Recall: Function Calls to Create Transform Matrices

- Previously made function calls to generate 4x4 matrices for identity, translate, scale, rotate transforms
- Put transform matrix into **CTM**
- Example

```
mat4 m = Identity();
```

→

CTM Matrix

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

Arbitrary Matrices

- Can multiply by matrices from transformation commands (Translate, Rotate, Scale) into CTM
- Can also load arbitrary 4x4 matrices into CTM

Load into
CTM Matrix

$$\begin{pmatrix} 1 & 0 & 15 & 3 \\ 0 & 2 & 0 & 12 \\ 34 & 0 & 3 & 12 \\ 0 & 24 & 0 & 1 \end{pmatrix}$$

Matrix Stacks

- CTM is actually not just 1 matrix but a matrix **STACK**
  - Multiple matrices in stack, “current” matrix at top
  - Can save transformation matrices for use later (push, pop)
- E.g: Traversing hierarchical data structures (Ch. 8)
- Pre 3.1 OpenGL also maintained matrix stacks
- Right now just implement 1-level CTM
- Matrix stack later for hierarchical transforms

Reading Back State

- Can also access OpenGL variables (and other parts of the state) by *query* functions

```
glGetIntegerv
glGetFloatv
glGetBooleanv
glGetDoublev
glIsEnabled
```

- Example: to find out maximum number of texture units

```
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &MaxTextureUnits);
```

Using Transformations

- **Example:** use idle function to rotate a cube and mouse function to change direction of rotation
- Start with program that draws cube as before
  - Centered at origin
  - Sides aligned with axes

Recall: 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();
}
```

Calls spinCube continuously
Whenever OpenGL program is idle

Recall: Idle and Mouse callbacks

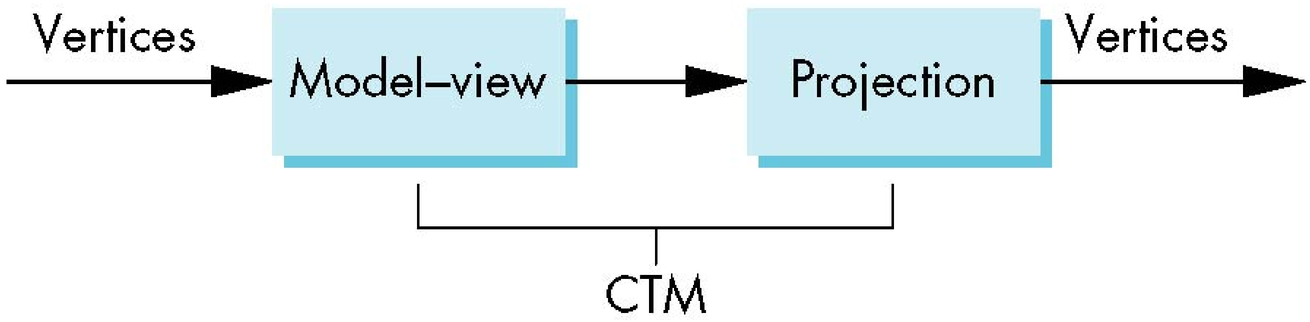
```
void spinCube()
{
   theta[axis] += 2.0;
   if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
   glutPostRedisplay();
}

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

Display callback

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   ctm = RotateX(theta[0])*RotateY(theta[1])
                                     *RotateZ(theta[2]);
    glUniformMatrix4fv(matrix_loc,1,GL_TRUE,ctm);
     glDrawArrays(GL_TRIANGLES, 0, N);
   glutSwapBuffers();
}
```

- Alternatively, we can
  - send rotation angle + axis to vertex shader,
  - Let shader form CTM then do rotation
- Inefficient: if mesh has 10,000 vertices each one forms CTM, redundant!!!!

Using the Model-view Matrix



- In OpenGL the model-view matrix used to
  - Transform 3D models (translate, scale, rotate)
  - Position camera (using LookAt function) (next)
- The projection matrix used to define view volume and select a camera lens (later)
- Although these matrices no longer part of OpenGL, good to create them in our applications (as CTM)

3D? Interfaces

- Major interactive graphics problem: how to use 2D devices (e.g. mouse) to control 3D objects
- Some alternatives
  - Virtual trackball
  - 3D input devices such as the spaceball
  - Use areas of the screen
    - Distance from center controls angle, position, scale depending on mouse button depressed

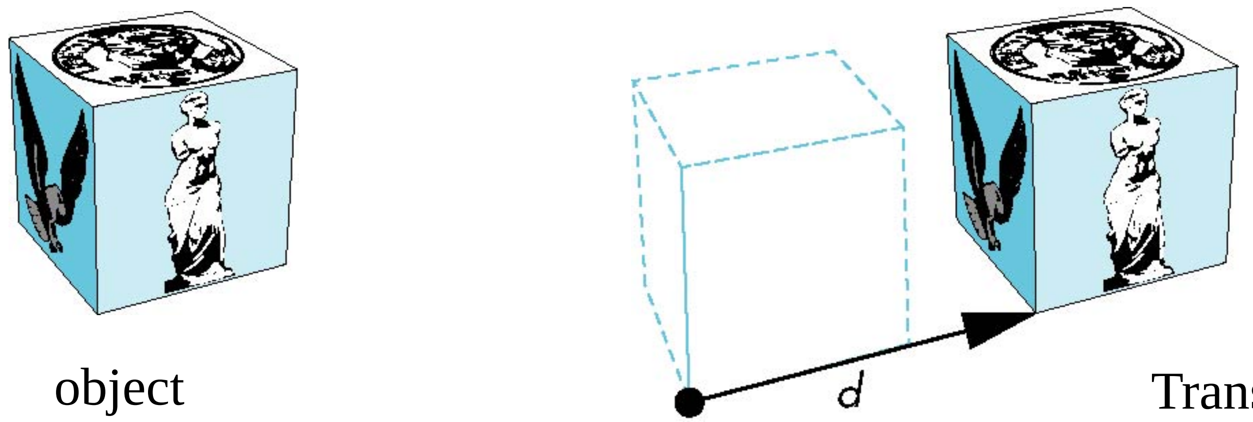
Computer Graphics 4731
Lecture 10: Rotations and Matrix Concatenation

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Recall: 3D Translate Example



- **Example:** If we translate a point (2,2,2) by displacement (2,4,6), new location of point is (4,6,8)

Translate(2,4,6)

- Translated x: 2 + 2 = 4
- Translated y: 2 + 4 = 6
- Translated z: 2 + 6 = 4

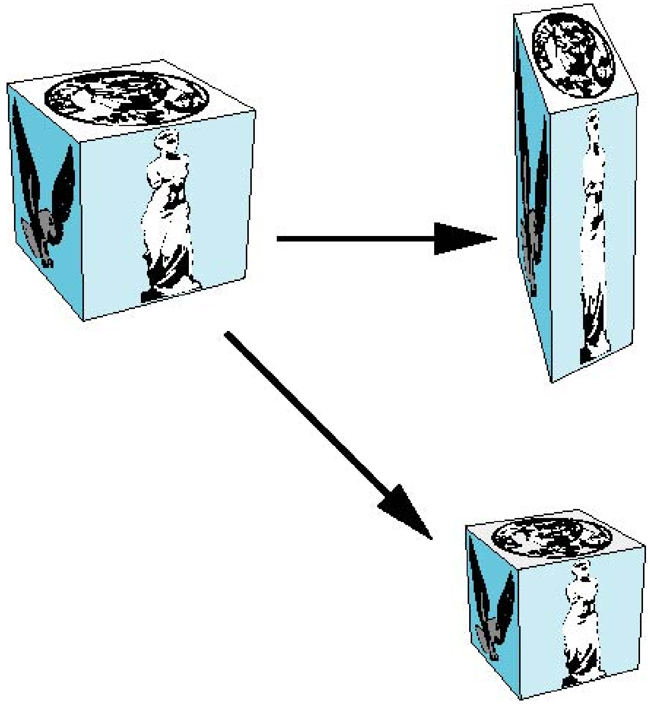
$$\begin{pmatrix} 4 \\ 6 \\ 8 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 2 \\ 2 \\ 1 \end{pmatrix}$$

Translated point — **Translation Matrix** — **Original point**

Recall: 3D Scale Example

If we scale a point (2,4,6) by scaling factor (0.5,0.5,0.5)
Scaled point position = (1, 2, 3)

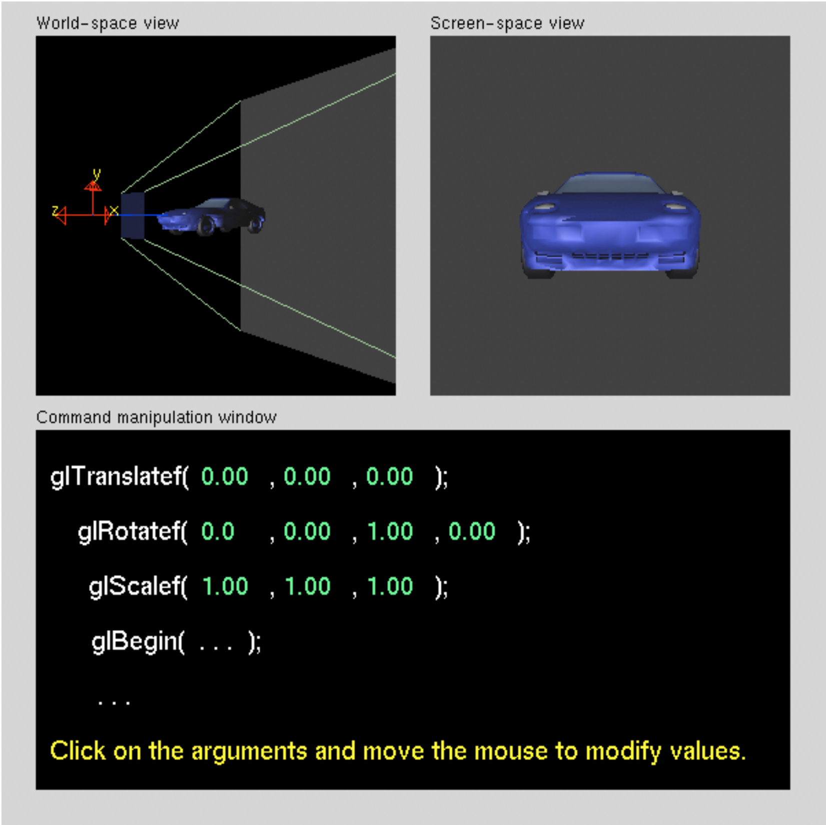
- Scaled x: 2 x 0.5 = 1
- Scaled y: 4 x 0.5 = 2
- Scaled z: 6 x 0.5 = 3

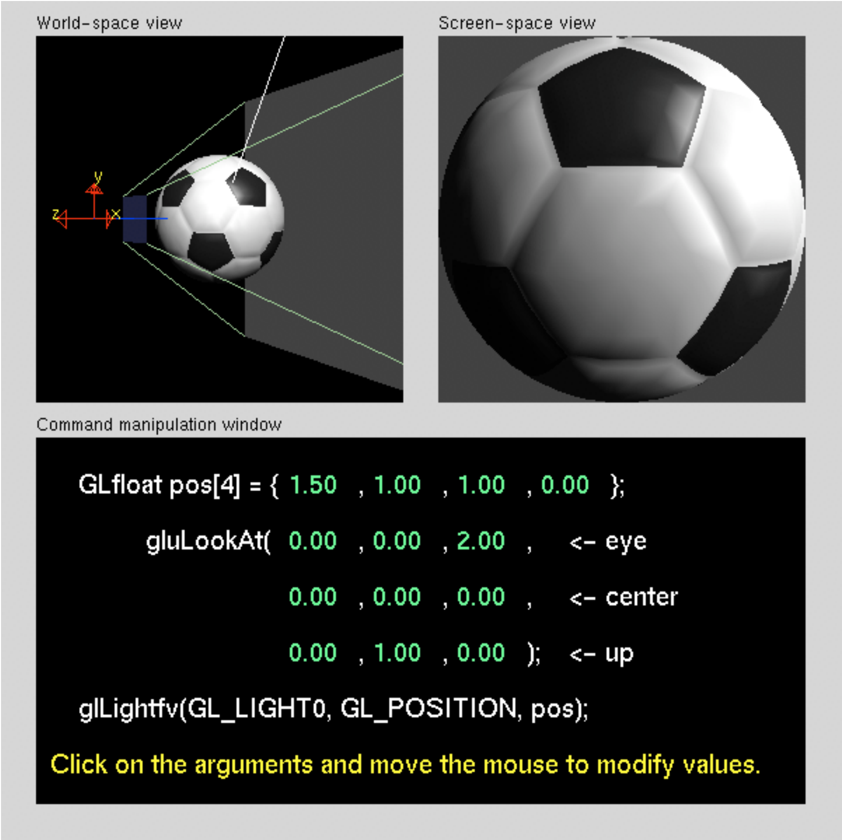


$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 \\ 4 \\ 6 \\ 1 \end{pmatrix}$$

Scaled point | **Scale Matrix for Scale(0.5, 0.5, 0.5)** | **Original point**

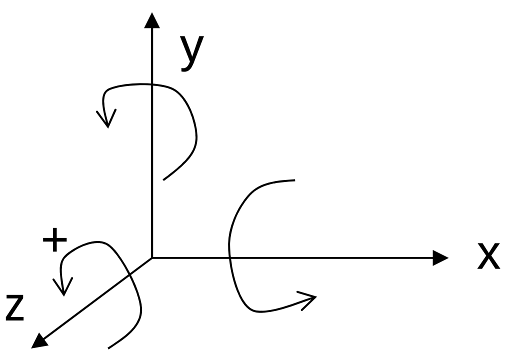
Nate Robbins Translate, Scale Rotate Demo





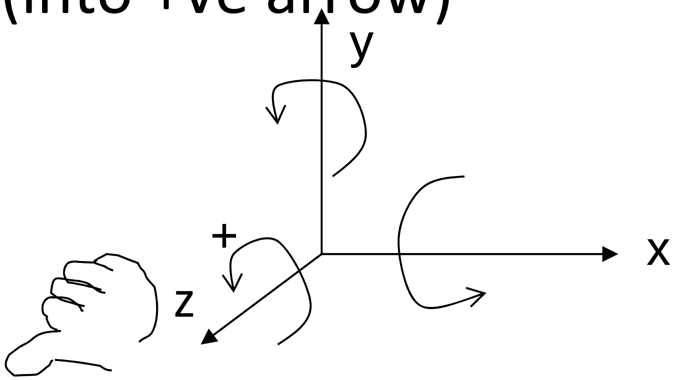
Rotating in 3D

- Many degrees of freedom. Rotate about what axis?
- 3D rotation: about a defined axis
- Different transform matrix for:
  - Rotation about x-axis
  - Rotation about y-axis
  - Rotation about z-axis

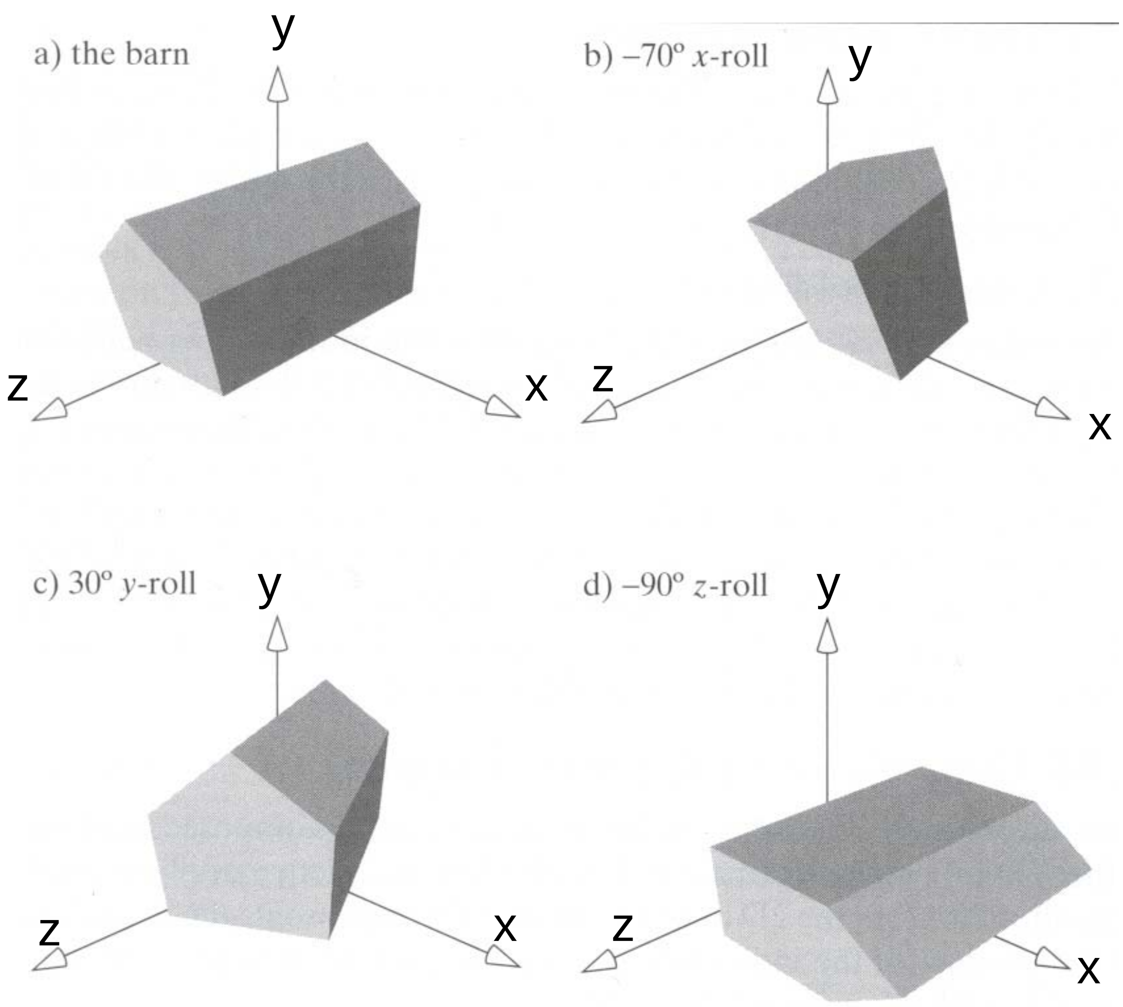


Rotating in 3D

- New terminology
  - **X-roll:** rotation about x-axis
  - **Y-roll:** rotation about y-axis
  - **Z-roll:** rotation about z-axis
- Which way is +ve rotation
  - Look in –ve direction (into +ve arrow)
  - CCW is +ve rotation



Rotating in 3D



Rotating in 3D

- For a rotation angle, β about an axis
- Define:

$$c = \cos(\beta) \qquad s = \sin(\beta)$$

x-roll or (RotateX)

$$R_x(\beta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotating in 3D

y-roll (or RotateY)

$$R_y(\beta) = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rules:

•Write 1 in rotation row, column

•Write 0 in the other rows/columns

•Write c,s in rect pattern

z-roll (or RotateZ)

$$R_z(\beta) = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example: Rotating in 3D

Question: Using **y-roll** equation, rotate P = (3,1,4) by 30 degrees:

Answer: c = cos(30) = 0.866, s = sin(30) = 0.5, and

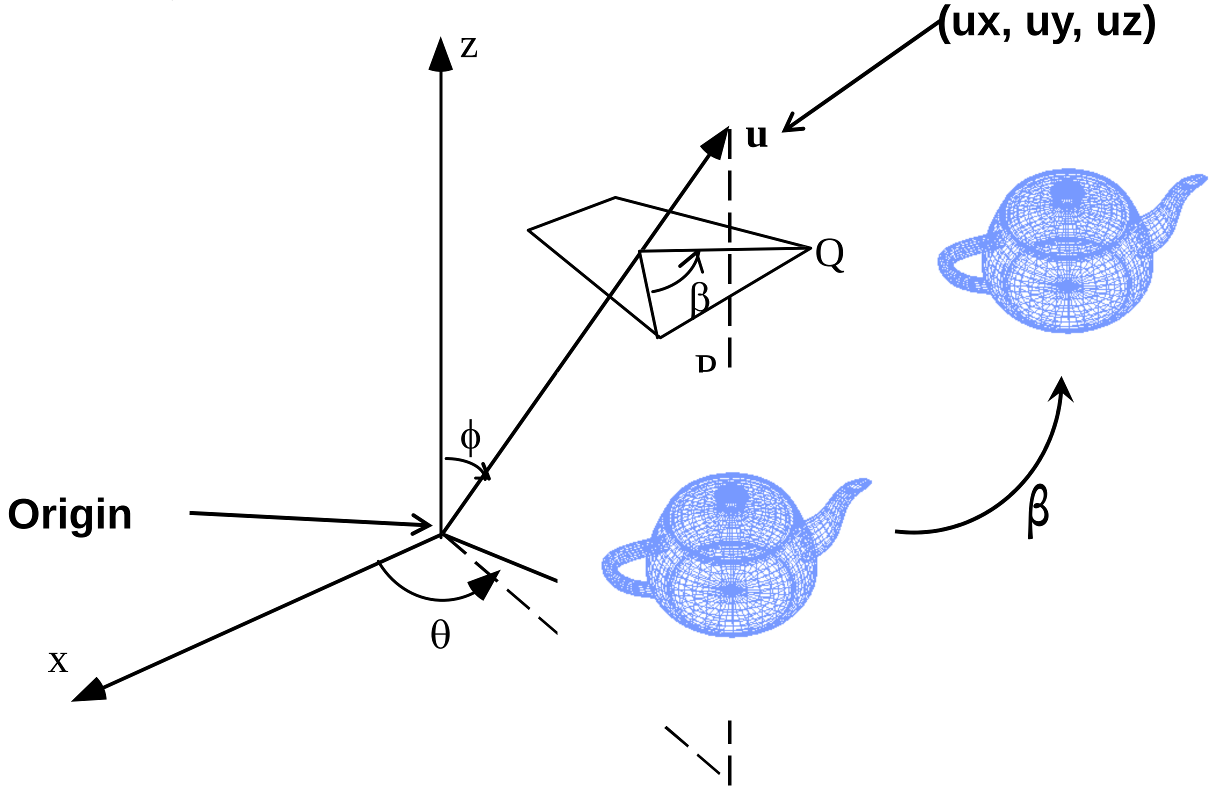
$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Line 1: 3.c + 1.0 + 4.s + 1.0

= 3 x 0.866 + 4 x 0.5 = 4.6

3D Rotation

- **Rotate(angle, ux, uy, uz):** rotate by angle β about an **arbitrary** axis (a vector) passing through **origin** and **(ux, uy, uz)**
- **Note:** Angular position of **u** specified as azimuth/longitude (θ) and latitude (ϕ)



Approach 1: 3D Rotation About Arbitrary Axis

- Can compose arbitrary rotation as combination of:
  - X-roll (by an angle β_1)
  - Y-roll (by an angle β_2)
  - Z-roll (by an angle β_3)

$$M = R_z(\beta_3)R_y(\beta_2)R_x(\beta_1)$$

←

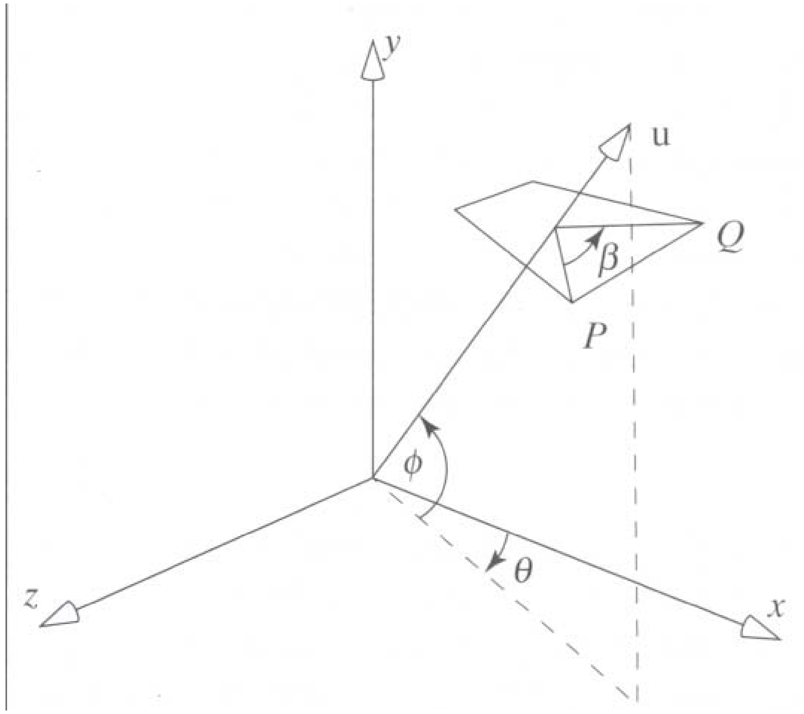
Read in reverse order

Approach 1: 3D Rotation using Euler Theorem

- **Classic:** use Euler's theorem
- **Euler's theorem:** any sequence of rotations = one rotation about some axis
- Want to rotate β about arbitrary axis **u** through origin
- Our approach:
  1. Use two rotations to align **u** and **x-axis**
  2. Do **x-roll** through angle β
  3. Negate two previous rotations to de-align **u** and **x-axis**

Approach 1: 3D Rotation using Euler Theorem

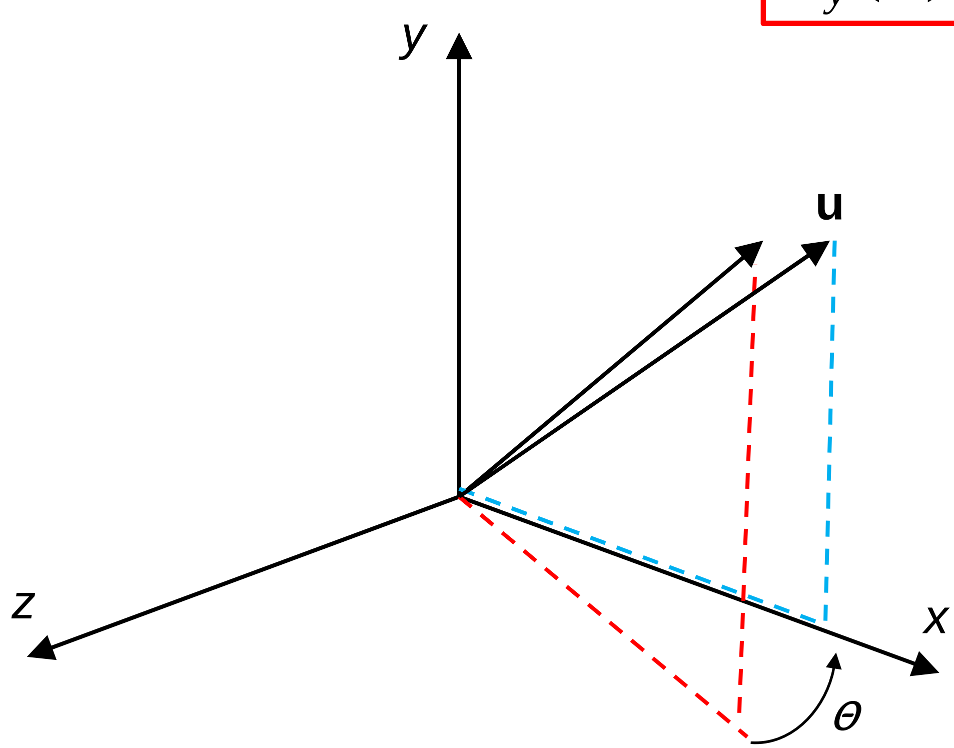
- **Note:** Angular position of **u** specified as azimuth (Θ) and latitude (ϕ)
- First try to align **u** with x axis



Approach 1: 3D Rotation using Euler Theorem

- **Step 1:** Do y-roll to line up rotation axis with x-y plane

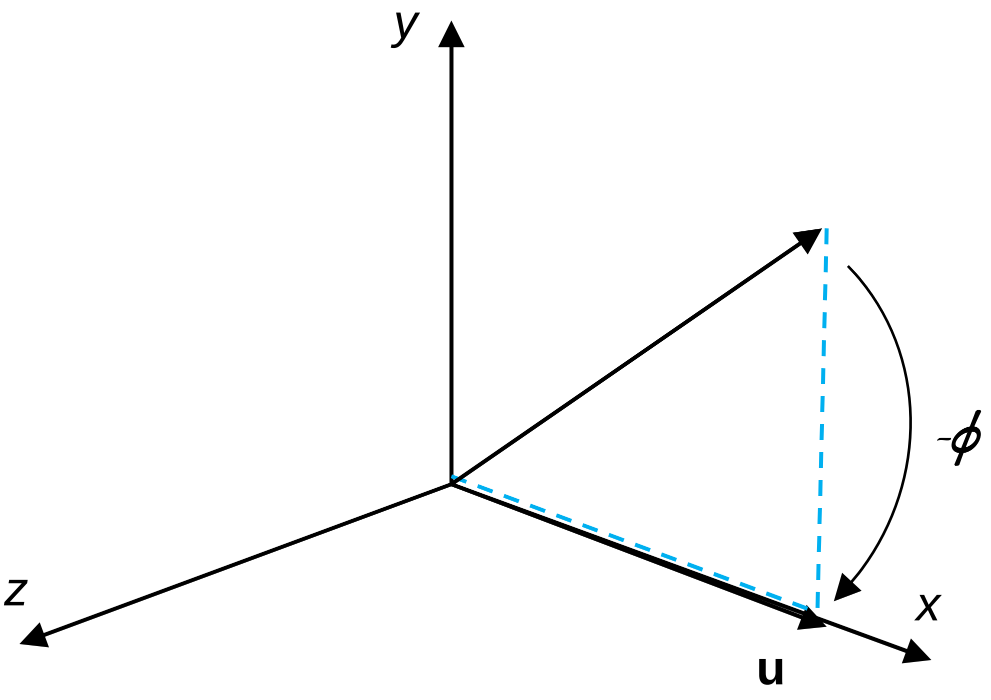
$$R_y(\theta)$$



Approach 1: 3D Rotation using Euler Theorem

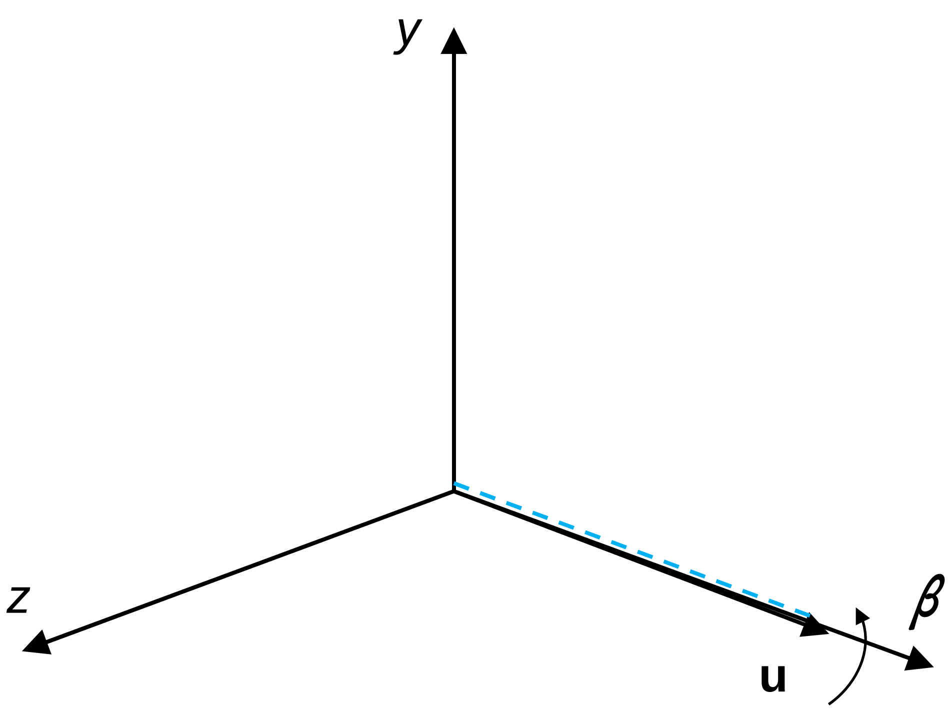
- **Step 2:** Do z-roll to line up rotation axis with x axis

$$R_z(-\phi)R_y(\theta)$$

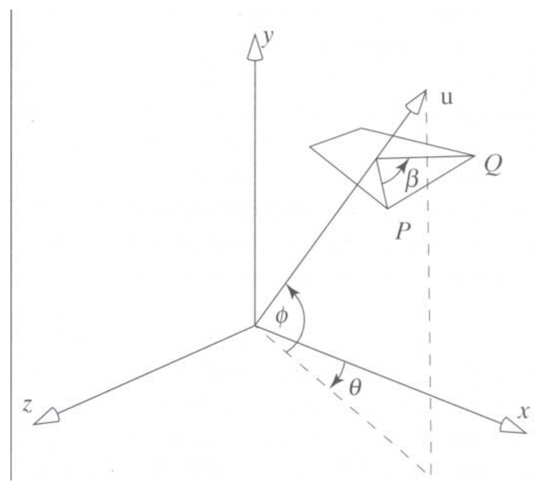


Approach 1: 3D Rotation using Euler Theorem

- **Remember:** Our goal is to do rotation by β around **u**
- But axis **u** is now lined up with x axis. So,
- **Step 3:** Do x-roll by β around axis **u**



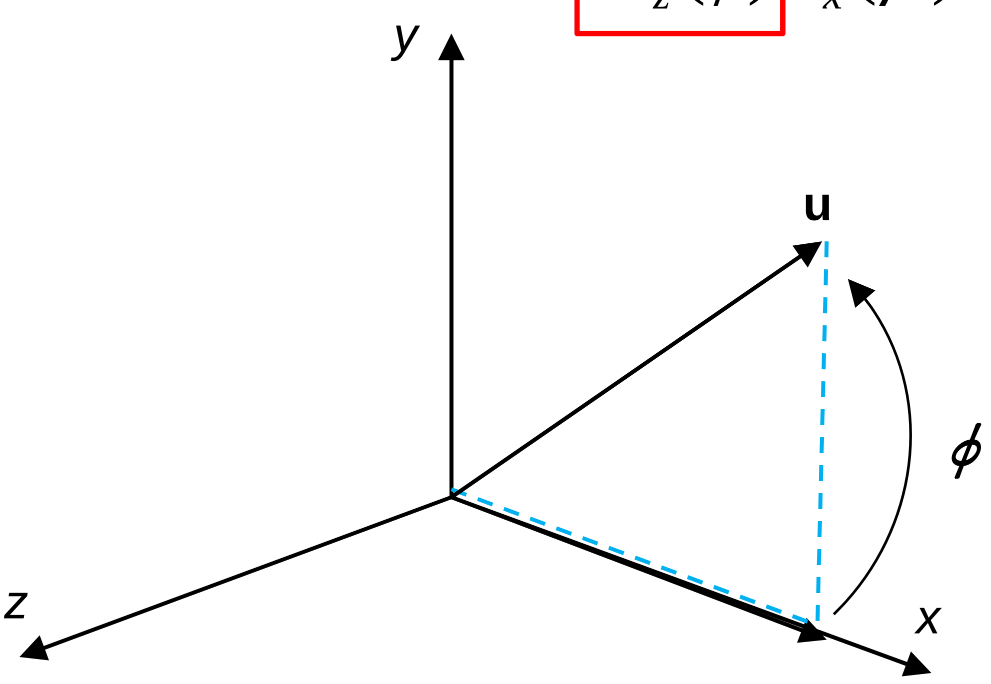
$$R_x(\beta) R_z(-\phi) R_y(\theta)$$



Approach 1: 3D Rotation using Euler Theorem

- Next 2 steps are to return vector **u** to original position
- **Step 4:** Do z-roll in x-y plane

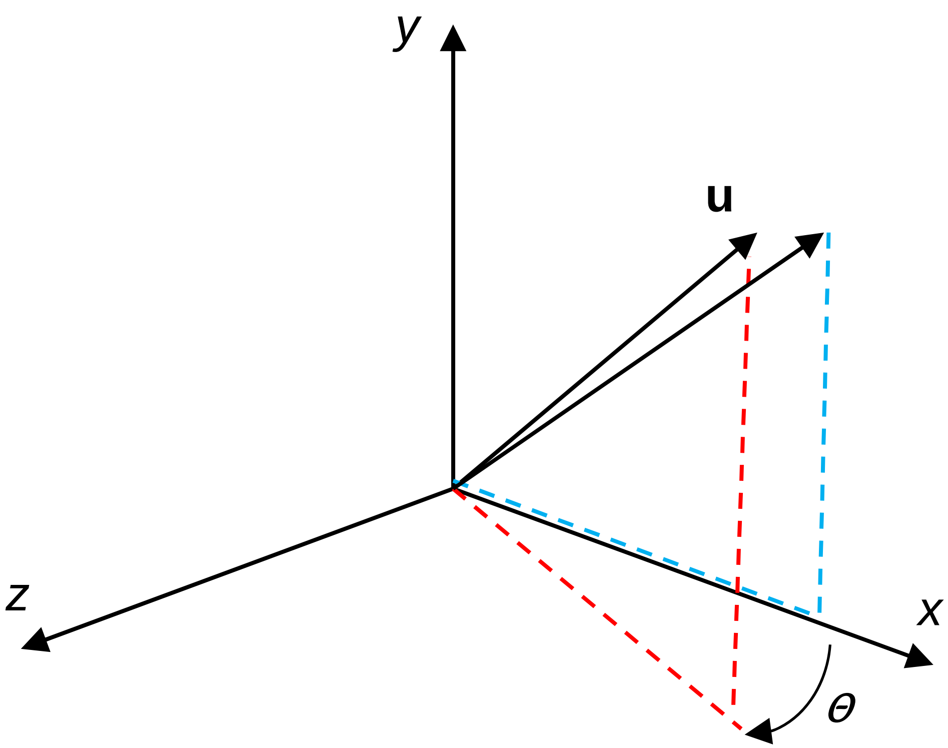
$$R_z(\phi) R_x(\beta) R_z(-\phi) R_y(\theta)$$

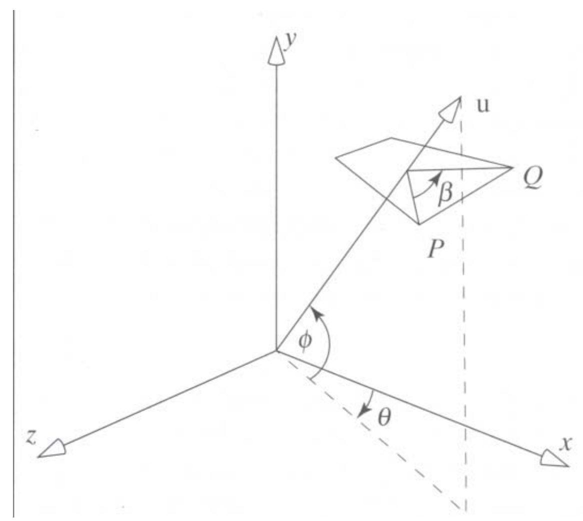


Approach 1: 3D Rotation using Euler Theorem

- **Step 5:** Do y-roll to return **u** to original position

$$R_u(\beta) = R_y(-\theta) R_z(\phi) R_x(\beta) R_z(-\phi) R_y(\theta)$$





Approach 2: Rotation using Quaternions

- Extension of imaginary numbers from 2 to 3 dimensions
- Requires 1 real and 3 imaginary components $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k}$$

- Quaternions can express rotations on sphere smoothly and efficiently

Approach 2: Rotation using Quaternions

- Derivation skipped! Check answer
- Solution has lots of symmetry

$$R(\beta)=\begin{pmatrix} c+(1-c)\mathbf{u}_x{}^2 & (1-c)\mathbf{u}_y\mathbf{u}_x+s\mathbf{u}_z & (1-c)\mathbf{u}_z\mathbf{u}_x+s\mathbf{u}_y & 0 \\ (1-c)\mathbf{u}_x\mathbf{u}_y+s\mathbf{u}_z & c+(1-c)\mathbf{u}_y{}^2 & (1-c)\mathbf{u}_z\mathbf{u}_y-s\mathbf{u}_x & 0 \\ (1-c)\mathbf{u}_x\mathbf{u}_z-s\mathbf{u}_y & (1-c)\mathbf{u}_y\mathbf{u}_z-s\mathbf{u}_x & c+(1-c)\mathbf{u}_z{}^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$c=\cos(\beta)$ $s=\sin(\beta)$ Arbitrary axis **u**

Inverse Matrices

- Can compute inverse matrices by general formulas
- But some easy **inverse transform** observations
  - Translation: $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
  - Scaling: $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$
  - Rotation: $\mathbf{R}^{-1}(q) = \mathbf{R}(-q)$
    - Holds for any rotation matrix

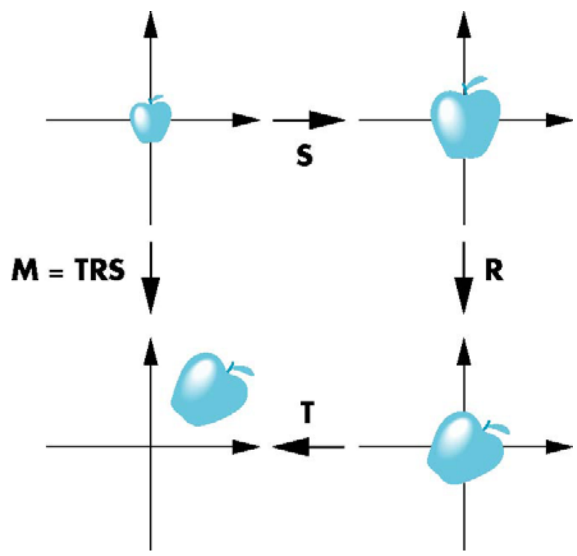
Instancing

- During modeling, often start with simple object centered at origin, aligned with axis, and unit size
- Can declare one copy of each shape in scene
- E.g. declare 1 mesh for soldier, 500 instances to create army
- Then apply *instance transformation* to its vertices to

    Scale

    Orient

    Locate



References

- Angel and Shreiner, Chapter 3
- Hill and Kelley, Computer Graphics Using OpenGL, 3rd edition