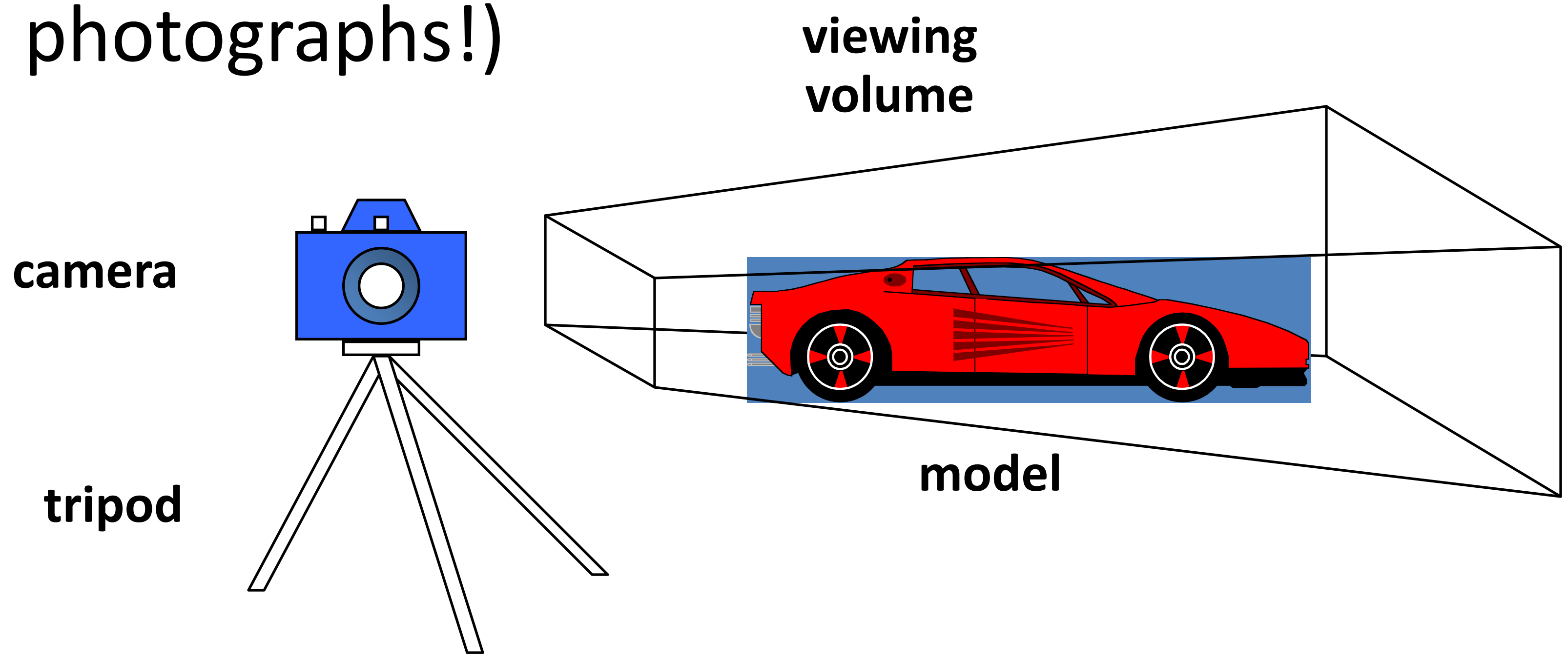
Transformations

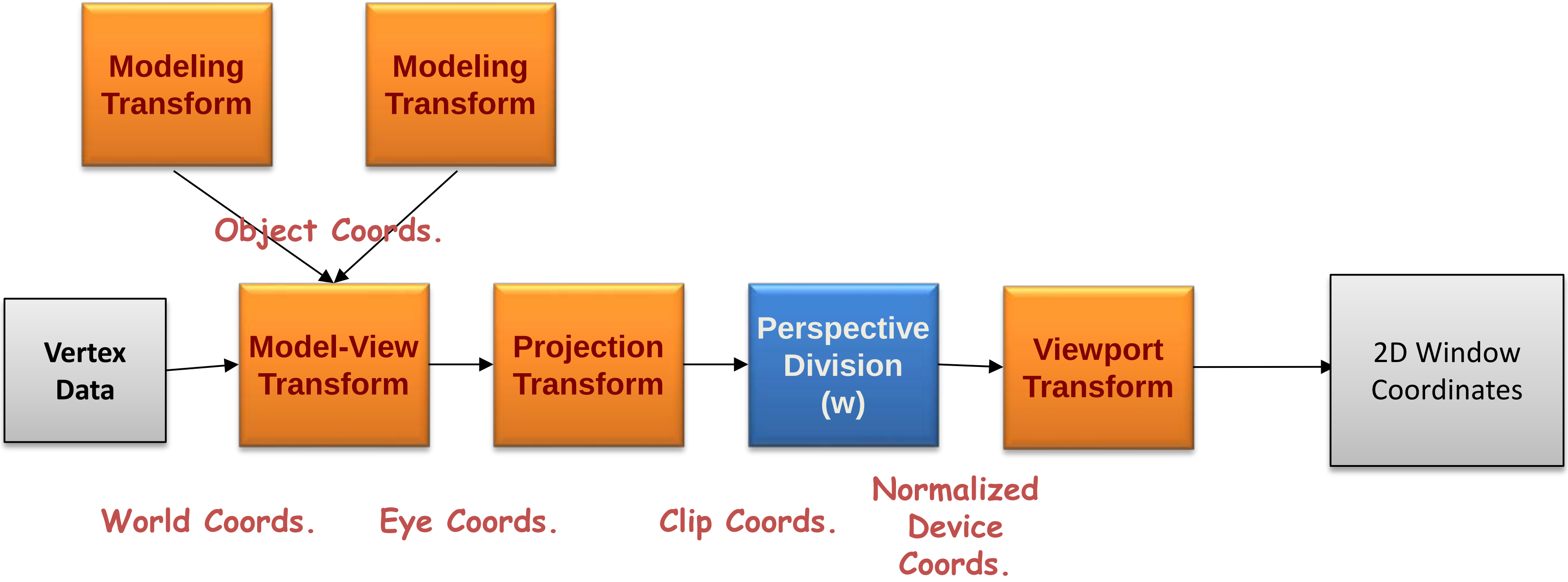
Camera Analogy

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



Transformations

- Transformations take us from one "space" to another
  - All of our transforms are 4×4 matrices



Camera Analogy and 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

3D Transformations

- A vertex is transformed by 4×4 matrices
  - all affine operations are matrix multiplications
- All matrices are stored column-major in OpenGL
  - this is opposite of what "C" programmers expect
- 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 What You Can See

- Set up a viewing frustum to specify how much of the world we can see
- Done in two steps
  - specify the size of the frustum (projection transform)
  - specify its location in space (model-view transform)
- Anything outside of the viewing frustum is clipped
  - primitive is either modified or discarded (if entirely outside frustum)

Specifying What You Can See (cont'd)

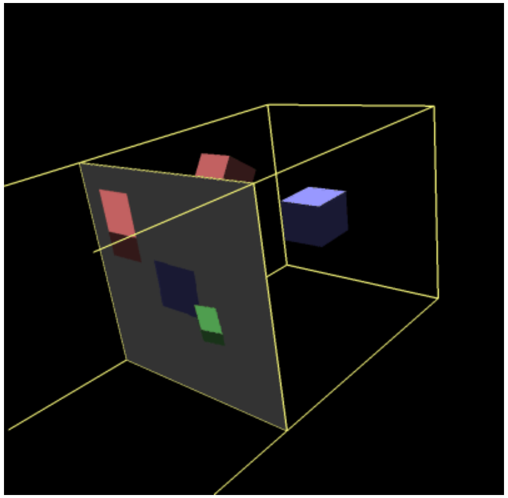
- OpenGL projection model uses eye coordinates
  - the "eye" is located at the origin
  - looking down the -z axis
- Projection matrices use a six-plane model:
  - near (image) plane and far (infinite) plane
    - both are distances from the eye (positive values)
  - enclosing planes
    - top & bottom, left & right

Orthographic vs Perspective Projection

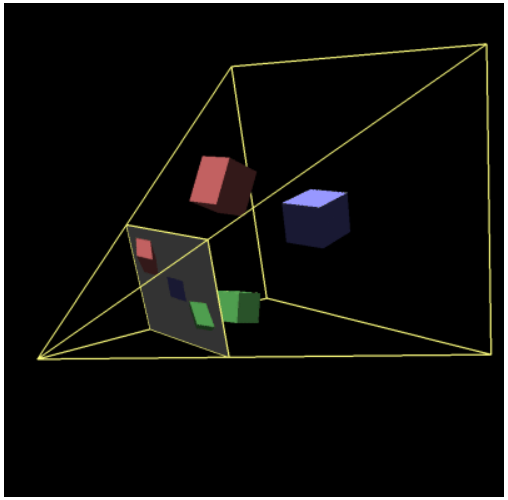
- Orthographic Projection
  - Parallel projection
  - Preserve size
    - Good for determining relative size
- Perspective Projection
  - Projection along rays
  - Closer objects appears larger
  - Human vision!
- Only work with: **Perspective Projection**

Specifying What You Can See (cont'd)

Orthographic View



Perspective View



$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

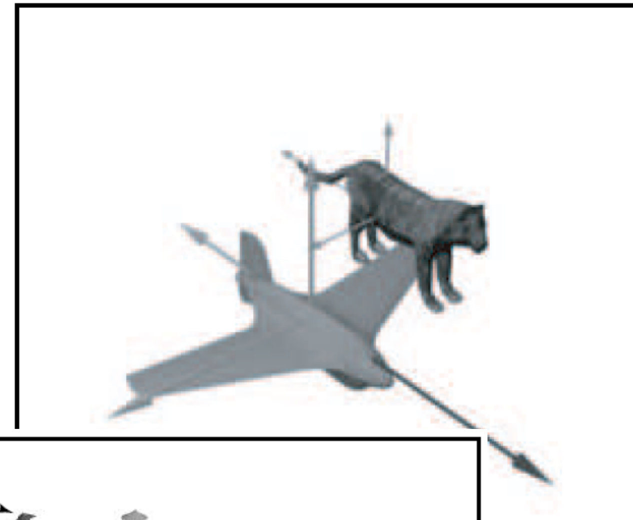
$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

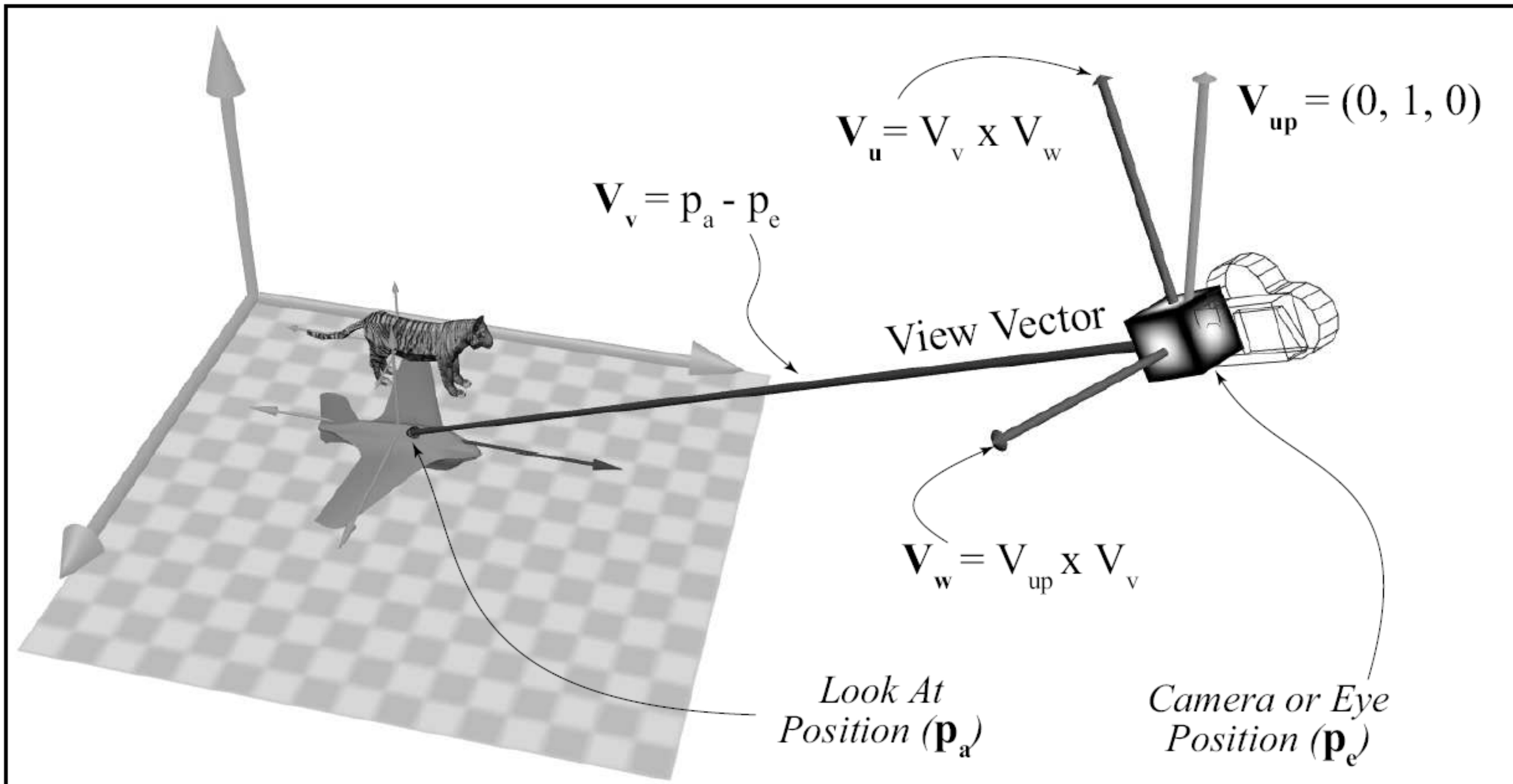
Coordonate Transformation Pipeline

- Recall: $\mathbf{V}_o = \mathbf{V}_i \; \mathbf{M}_W \; \mathbf{M}_V \; \mathbf{M}_P$
- Transforms $\overbrace{OC(\mathbf{V}_i) \xrightarrow{\mathbf{M}_W} WC}^{World} \; \overbrace{\xrightarrow{\mathbf{M}_V} EC}^{Eye} \; \overbrace{\xrightarrow{\mathbf{M}_P} NDC(\mathbf{V}_o)}^{Projection}$
  - World Transform (M_W)
    - Object Space (OC) To World Space (WC)
  - View Transform (M_V)
    - WC to Eye (Camera) Space (EC)
  - Projection Transform (M_P)
    - EC To NDC (Normalize Device)

$$\mathbf{V}_o = \begin{bmatrix} x_o & y_o & z_o \end{bmatrix} \; where \; \begin{cases} -1 \le x_o \le +1 \\ -1 \le y_o \le +1 \\ -1 \le z_o \le +1 \end{cases}$$

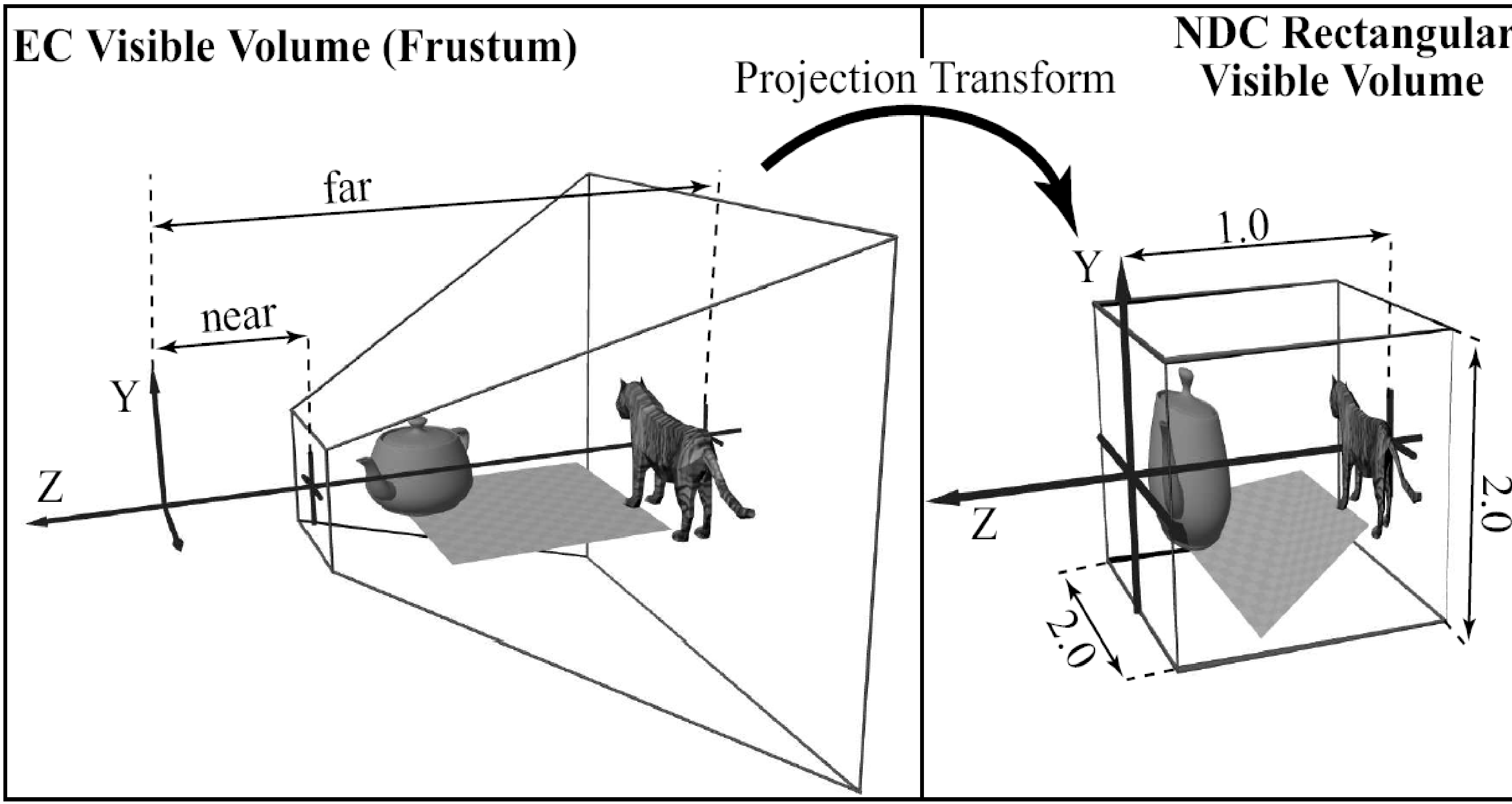
Example





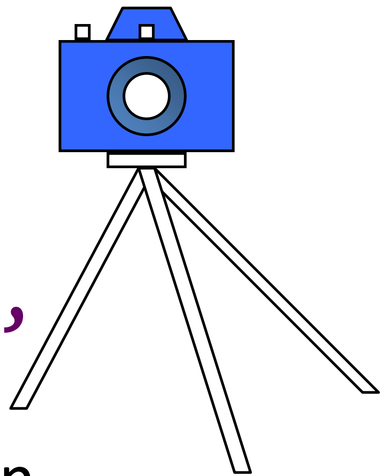
$V_u = V_v \times V_w$
$V_{up} = (0, 1, 0)$
$V_v = p_a - p_e$
View Vector
$V_w = V_{up} \times V_v$
Look At Position (p_a)
Camera or Eye Position (p_e)

View Frustum to NDC Cube



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
- `LookAt( eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz )`
  - up vector determines unique orientation
  - careful of degenerate positions



Creating the LookAt Matrix

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|}$$

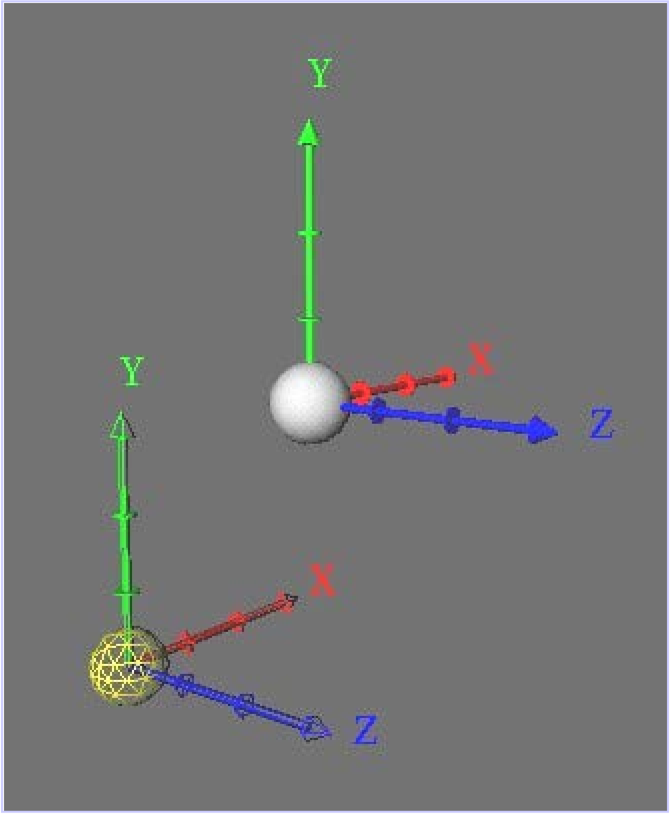
$$\hat{v} = \hat{u} \times \hat{n}$$

$$\begin{bmatrix} u_x & u_y & u_z & -(eye \cdot \vec{u}) \\ v_x & v_y & v_z & -(eye \cdot \vec{v}) \\ -n_x & -n_y & -n_z & -(eye \cdot \vec{n}) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

- Move the origin to a new location

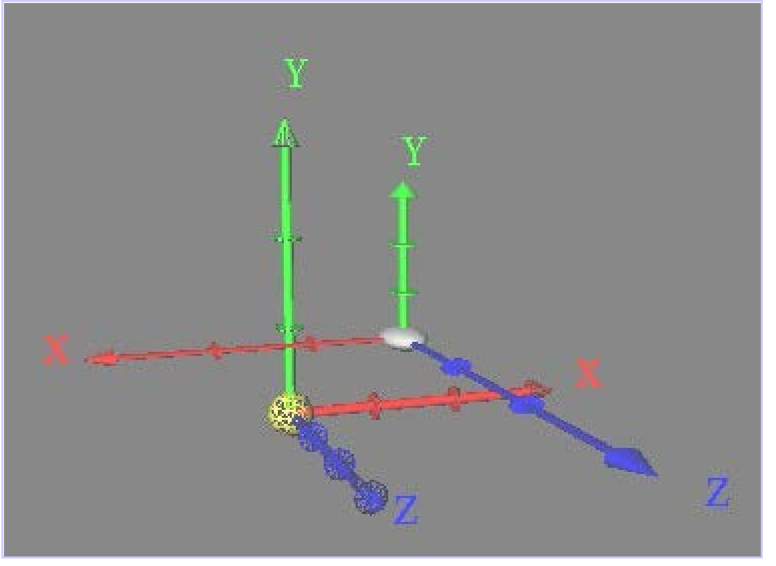
$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Scale

- Stretch, mirror or decimate a coordinate direction

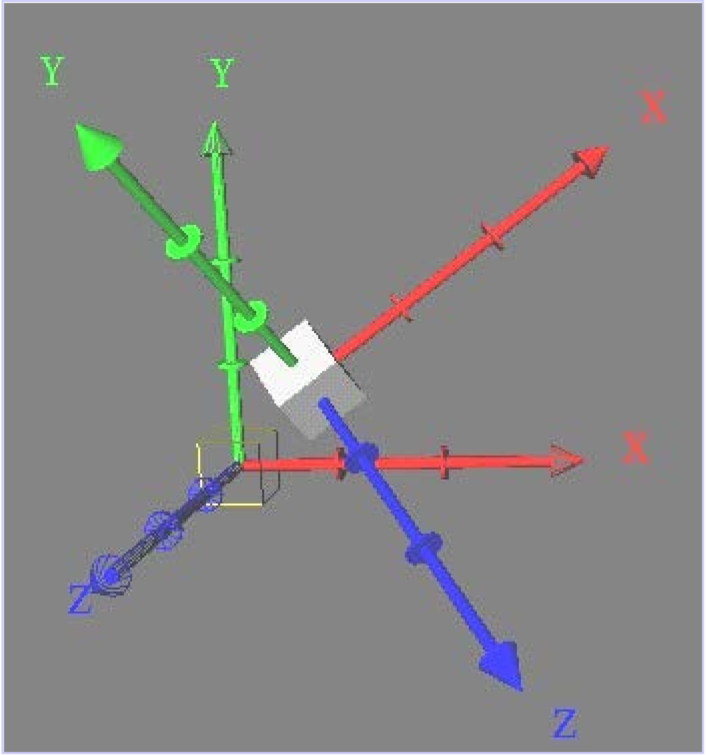
$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Note, there's a translation applied here to make things easier to see

Rotation

- Rotate coordinate system about an axis in space



Note, there's a translation applied here
to make things easier to see

Rotation (cont'd)

$$\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}$$

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = \begin{pmatrix} x' & y' & z' \end{pmatrix}$$

$$M = \vec{u}^t \vec{u} + \cos(\theta)(I - \vec{u}^t \vec{u}) + \sin(\theta) S$$

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_{\vec{v}}(\theta) = \begin{pmatrix} & M & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Vertex Shader for Rotation of Cube

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Vertex Shader for Rotation of Cube

(cont'd)

```
// Remember: these matrices are column-major

mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );

mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Vertex Shader for Rotation of Cube

(cont'd)

```
    mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                    s.z,  c.z, 0.0, 0.0,
                    0.0,  0.0, 1.0, 0.0,
                    0.0,  0.0, 0.0, 1.0 );

    color = vColor;
    gl_Position = rz * ry * rx *
vPosition;
}
```

Sending Angles from Application

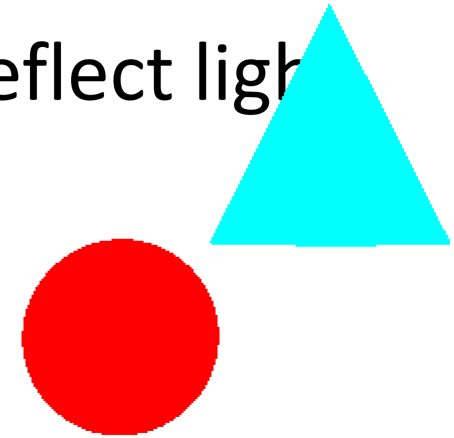
- Here, we compute our angles (`Theta`) in our mouse callback

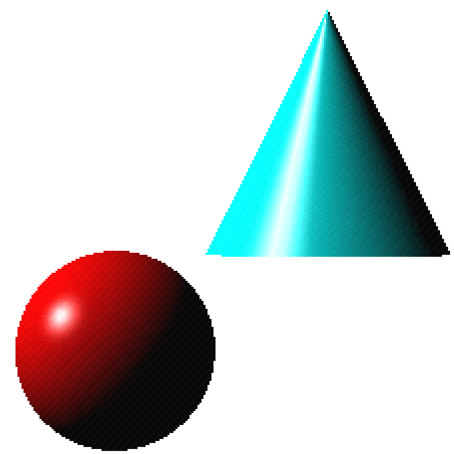
```
GLuint theta;  // theta uniform location
vec3  Theta;   // Axis angles

void display( void )
{
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

   glUniform3fv( theta, 1, Theta );
   glDrawArrays( GL_TRIANGLES, 0, NumVertices );

   glutSwapBuffers();
}
```

Lighting

Lighting Principles

- Lighting simulates how objects reflect ligh
  - material composition of object
  - light's color and position
  - global lighting parameters
- Usually implemented in
  - vertex shader for faster speed
  - fragment shader for nicer shading



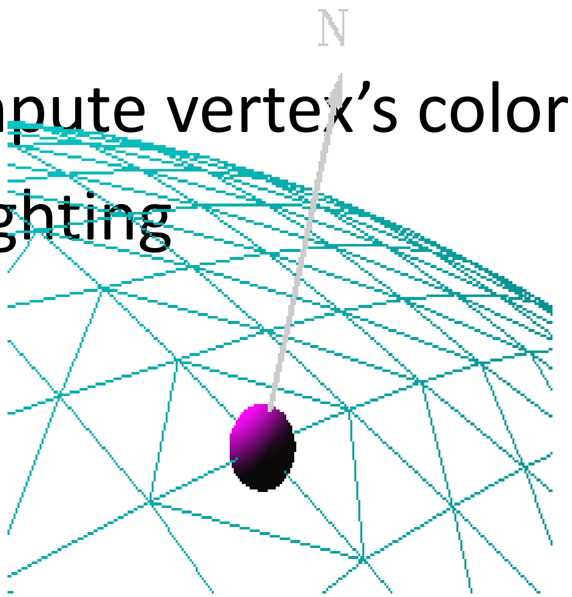


Modified Phong Model

- Computes a color for each vertex using
  - Surface normals
  - Diffuse and specular reflections
  - Viewer’s position and viewing direction
  - Ambient light
  - Emission
- Vertex colors are interpolated across polygons by the rasterizer
  - *Phong shading* does the same computation per pixel, interpolating the normal across the polygon
    - more accurate results

Surface Normals

- Normals define how a surface reflects light
  - Application usually provides normals as a vertex atttribute
  - Current normal is used to compute vertex's color
  - Use unit normals for proper lighting
    - scaling affects a normal's length



Material Properties

- Defi

Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

  – you can have separate materials for front and back

Adding Lighting to Cube

```
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4
    AmbientProduct, DiffuseProduct,
SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Adding Lighting to Cube (cont'd)

```
void main()
{
   // Transform vertex  position into eye
coordinates
   vec3 pos = vec3(ModelView * vPosition);

   vec3 L = normalize(LightPosition.xyz - pos);
   vec3 E = normalize(-pos);
   vec3 H = normalize(L + E);

   // Transform vertex normal into eye coordinates
   vec3 N = normalize(vec3(ModelView * vNormal));
```

Adding Lighting to Cube (cont’d)

```
 // Compute terms in the illumination equation
vec4 ambient = AmbientProduct;

float Kd = max( dot(L, N), 0.0 );
vec4  diffuse = Kd*DiffuseProduct;

float Ks = pow( max(dot(N, H), 0.0), Shininess );
vec4  specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0)

gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
```