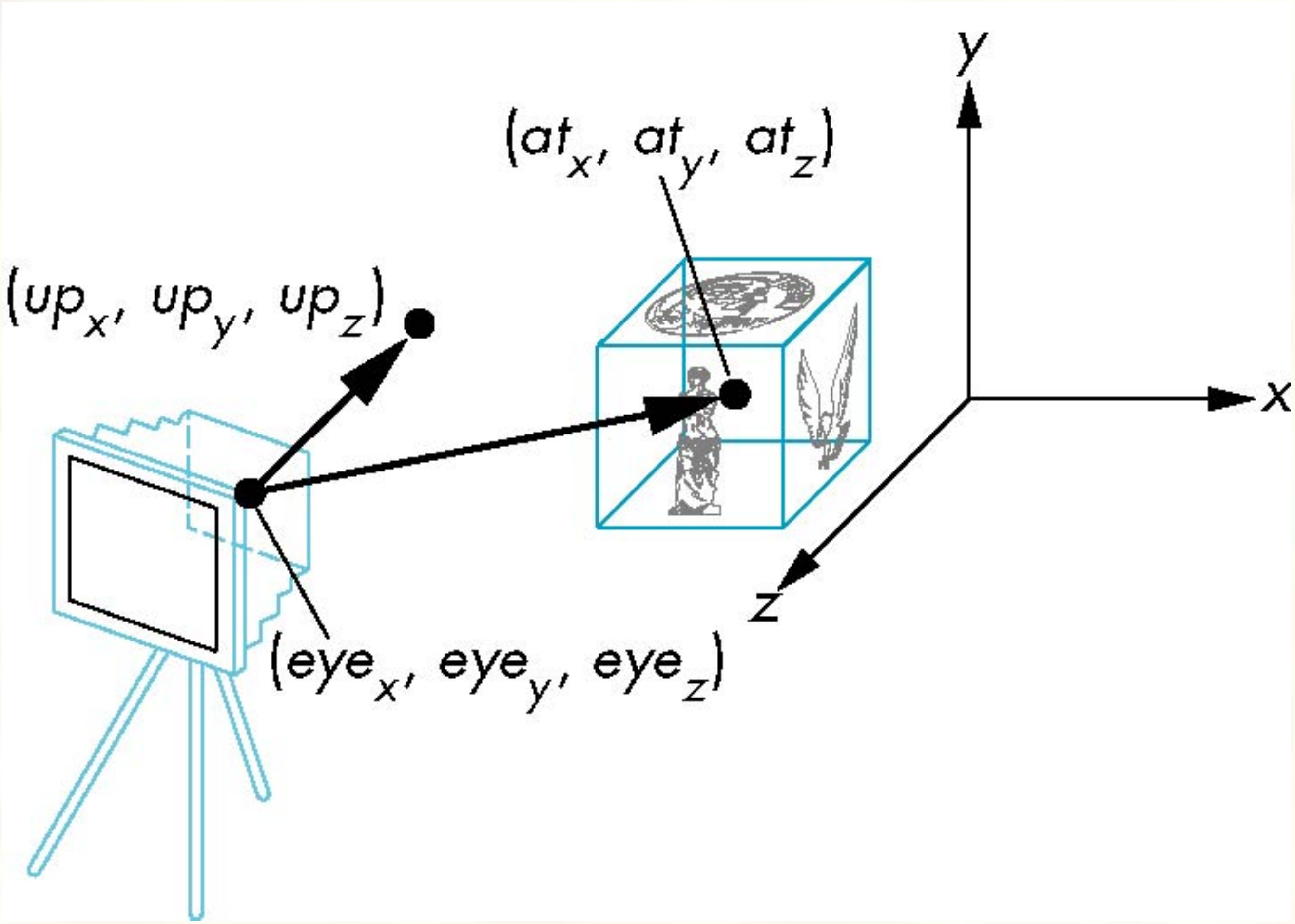
A Better Viewing Matrix

- "Look at" Transform
  - Concept
    - Given the following
      - a 3D world-space "eye" position
      - a 3D world-space center of view position (looking "at"), and
      - an 3D world-space "up" vector
    - Then an affine (non-projective) 4x4 matrix can be constructed
      - For a view transform mapping world-space to eye-space
- A ready implementation
  - The OpenGL Utility library (GLU) provides it
    - gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,
      GLdouble atx, GLdouble atz, GLdouble atz,
      GLdouble upx, GLdouble upy, GLdouble upz);

<u>Primary OpenGL libraries</u>
Link with –lglut for GLUT
Link with –lGLU for GLU
Link with –lGL for OpenGL

gluLookAt

`gluLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)`



"Look At" in Practice

- Consider our prior view situation
  - Instead of an arbitrary view…
  - …we just translated by 14 in negative Z direction
    - glTranslatef(0,0,14)
- What this means in "Look At" parameters
  - (eyex,eyey,eyez) = (0,0,14)
  - (atx,aty,atz) = (0,0,0)
  - (upx,upy,upz) = (0,1,0)

glTranslatef(0,0,-14)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Same matrix;
same transform

gluLookAt(0,0,14,
0,0,0,
0,1,0)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Not surprising both are "just translates in Z"
since the "Look At" parameters
already have use looking down the negative Z axis

The "Look At" Algorithm

- Vector math
  - Z = eye – at
  - Z = normalize(Z) /* normalize means Z / length(Z) */
  - Y = up
  - X = Y × Z   /* × means vector cross product! */
  - Y = Z × X   /* orthgonalize */
  - X = normalize(X)
  - Y = normalize(Y)
- Then build the following affine 4x4 matrix

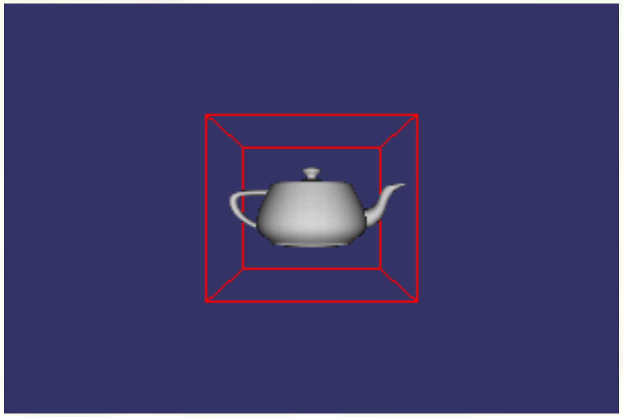
$$\begin{bmatrix} X_x & X_y & X_z & -X \bullet eye \\ Y_x & Y_y & Y_z & -Y \bullet eye \\ Z_x & Z_y & Z_z & -Z \bullet eye \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Warning: *Algorithm is prone to failure if normalize divides by zero (or very nearly does)*

So

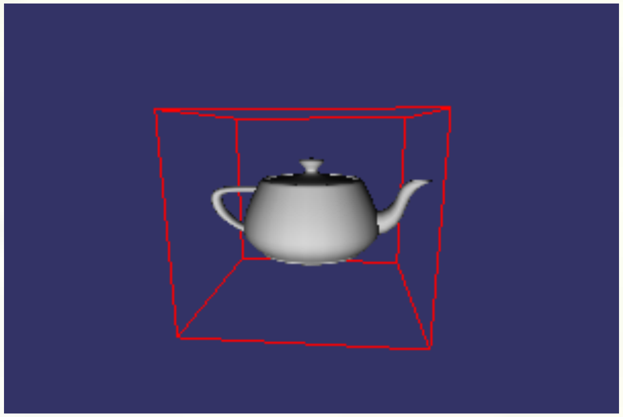
1. *Don't let Z or up be zero length vectors*
2. *Don't let Z and up be coincident vectors*

"Look At" Examples



```
gluLookAt(0,0,14,   // eye (x,y,z)
          0,0,0,    // at (x,y,z)
          0,1,0);   // up (x,y,z)
```

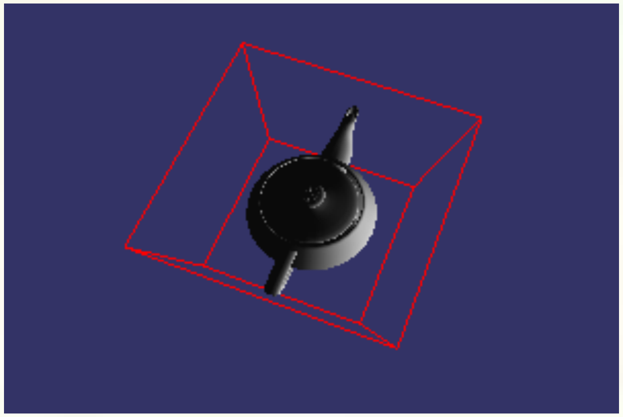
Same as the glTranslatef(0,0,-14) as expected



```
gluLookAt(1,2.5,11,  // eye (x,y,z)
          0,0,0,     // at (x,y,z)
          0,1,0);    // up (x,y,z)
```

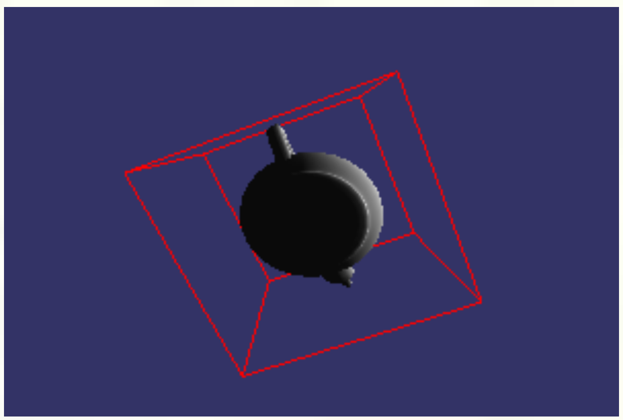
Similar to original, but just a little off angle due to slightly perturbed eye vector

"Look At" Major Eye Changes



```
gluLookAt(-2.5,11,1,    // eye (x,y,z)
          0,0,0,        // at (x,y,z)
          0,1,0);       // up (x,y,z)
```

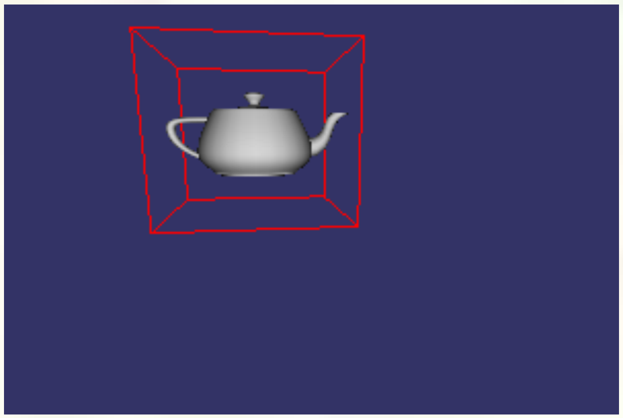
Eye is "above" the scene



```
gluLookAt(-2.5,-11,1,   // eye (x,y,z)
          0,0,0,        // at (x,y,z)
          0,1,0);       // up (x,y,z)
```

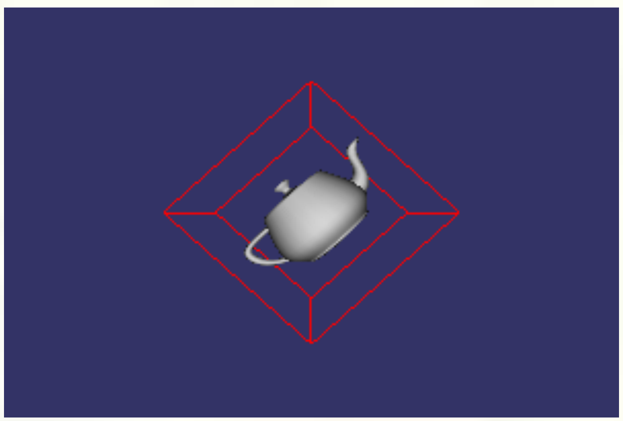
Eye is "below" the scene

"Look At" Changes to AT and UP



```
gluLookAt(0,0,14,   // eye (x,y,z)
          2,-3,0,   // at (x,y,z)
          0,1,0);   // up (x,y,z)
```

Original eye position, but "at" position shifted



```
gluLookAt(0,0,14,   // eye (x,y,z)
          0,0,0,    // at (x,y,z)
          1,1,0);   // up (x,y,z)
```

Eye is "below" the scene

The LookAt Function

- The GLU library contains the function gluLookAt to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
- Still need to initialize
- Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```
glMatrixMode(GL_MODELVIEW):
glLoadIdentity();
gluLookAt(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0., 1.0. 0.0);
```

Other Viewing APIs

- The LookAt function is only one possible API for positioning the camera
- Others include
  - View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles

Two Transforms in Sequence

- OpenGL thinks of the projective transform as really two 4x4 matrix transforms

FIRST
object-space
to
eye-space

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} MV_0 & MV_4 & MV_8 & MV_{12} \\ MV_1 & MV_5 & MV_9 & MV_{13} \\ MV_2 & MV_6 & MV_{10} & MV_{14} \\ MV_3 & MV_7 & MV_{11} & MV_{15} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

16 Multiply-Add operations

SECOND
eye-space
to
clip-space

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} P_0 & P_4 & P_8 & P_{12} \\ P_1 & P_5 & P_9 & P_{13} \\ P_2 & P_6 & P_{10} & P_{14} \\ P_3 & P_7 & P_{11} & P_{15} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Another
16 Multiply-Add
operations

Modelview-Projection Transform

- Matrixes can associate (combine)
  - Combination of the modelview and projection matrix = modelview-projection matrix
    - or often simply the "MVP" matrix

$$\begin{bmatrix} MVP_0 & MVP_4 & MVP_8 & MVP_{12} \\ MVP_1 & MVP_5 & MVP_9 & MVP_{13} \\ MVP_2 & MVP_6 & MVP_{10} & MVP_{14} \\ MVP_3 & MVP_7 & MVP_{11} & MVP_{15} \end{bmatrix} = \underbrace{\begin{bmatrix} P_0 & P_4 & P_8 & P_{12} \\ P_1 & P_5 & P_9 & P_{13} \\ P_2 & P_6 & P_{10} & P_{14} \\ P_3 & P_7 & P_{11} & P_{15} \end{bmatrix} \begin{bmatrix} MV_0 & MV_4 & MV_8 & MV_{12} \\ MV_1 & MV_5 & MV_9 & MV_{13} \\ MV_2 & MV_6 & MV_{10} & MV_{14} \\ MV_3 & MV_7 & MV_{11} & MV_{15} \end{bmatrix}}$$

Matrix multiplication
is **associative** (but not commutative)
A(BC) = (AB)C, but ABC≠CBA

concatenation is
64 Multiply-Add
operations, done by OpenGL driver

Specifying the Transforms

- Specified in two parts
- First the projection
  - glMatrixMode(GL_PROJECTION);
  - glLoadIdentity();
  - glFrustum(-4, +4, // left & right
    -3, +3, // top & bottom
    5, 80); // near & far
- Second the model-view
  - glMatrixMode(GL_MODELVIEW);
  - glLoadIdentity();
  - glTranslatef(0, 0, -14);
    - So objects centered at (0,0,0) would be at (0,0,-14) in eye-space

Resulting projection matrix

$$\begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Resulting modelview matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Modelview-Projection Matrix

- Transform composition via matrix multiplication

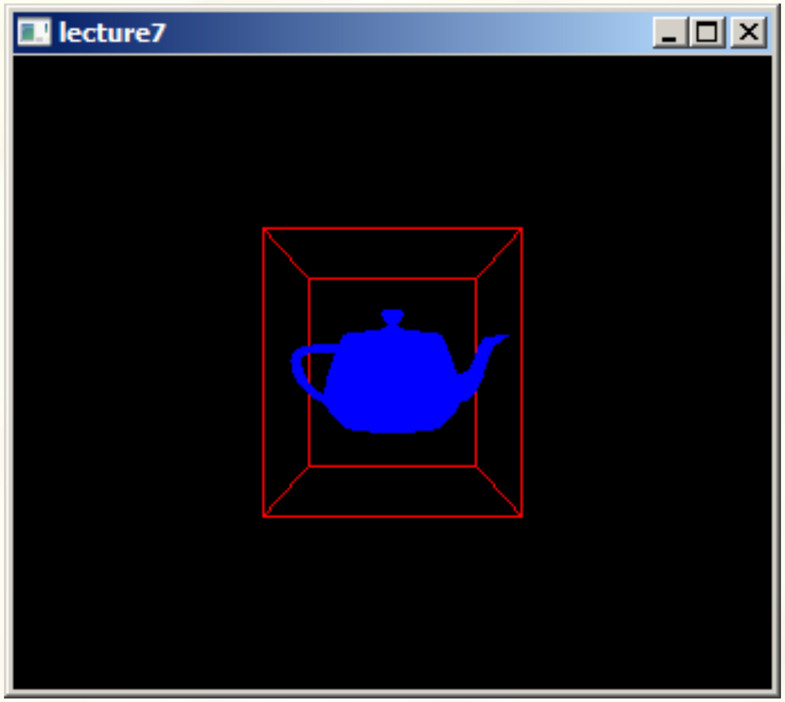
$$\begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & 5.2 \\ 0 & 0 & -1 & 14 \end{bmatrix}$$

Resulting modelview-projection matrix

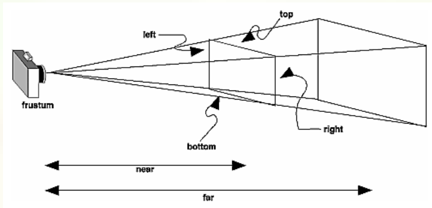
Now Draw Some Objects

- Draw a wireframe cube
  - glColor3f(1,0,0); // red
  - glutWireCube(6);
    - 6x6x6 unit cube centered at origin (0,0,0)
- Draw a teapot in the cube
  - glColor3f(0,0,1); // blue
  - glutSolidTeapot(2.0);
    - centered at the origin (0,0,0)
    - handle and spout point down the X axis
    - top and bottom in the Y axis
- *As we'd expect given a frustum transform, the cube is in perspective*
  - The teapot is too but more obvious to observe with a wireframe cube

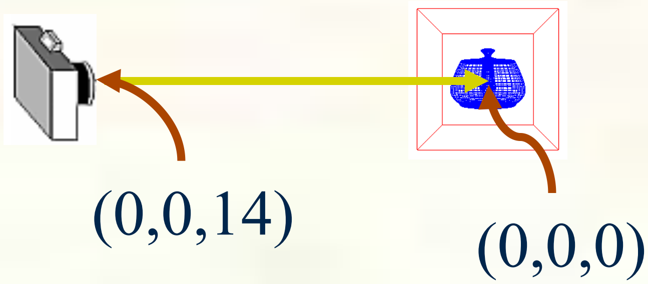


What We've Accomplished

- Simple perspective
  - With glFrustum
  - Establishes how eye-space maps to clip-space



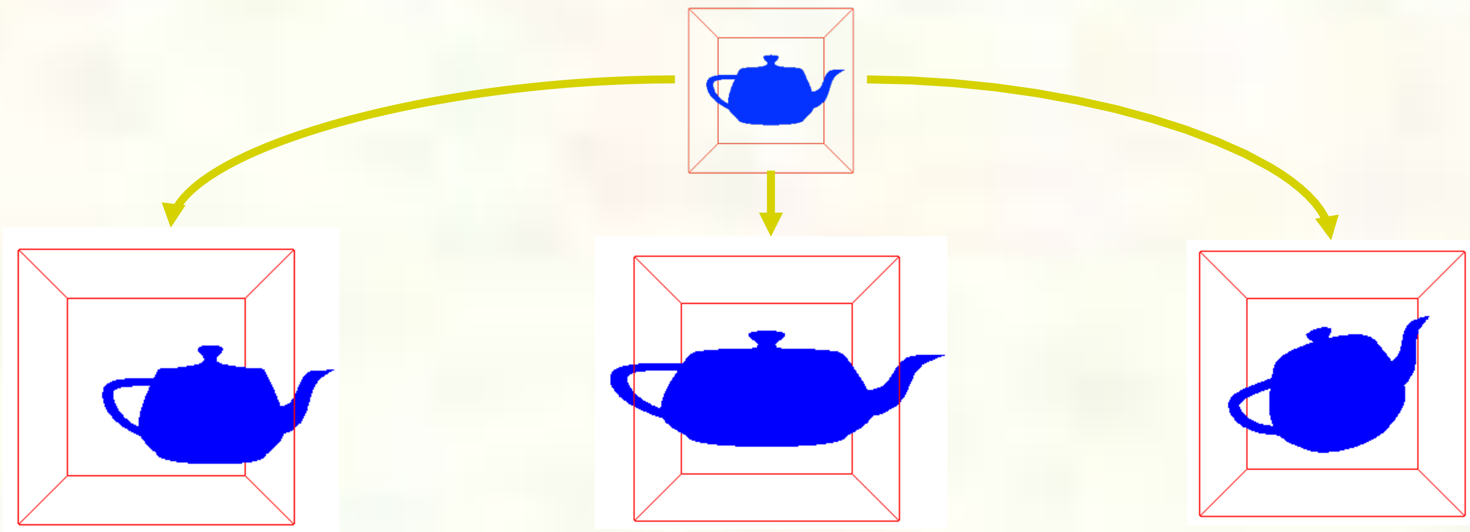
- Simple viewing
  - With glTranslatef
  - Establishes how world-space maps to eye-space



  - All we really did was "wheel" the camera 14 units up the Z axis
  - No actual "modeling transforms", just viewing
    - Modeling would be rotating, scaling, or otherwise transform the objects with the view
    - Arguably the modelview matrix is really just a "view" matrix in this example

Some Simple Modeling

- Try some modeling transforms to move teapot
  - But leave the cube alone for reference



```
glPushMatrix(); {
  glTranslatef(1.5, -0.5, 0);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

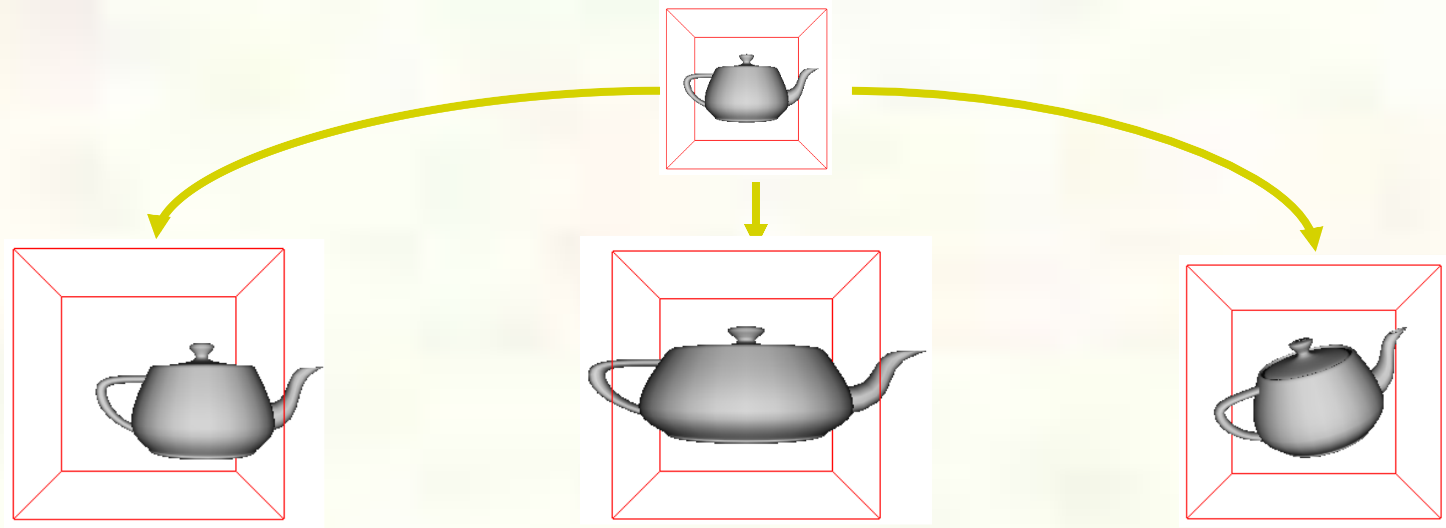
```
glPushMatrix(); {
  glScalef(1.5, 1.0, 1.5);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

```
glPushMatrix(); {
  glRotatef(30, 1,1,1);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

We "bracket" the modeling transform with glPushMatrix/glPopMatrix commands so the modeling transforms are "localized" to the particular object

Add Some Lighting

- Some lighting makes the modeling more intuitive



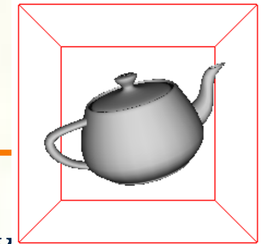
```
glPushMatrix(); {
  glTranslatef(1.5, -0.5, 0);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

```
glPushMatrix(); {
  glScalef(1.5, 1.0, 1.5);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

```
glPushMatrix(); {
  glRotatef(30, 1,1,1);
  glutSolidTeapot(2.0);
} glPopMatrix();
```

We've not discussed lighting yet but per-vertex lighting allows a virtual light source to "interact" with the object's surface orientation and material properties

Modelview-Projection Matrix



- Let's consider the "combined" modelview matrix with the rotation
  - glRotate(30, 1,1,1) defines a rotation matrix
    - Rotating 30 degrees…
    - …around an axis in the (1,1,1) direction

$$\begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\text{projection}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{view}} \underbrace{\begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{model}}$$

Combining All Three

Matrix-by-matrix multiplication is associative so
PVM = P (V M) = (P V) M

OpenGL keeps V and M "together" because eye-space is a convenient space for lighting

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{view}} \underbrace{\begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{model}}$$

$$\underbrace{\begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix}}_{\text{projection}} \underbrace{\begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{modelview}}$$

$$\underbrace{\begin{bmatrix} 1.1384 & -0.3050 & 0.4167 & 0 \\ 0.5556 & 1.5178 & -0.4067 & 0 \\ 0.2766 & -0.3778 & -1.0321 & 5.2 \\ 0.2440 & -0.3333 & -0.9107 & 14 \end{bmatrix}}_{\text{modelview-projection}}$$

Object- to Clip-space

model

$$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ w_{world} \end{bmatrix} = \begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{object} \\ y_{object} \\ z_{object} \\ w_{object} \end{bmatrix}$$

view

$$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ w_{world} \end{bmatrix}$$

projection

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix}$$

object-to-world-to-eye-to-clip

modelview

$$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix} = \begin{bmatrix} 0.9107 & -0.2440 & 0.3333 & 0 \\ 0.3333 & 0.9107 & -0.2440 & 0 \\ -0.2440 & 0.3333 & 0.9107 & -14 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{object} \\ y_{object} \\ z_{object} \\ w_{object} \end{bmatrix}$$

projection

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \begin{bmatrix} 1.25 & 0 & 0 & 0 \\ 0 & 1.667 & 0 & 0 \\ 0 & 0 & -1.1333 & -10.667 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix}$$

object-to-eye-to-clip

modelview-projection

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = \begin{bmatrix} 1.1384 & -0.3050 & 0.4167 & 0 \\ 0.5556 & 1.5178 & -0.4067 & 0 \\ 0.2766 & -0.3778 & -1.0321 & 5.2 \\ 0.2440 & -0.3333 & -0.9107 & 14 \end{bmatrix} \begin{bmatrix} x_{object} \\ y_{object} \\ z_{object} \\ w_{object} \end{bmatrix}$$

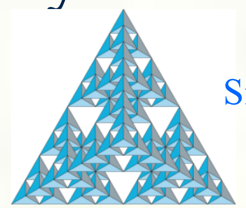
object-to-clip

Complex Scene Example

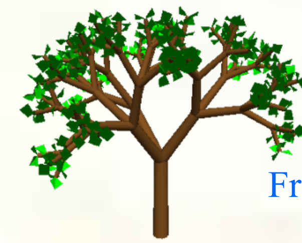
Each character, wall, ceiling, floor, and light have their own modeling transformation

Representing Objects

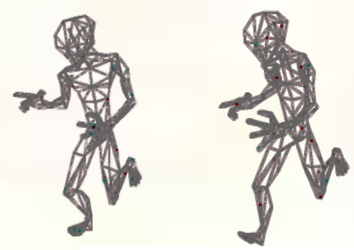
- Interested in object's boundary
- Various approaches
  - Procedural representations
    - Often fractal
  - Explicit polygon (triangle) meshes
    - By far, the most popular method
  - Curved surface patches
    - Often displacement mapped
  - Implicit representation
    - Blobby, volumetric



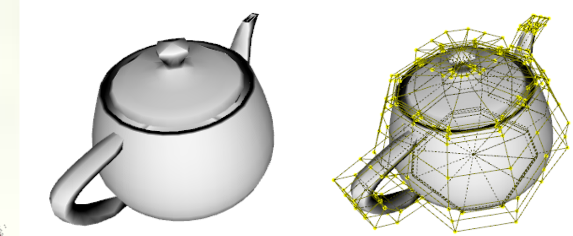
Sierpinski gasket



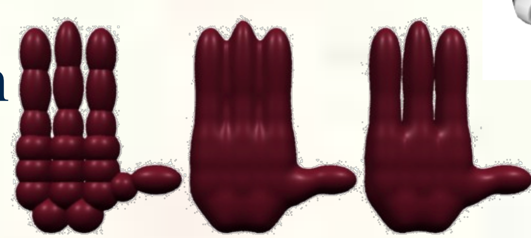
Fractal tree [Philip Winston]



Quake 2 key frame triangle meshes



Utah Teapot



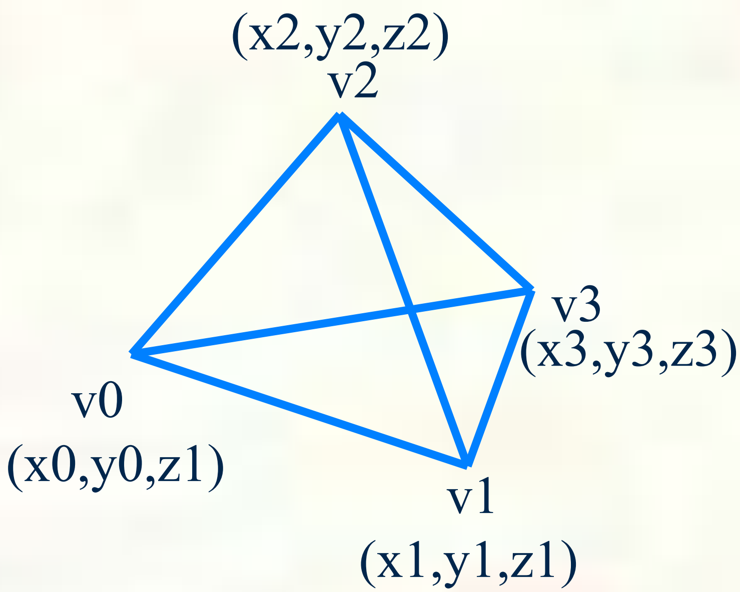
Blobby modeling in RenderMan

Focus on Triangle Meshes

- Easiest approach to representing object boundaries
- So what is a mesh and how should it be stored?
  - Simplest view
    - A set of triangles, each with its "own" 3 vertices
      - Essentially "triangle soup"
    - Yet triangles in meshes share edges by design
      - Sharing edges implies sharing vertices
  - More sophisticated view
    - Store single set of unique vertexes in array
    - Then each primitive (triangle) specifies 3 indices into array of vertexes
    - More compact
      - Vertex data size >> index size
      - Avoids redundant vertex data
    - Separates "topology" (how the mesh is connected) from its "geometry" (vertex positions and attributes)
      - Connectivity can be deduced more easily
      - Makes mesh processing algorithms easier
      - Geometry data can change without altering the topology

Consider a Tetrahedron

- Simplest closed volume
  - Consists of 4 triangles and 4 vertices
    - (and 4 edges)



triangle list	**vertex list**
0: v0,v1,v2	0: (x0,y0,z0)
1: v1,v3,v2	1: (x1,y1,z1)
2: v3,v0,v2	2: (x2,y2,z2)
3: v1,v0,v3	3: (x3,y3,z3)
topology	*geometry*
	potentially on-GPU!

Benefits of Vertex Array Approach

- Unique vertices are stored once
  - Saves memory
    - On CPU, on disk, and on GPU
- Matches OpenGL vertex array model of operation
  - And this matches the efficient GPU mode of operation
    - The GPU can “cache” post-transformed vertex results by vertex index
      - Saves retransformation and redundant vertex fetching
    - Direct3D has the same model
  - Allows vertex data to be stored on-GPU for even faster vertex processing
    - OpenGL supported vertex buffer objects for this

Next Lecture

- More about triangle mesh representation
- Scene graphs