

# Computer Graphics

---

Jeng-Sheng Yeh 葉正聖

Ming Chuan University

(modified from Bing-Yu Chen's slides)

# Introduction to OpenGL

---

- General OpenGL Introduction
- An Example OpenGL Program
- Drawing with OpenGL
- Transformations
- Animation and Depth Buffering
- Lighting
- Evaluation and NURBS
- Texture Mapping
- Advanced OpenGL Topics
- Imaging

modified from  
Dave Shreiner, Ed Angel, and Vicki Shreiner.  
**An Interactive Introduction to OpenGL Programming.**  
*ACM SIGGRAPH 2001 Conference Course Notes #54.*  
& *ACM SIGGRAPH 2004 Conference Course Notes #29.*

# Transformations in OpenGL

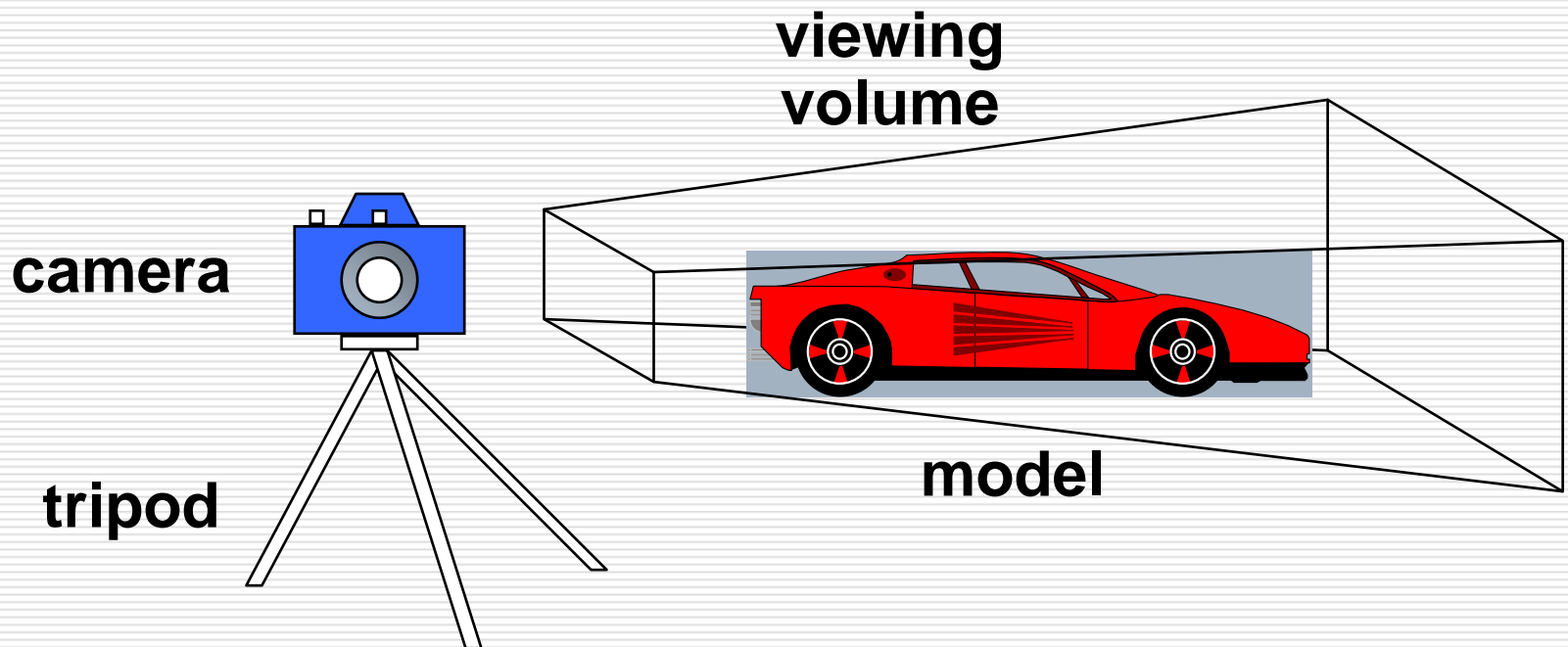
---

- Modeling
  - Viewing
    - orient camera
    - projection
  - Animation
  - Map to screen
-

# Camera Analogy

---

- 3D is just like taking a photograph (lots of photographs!)



# Camera Analogy & Transformations

---

- 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
  - Viewport transformations
    - enlarge or reduce the physical photograph
-

# Coordinate Systems & Transformations

---

- Steps in Forming an Image
    - specify geometry (world coordinates)
    - specify camera (camera coordinates)
    - project (window coordinates)
    - map to viewport (screen coordinates)
  - Each step uses transformations
  - Every transformation is equivalent to a change in coordinate systems (frames)
-

# Affine Transformations

---

- Want transformations which preserve geometry
    - lines, polygons, quadrics
  - Affine = line preserving
    - Rotation, translation, scaling
    - Projection
    - Concatenation (composition)
-

# Homogeneous Coordinates

---

- each vertex is a column vector

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

- $w$  is usually 1.0
  - all operations are matrix multiplications
  - directions (directed line segments) can be represented with  $w = 0.0$
-



# 3D Transformations

---

- A vertex is transformed by 4 x 4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
  - matrices are always post-multiplied
  - product of matrix and vector is  $\mathbf{M}\vec{v}$

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

---

# Specifying Transformations

---

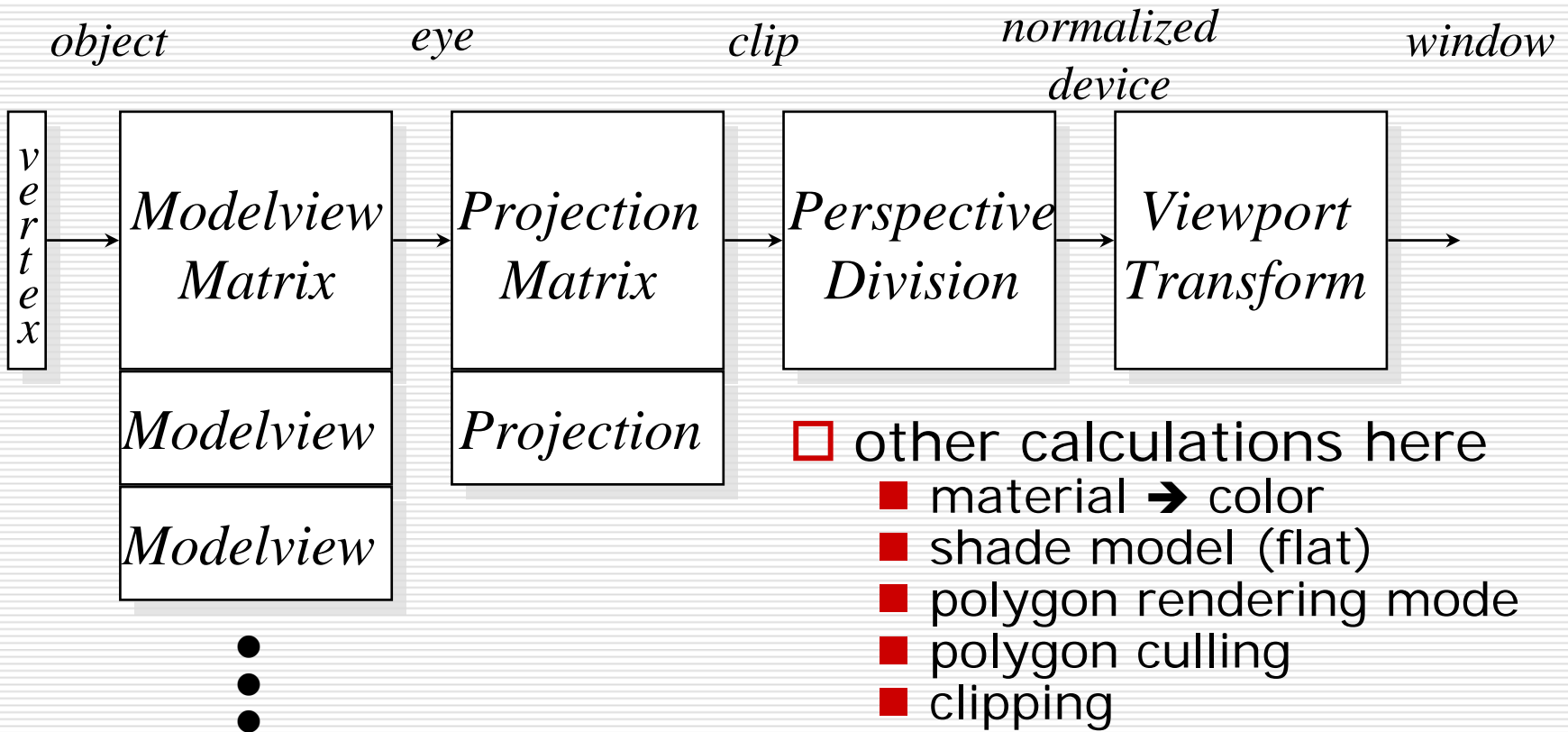
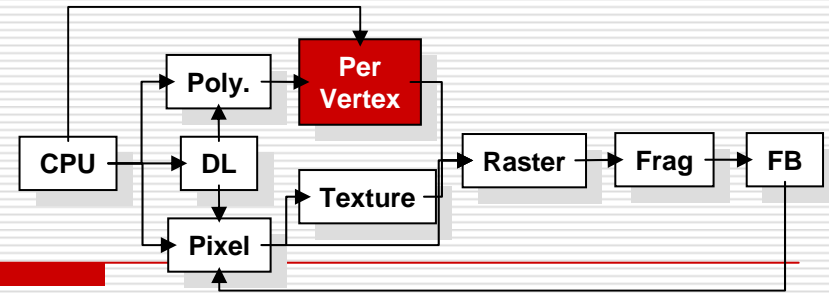
- Programmer has two styles of specifying transformations
    - specify matrices (`glLoadMatrix`, `glMultMatrix`)
    - specify operation (`glRotate`, `glOrtho`)
  
  - Programmer does not have to remember the exact matrices
    - check appendix of Red Book (Programming Guide)
-

# Programming Transformations

---

- Prior to rendering, view, locate, and orient:
    - eye/camera position
    - 3D geometry
  - Manage the matrices
    - including matrix stack
  - Combine (composite) transformations
-

# Transformation Pipeline



# OpenGL Matrices

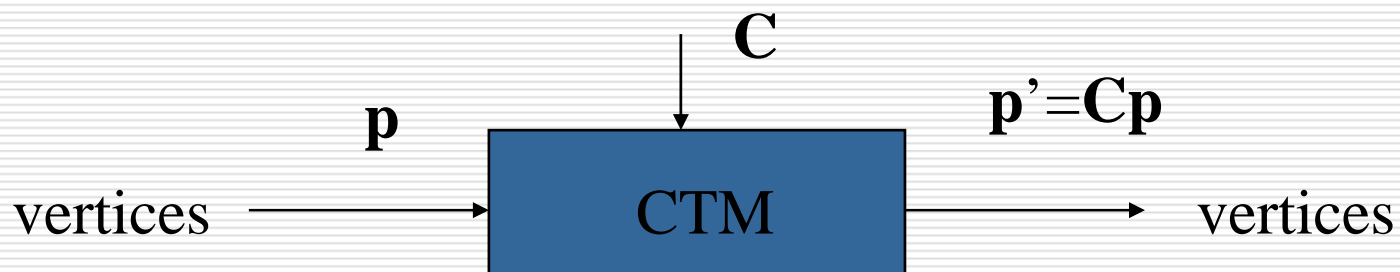
---

- ❑ In OpenGL matrices are part of the state
  - ❑ Three types
    - Model-View (`GL_MODEL_VIEW`)
    - Projection (`GL_PROJECTION`)
    - Texture (`GL_TEXTURE`) (ignore for now)
  - ❑ Single set of functions for manipulation
  - ❑ Select which to manipulated by
    - `glMatrixMode(GL_MODEL_VIEW);`
    - `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 CTM can be altered either by loading a new CTM or by postmultiplication
    - Load an identity matrix:  $\mathbf{C} \leftarrow \mathbf{I}$
    - Load an arbitrary matrix:  $\mathbf{C} \leftarrow \mathbf{M}$
  
    - Load a translation matrix:  $\mathbf{C} \leftarrow \mathbf{T}$
    - Load a rotation matrix:  $\mathbf{C} \leftarrow \mathbf{R}$
    - Load a scaling matrix:  $\mathbf{C} \leftarrow \mathbf{S}$
  
    - Postmultiply by an arbitrary matrix:  $\mathbf{C} \leftarrow \mathbf{C}\mathbf{M}$
    - Postmultiply by a translation matrix:  $\mathbf{C} \leftarrow \mathbf{C}\mathbf{T}$
    - Postmultiply by a rotation matrix:  $\mathbf{C} \leftarrow \mathbf{C}\mathbf{R}$
    - Postmultiply by a scaling matrix:  $\mathbf{C} \leftarrow \mathbf{C}\mathbf{S}$
-

# Rotation about a Fixed Point

---

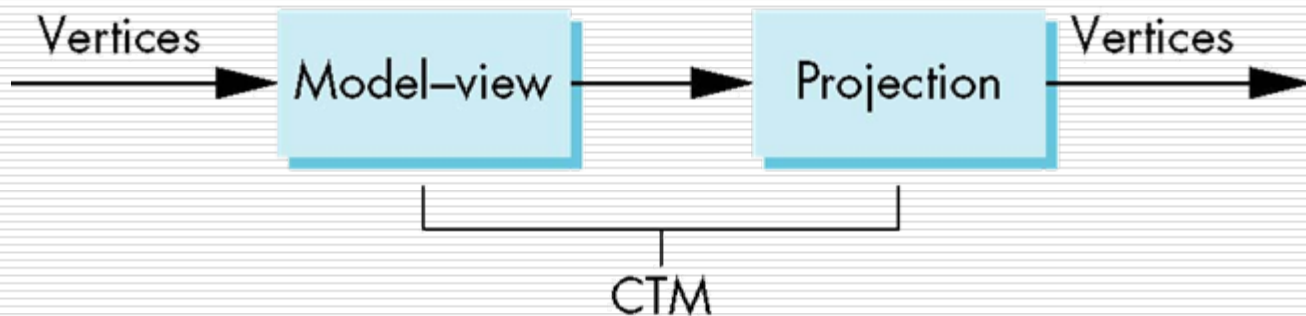
- Start with identity matrix:  $\mathbf{C} \leftarrow \mathbf{I}$
  - Move fixed point to origin:  $\mathbf{C} \leftarrow \mathbf{CT}^{-1}$
  - Rotate:  $\mathbf{C} \leftarrow \mathbf{CR}$
  - Move fixed point back:  $\mathbf{C} \leftarrow \mathbf{CT}$
  
  - Result:  $\mathbf{C} = \mathbf{T}^{-1}\mathbf{RT}$
  
  - 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 matrix mode



# Matrix Operations

---

- Specify Current Matrix Stack

`glMatrixMode( GL_MODELVIEW or GL_PROJECTION )`

- Other Matrix or Stack Operations

`glLoadIdentity()`

`glPushMatrix()`

`glPopMatrix()`

- Viewport

- usually same as window size

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

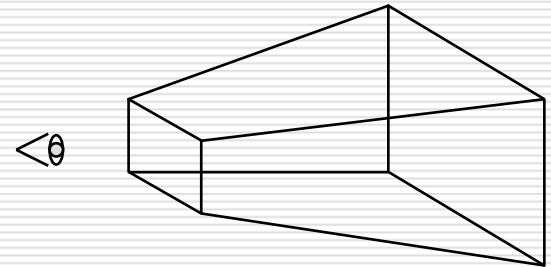
`glViewport( x, y, width, height )`

---

# Projection Transformation

---

- ❑ Shape of viewing frustum
- ❑ Perspective projection



`gluPerspective( fovy, aspect, zNear, zFar )`

`glFrustum( left, right, bottom, top, zNear, zFar )`

- ❑ Orthographic parallel projection

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

`gluOrtho2D( left, right, bottom, top )`

- ❑ calls `glOrtho` with z values near zero
-

# Applying Projection Transformations

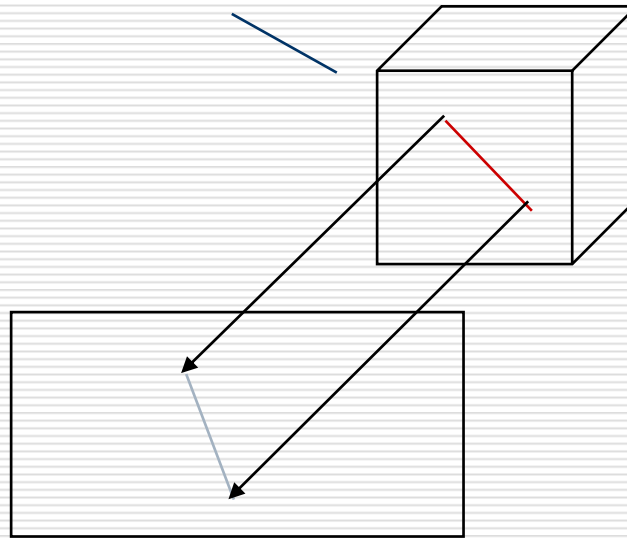
---

- Typical use (orthographic projection)

```
glMatrixMode( GL_PROJECTION );
```

```
glLoadIdentity();
```

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



# Viewing Transformations

---

□ Position the camera/eye in the scene

■ place the tripod down; aim camera

□ To “fly through” a scene

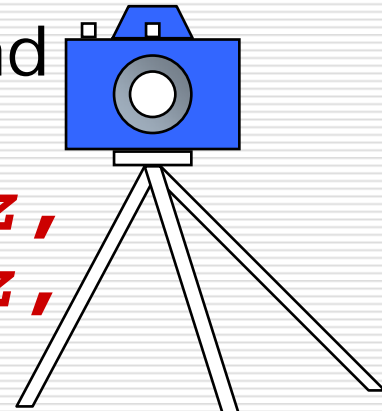
■ change viewing transformation and redraw scene

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

■ up vector determines unique orientation

■ careful of degenerate positions

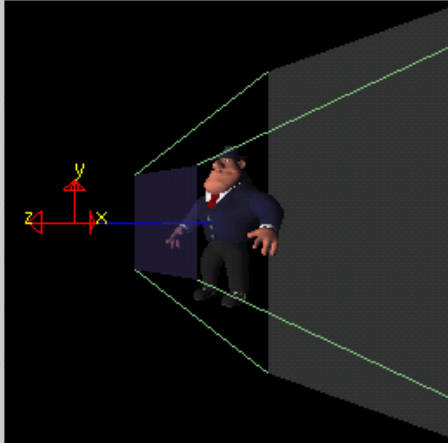
tripod




# Projection Tutorial

Projection

World-space view



Screen-space view



Command manipulation window

```
fovy aspect zNear zFar  
gluPerspective( 60.0 , 1.00 , 1.0 , 10.0 );  
gluLookAt( 0.00 , 0.00 , 2.00 , <- eye  
          0.00 , 0.00 , 0.00 , <- center  
          0.00 , 1.00 , 0.00 ); <- up
```

Click on the arguments and move the mouse to modify values.

# Modeling Transformations

---

- Move object

```
glTranslate{fd}( x, y, z )
```

- Rotate object around arbitrary axis (x y z)

```
glRotate{fd}( angle, x, y, z )
```

- angle is in degrees

- Dilate (stretch or shrink) or mirror object

```
glScale{fd}( x, y, z )
```

---

# Example

---

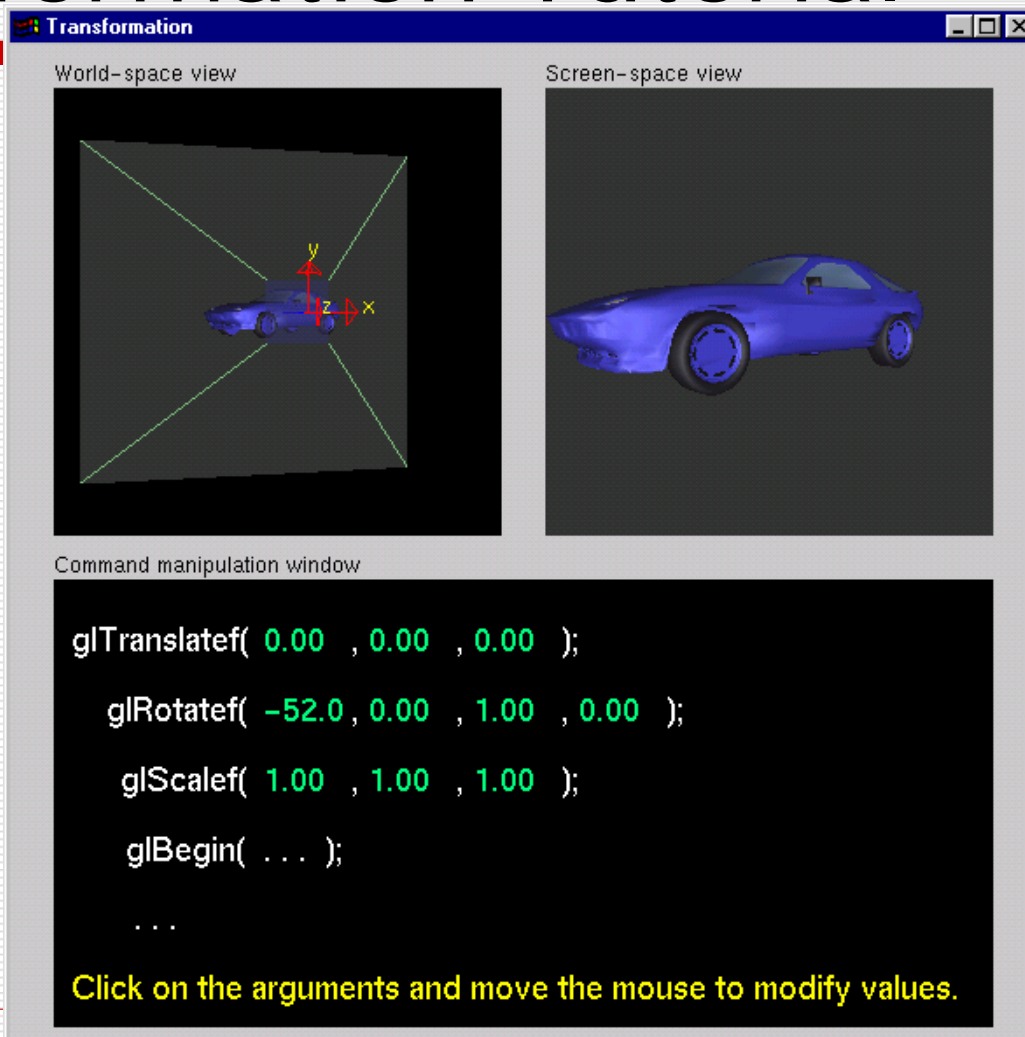
- Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(1.0, 2.0, 3.0);  
glRotatef(30.0, 0.0, 0.0, .10);  
glTranslatef(-1.0, -2.0, -3.0);
```

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



# Transformation Tutorial



The screenshot shows a window titled "Transformation" with two main views and a command manipulation window. The "World-space view" on the left shows a blue car centered in a 3D coordinate system with x, y, and z axes. The "Screen-space view" on the right shows the same car from a perspective view. Below these views is a "Command manipulation window" containing the following OpenGL code:

```
glTranslatef( 0.00 , 0.00 , 0.00 );  
glRotatef( -52.0 , 0.00 , 1.00 , 0.00 );  
glScalef( 1.00 , 1.00 , 1.00 );  
glBegin( ... );  
...  
Click on the arguments and move the mouse to modify values.
```

# 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
    - 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 *enquiry (query)* functions
    - **glGetIntegerv**
    - **glGetFloatv**
    - **glGetBooleanv**
    - **glGetDoublev**
    - **glIsEnabled**
  - For matrices, we use as
    - **double m[16];**
    - **glGetFloatv(GL\_MODELVIEW, m);**
-

# Connection:

## Viewing and Modeling

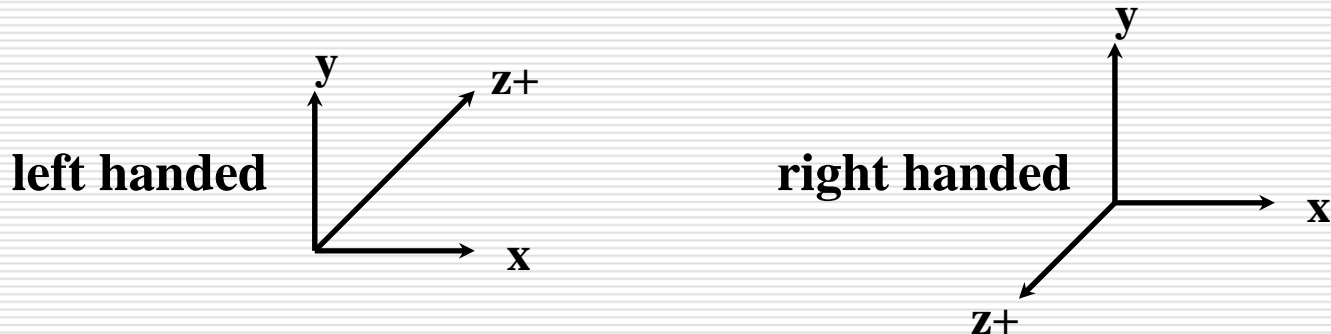
---

- Moving camera is equivalent to moving every object in the world towards a stationary camera
  - Viewing transformations are equivalent to several modeling transformations
    - `gluLookAt()` has its own command
    - can make your own *polar view* or *pilot view*
-

# Projection is left handed

---

- Projection transformations (**gluPerspective**, **glOrtho**) are left handed
  - think of *zNear* and *zFar* as distance from view point
- Everything else is right handed, including the vertexes to be rendered



# Common Transformation Usage

---

- 3 examples of **resize()** routine
    - restate projection & viewing transformations
  - Usually called when window resized
  - Registered as callback for **glutReshapeFunc()**
-

# resize() :

## Perspective & LookAt

---

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLdouble) w / h,
                   1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0,
              0.0, 0.0, 0.0,
              0.0, 1.0, 0.0 );
}
```

---



# resize() :

## Perspective & Translate

---

Same effect as previous LookAt

```
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLdouble) w/h,
                  1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -5.0 );
}
```

---

# resize(): Ortho (part 1)

---

```
void resize( int width, int height )
{
    GLdouble aspect = (GLdouble) width / height;
    GLdouble left = -2.5, right = 2.5;
    GLdouble bottom = -2.5, top = 2.5;
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    ... continued ...
```

---

# resize():

## Ortho (part 2)

---

```
if ( aspect < 1.0 ) {
    left /= aspect;
    right /= aspect;
} else {
    bottom *= aspect;
    top *= aspect;
}
glOrtho( left, right, bottom, top, near,
far );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
}
```

---

# Compositing Modeling Transformations

---

- Problem 1: hierarchical objects
    - one position depends upon a previous position
    - robot arm or hand; sub-assemblies
  
  - Solution 1: moving local coordinate system
    - modeling transformations move coordinate system
    - post-multiply column-major matrices
    - OpenGL post-multiplies matrices
-

# Compositing

## Modeling Transformations

---

- Problem 2: objects move relative to absolute world origin
    - my object rotates around the wrong origin
      - make it spin around its center or something else
  - Solution 2: fixed coordinate system
    - modeling transformations move objects around fixed coordinate system
    - pre-multiply column-major matrices
    - OpenGL post-multiplies matrices
    - must reverse order of operations to achieve desired effect
-

# Additional Clipping Planes

---

- At least 6 more clipping planes available
- Good for cross-sections
- Modelview matrix moves clipping plane  $Ax + By + Cz + D < 0$  clipped

`glEnable( GL_CLIP_PLANEi )`

`glClipPlane( GL_CLIP_PLANEi, GLdouble* coeff )`

---

# Reversing Coordinate Projection

---

- Screen space back to world space

```
glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )
glGetDoublev( GL_MODELVIEW_MATRIX,
              GLdouble mvmatrix[16] )
glGetDoublev( GL_PROJECTION_MATRIX,
              GLdouble projmatrix[16] )
gluUnProject( GLdouble winx, winy, winz,
              mvmatrix[16], projmatrix[16],
              GLint viewport[4],
              GLdouble *objx, *objy, *objz )
```

- `gluProject` goes from world to screen space
-

# Smooth Rotation

---

- From a practical standpoint, we often want to use transformations to move and reorient an object smoothly
    - Problem: find a sequence of model-view matrices  $\mathbf{M}_0, \mathbf{M}_1, \dots, \mathbf{M}_n$  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
-



# Incremental Rotation

---

- Consider the two approaches
    - For a sequence of rotation matrices  $\mathbf{R}_0, \mathbf{R}_1, \dots, \mathbf{R}_n$ , 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 the angle
  - Quaternions can be more efficient than either
-

# Quaternions

---

- Extension of imaginary numbers from 2 to 3 dimensions
  - Requires one real and three imaginary components **i, j, k**
    - $q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} = [\mathbf{w}, \mathbf{v}]; \mathbf{w} = q_0, \mathbf{v} = (q_1, q_2, q_3)$
    - where  $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$
    - **w** is called **scalar** and **v** is called **vector**
  - Quaternions can express rotations on sphere smoothly and efficiently. Process:
    - Model-view matrix → Quaternion
    - Carry out operations with Quaternions
    - Quaternion → Model-view matrix
-

# Basic Operations Using Quaternions

---

## □ Addition

- $q + q' = [w + w', v + v']$

## □ Multiplication

- $q \cdot q' = [w \cdot w' - v \cdot v', v \times v' + w \cdot v' + w' \cdot v]$

## □ Conjugate

- $q^* = [w, -v]$

## □ Length

- $|q| = (w^2 + |v|^2)^{1/2}$

## □ Norm

- $N(q) = |q|^2 = w^2 + |v|^2 = w^2 + x^2 + y^2 + z^2$

## □ Inverse

- $q^{-1} = q^* / |q|^2 = q^* / N(q)$

## □ Unit Quaternion

- $q$  is a unit quaternion if  $|q| = 1$  and then  $q^{-1} = q^*$

## □ Identity

- $[1, (0, 0, 0)]$  (when involving multiplication)

- $[0, (0, 0, 0)]$  (when involving addition)

---

# Angle and Axis & Euler Angles

---

## □ Angle and Axis

- $q = [\cos(\theta/2), \sin(\theta/2) \cdot v]$

## □ Euler Angles

- $q = q_{\text{yaw}} \cdot q_{\text{pitch}} \cdot q_{\text{roll}}$

- $q_{\text{roll}} = [\cos(y/2), (\sin(y/2), 0, 0)]$

- $q_{\text{pitch}} = [\cos(q/2), (0, \sin(q/2), 0)]$

- $q_{\text{yaw}} = [\cos(f/2), (0, 0, \sin(f/2))]$

---

# Matrix-to-Quaternion Conversion

---

```
MatToQuat (float m[4][4], QUAT * quat) {
    float tr, s, q[4];
    int i, j, k;
    int nxt[3] = {1, 2, 0};
    tr = m[0][0] + m[1][1] + m[2][2];
    if (tr > 0.0) {
        s = sqrt (tr + 1.0);
        quat->w = s / 2.0;
        s = 0.5 / s;
        quat->x = (m[1][2] - m[2][1]) * s;
        quat->y = (m[2][0] - m[0][2]) * s;
        quat->z = (m[0][1] - m[1][0]) * s;
    } else {
        i = 0;
        if (m[1][1] > m[0][0]) i = 1;
        if (m[2][2] > m[i][i]) i = 2;
        j = nxt[i];
        k = nxt[j];
        s = sqrt ((m[i][i] - (m[j][j] + m[k][k])) + 1.0);
        q[i] = s * 0.5;
        if (s != 0.0) s = 0.5 / s;
        q[3] = (m[j][k] - m[k][j]) * s;
        q[j] = (m[i][j] + m[j][i]) * s;
        q[k] = (m[i][k] + m[k][i]) * s;
        quat->x = q[0];
        quat->y = q[1];
        quat->z = q[2];
        quat->w = q[3];
    }
}
```

---

# Quaternion-to-Matrix Conversion

---

```
QuatToMatrix (QUAT * quat, float m[4][4]) {  
    float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;  
    x2 = quat->x + quat->x; y2 = quat->y + quat->y;  
    z2 = quat->z + quat->z;  
    xx = quat->x * x2; xy = quat->x * y2; xz = quat->x * z2;  
    yy = quat->y * y2; yz = quat->y * z2; zz = quat->z * z2;  
    wx = quat->w * x2; wy = quat->w * y2; wz = quat->w * z2;  
    m[0][0] = 1.0 - (yy + zz); m[1][0] = xy - wz;  
    m[2][0] = xz + wy; m[3][0] = 0.0;  
    m[0][1] = xy + wz; m[1][1] = 1.0 - (xx + zz);  
    m[2][1] = yz - wx; m[3][1] = 0.0;  
    m[0][2] = xz - wy; m[1][2] = yz + wx;  
    m[2][2] = 1.0 - (xx + yy); m[3][2] = 0.0;  
    m[0][3] = 0; m[1][3] = 0;  
    m[2][3] = 0; m[3][3] = 1;  
}
```

---

# SLERP-Spherical Linear intERPolation

---

- Interpolate between two quaternion rotations along the shortest arc.

- $$\text{SLERP}(p, q, t) = \frac{p \cdot \sin((1-t) \cdot \theta) + q \cdot \sin(t \cdot \theta)}{\sin(\theta)}$$

- where 
$$\begin{aligned} \cos(\theta) &= w_p \cdot w_q + v_p \cdot v_q \\ &= w_p \cdot w_q + x_p \cdot x_q + y_p \cdot y_q + z_p \cdot z_q \end{aligned}$$

- If two orientations are too close, use linear interpolation to avoid any divisions by zero.
-