CSE 564: Visualization

Introduction to Computer Graphics

Klaus Mueller

Computer Science Department

Stony Brook University

Topics To Be Covered

Viewing Transformations

Texture mapping

Open GL programming

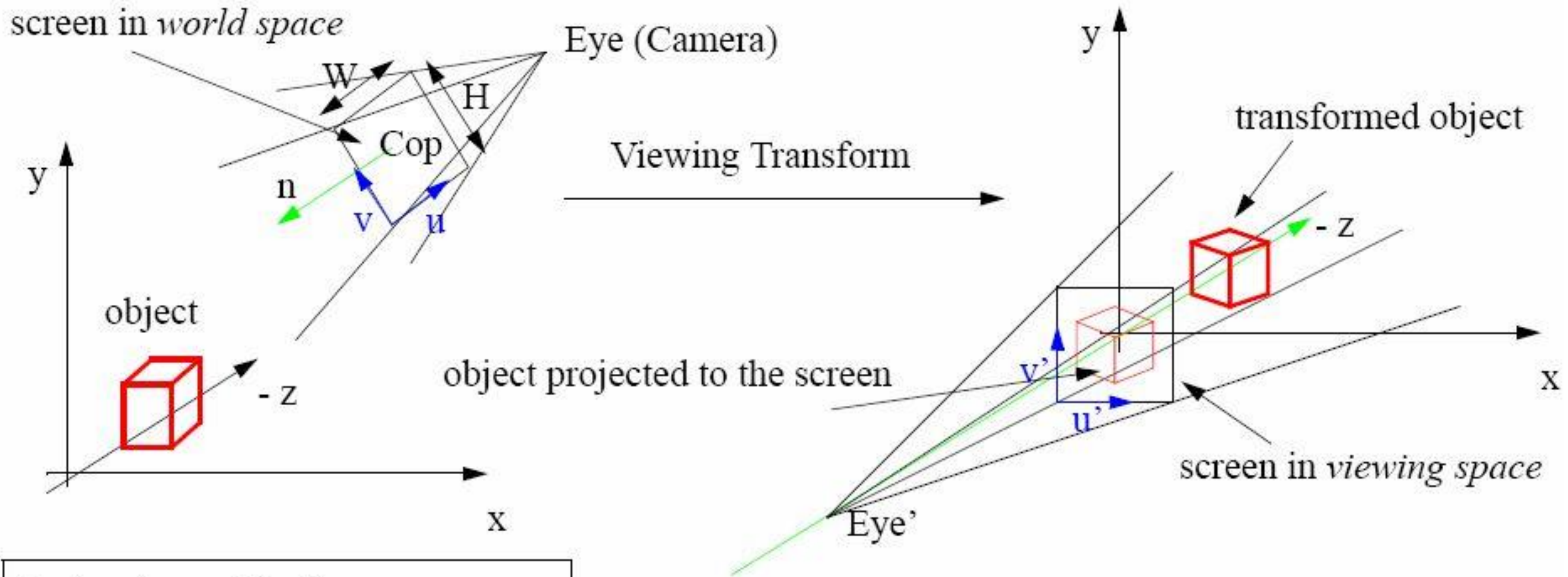
Shading and illumination

- discussion deferred to later

Polygonal graphics and the graphics pipeline

- presentation from Real-Time Volume Graphics book

Object-Order Viewing - Overview



A view is specified by:

- eye position (Eye)
- view direction vector (n)
- screen center position (Cop)
- screen orientation (u, v)
- screen width W, height H

u, v, n are orthonormal vectors

After the viewing transform:

- the screen center is at the coordinate system origin
- the screen is aligned with the x, y-axis
- the viewing vector points down the negative z-axis
- the eye is on the positive z-axis

All objects are transformed by the viewing transform

Step 1: Viewing Transform

- The sequence of transformations is:

  - *translate* the screen Center Of Projection (COP) to the coordinate system orgin (T_{view})

  - *rotate* the translated screen such that the view direction vector n points down the negative z-axis and the screen vectors u, v are aligned with the x, y-axis (R_{view})

- We get $M_{view} = R_{view} \cdot T_{view}$

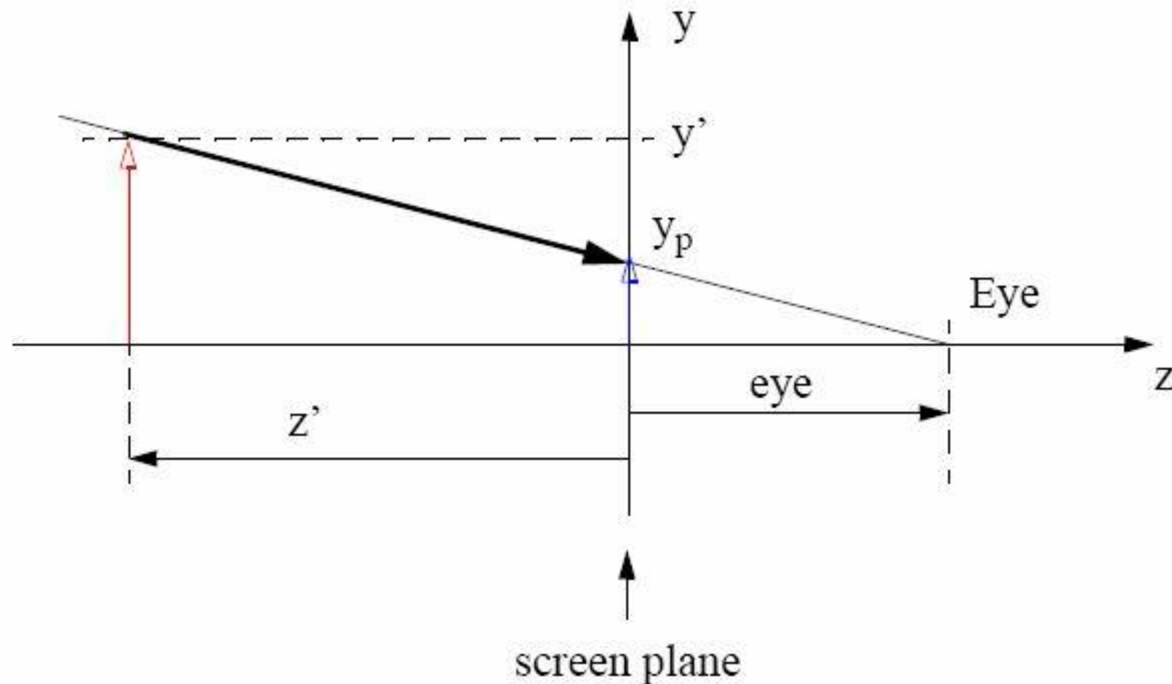
- We transform all object (points, vertices) by M_{view}:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -Cop_x \\ 0 & 1 & 0 & -Cop_y \\ 0 & 0 & 1 & -Cop_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Now the objects are easy to project since the screen is in a convenient position

  - but first we have to account for perspective distortion...

Step 2: Perspective Projection



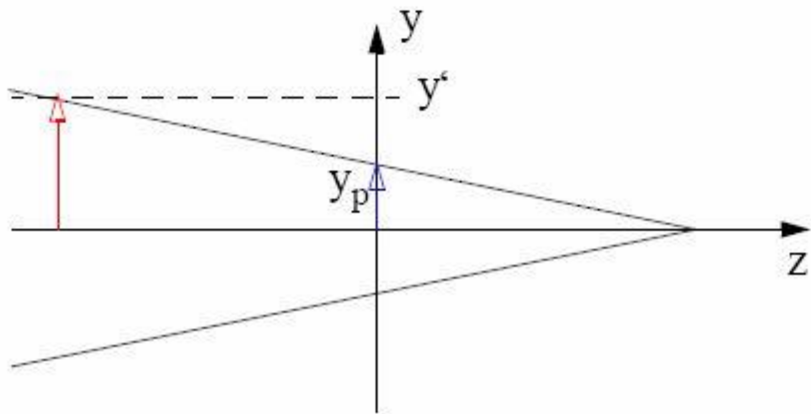
- A (view-transformed) vertex with coordinates (x', y', z') projects onto the screen as follows:

$$y_p = y' \cdot \frac{eye}{eye - z'} \qquad x_p = x' \cdot \frac{eye}{eye - z'}$$

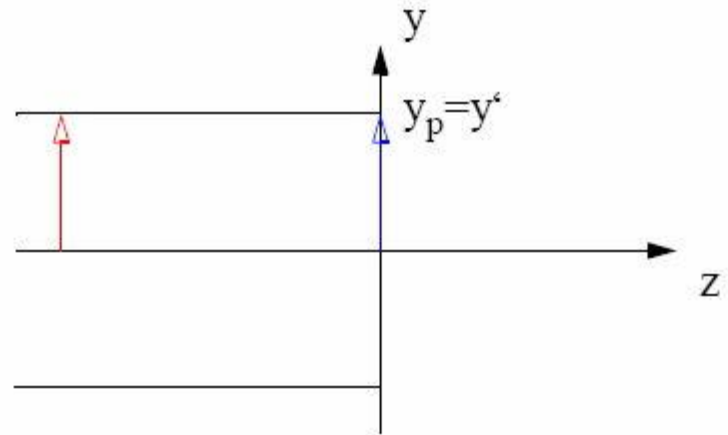
- x_p and y_p can be used to determine the screen coordinates of the object point (i.e., where to plot the point on the screen)

Or Use Simpler Parallel Projection

- Leave out the perspective mapping (step 2) in the viewing pipeline
- In orthographic projection, all object points project along parallel lines onto the screen



perspective projection



orthographic projection

Step 1 + Step 2 = World-To-Screen Transform

- The sequence of transformations is:
  - *translate* the screen Center Of Projection (COP) to the coordinate system orgin (T_{view})
  - *rotate* the translated screen such that the view direction vector n points down the negative z-axis and the screen vectors u, v are aligned with the x, y-axis (R_{view})
- We get $M_{view} = R_{view} \cdot T_{view}$
- We transform all object (points, vertices) by M_{view}:

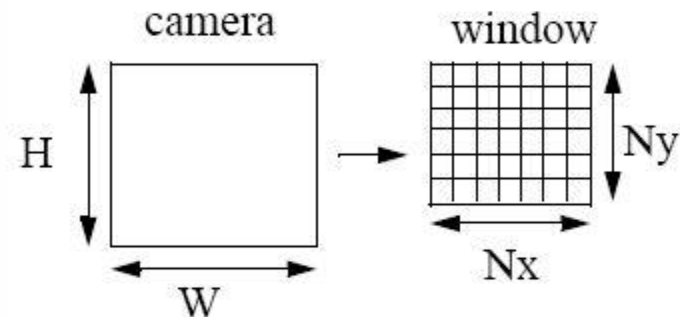
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -Cop_x \\ 0 & 1 & 0 & -Cop_y \\ 0 & 0 & 1 & -Cop_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Now the objects are easy to project since the screen is in a convenient position
  - but first we have to account for perspective distortion...

Step 3: Window Transform (1)

- Note: our camera screen is still described in world coordinates
- However, our display monitor is described on a pixel raster of size (Nx, Ny)
- The transformation of (perspective) viewing coordinates into pixel coordinates is called *window transform*
- Assume:
  - we want to display the rendered screen image in a window of size (Nx, Ny) pixels
  - the width and height of the camera screen in world coordinates is (W, H)
  - the center of the camera is at the center of the screen coordinate system
- Then:
  - the valid range of object coordinates is (-W/2 ... +W/2, -H/2 ... +H/2)
  - these have to be mapped into (0 ... Nx-1, 0 ... Ny-1):

$$x_s = \left(x_p + \frac{W}{2}\right) \cdot \frac{Nx-1}{W} \qquad y_s = \left(y_p + \frac{H}{2}\right) \cdot \frac{Ny-1}{H}$$



Step 3: Window Transform (2)

- The window transform can be written as the matrix M_{window}:

$$\begin{bmatrix} x_s \\ y_s \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{Nx-1}{W} & 0 & \frac{Nx-1}{2} \\ 0 & \frac{Ny-1}{H} & \frac{Ny-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}$$

- After the perspective divide, all object points (vertices) are multiplied by M_{window}
- Note: we could figure the window transform into M_{trans}
  - in that case, there is only one matrix multiply per object point (vertex) with a subsequent perspective divide
  - the OpenGL graphics pipeline does this

Rendering With OpenGl (1)

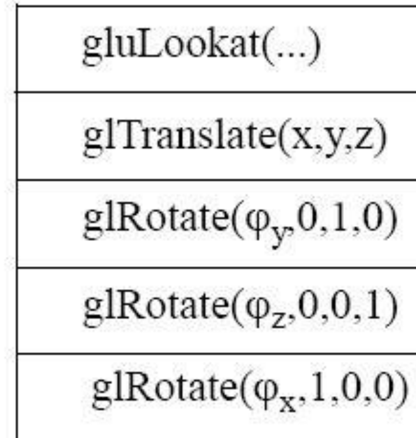
- glMatrixMode(GL_PROJECTION)
- Define the viewing window:

  glOrtho() for parallel projection

  glFrustum() for perspective projection
- glMatrixMode(GL_MODELVIEW)
- Specify the viewpoint

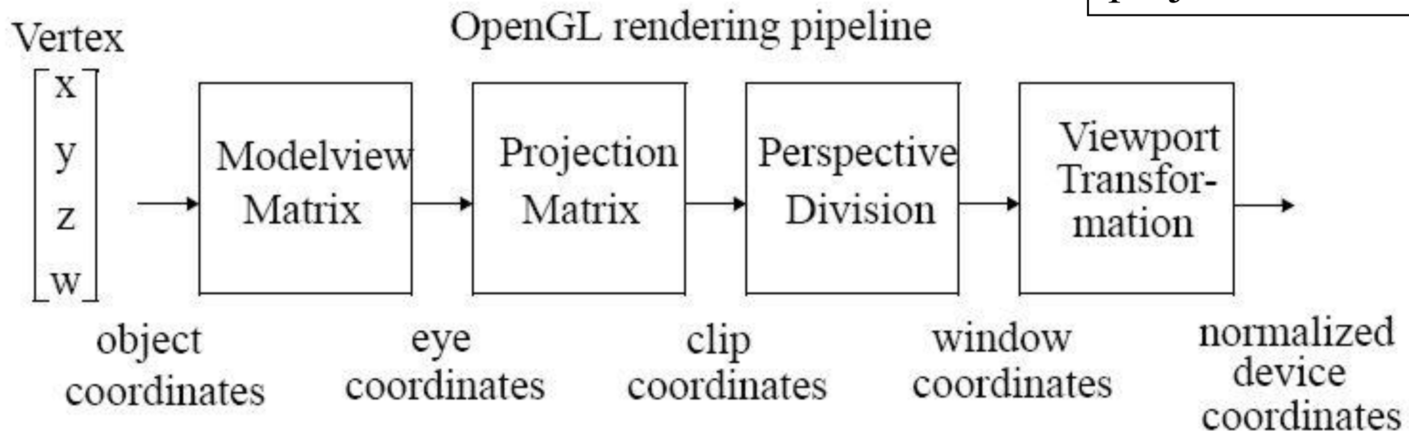
  gluLookat() /* need to have GLUT */
- Model the scene

  glTranslate(), glRotate(), glScale(), ...



rotate first, then translate, then do viewing...

use parallel projection for now



Rendering With OpenGL (2)

Specify the light sources: glLight()   Enable the z-buffer: glEnable(GL_DEPTH_TEST)

Enable lighting: glEnable(GL_LIGHTING)

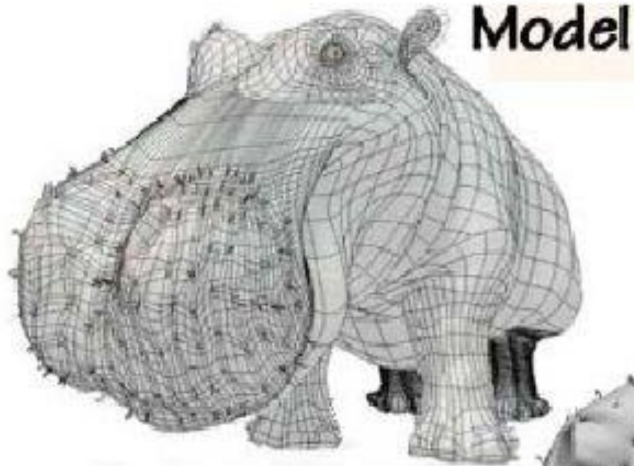
Enable light source *i*: glEnable(GL_LIGHT*i*) /* GL_LIGHT*i* is the symbolic name of light *i* */

Select shading model: glShadeModel() /* GL_FLAT or GL_SMOOTH */

We only need this for now

```
For each object:
/* duplicate the matrix on the stack if want to apply some extra transformations to the object */
    glPushMatrix();
    glTranslate(), glRotate(), glScale()   /* any specific transformation on this object */
    for all polygons of the object:   /* specify the polygon (assume a triangle here) */
        glBegin(GL_POLYGON);
            glColor3fv(c1);  glVertex3fv(v1);  glNormal3fv(n1);  /* vertex 1 */
            glColor3fv(c2);  glVertex3fv(v2);  glNormal3fv(n2);  /* vertex 2 */
            glColor3fv(c3);  glVertex3fv(v3);  glNormal3fv(n3);  /* vertex 3 */
        glEnd();
    glPopMatrix() /* get rid of the object-specific transformations, pop back the saved matrix */
```

Texture Mapping - Realistic Detail for Boring Polygons



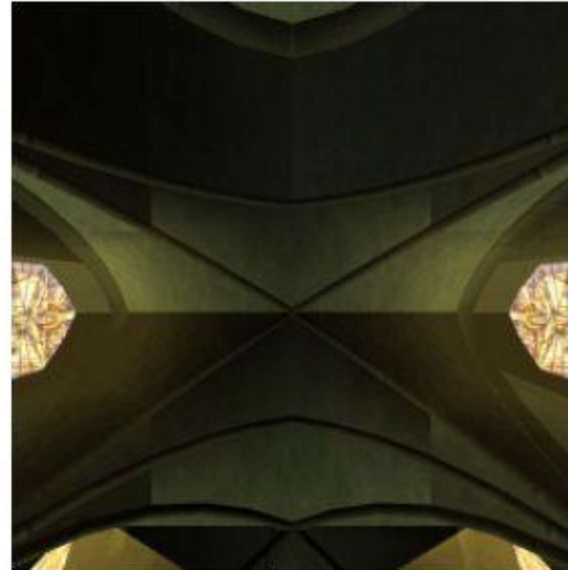
Model with Shading

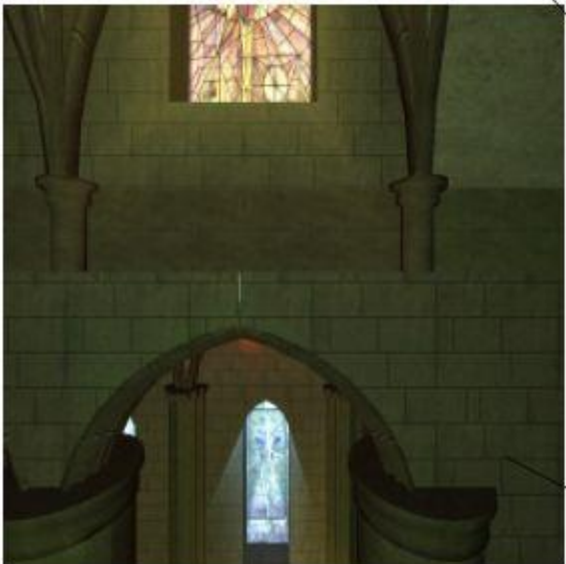
Model with Shading and Textures

At what point do things start looking realistic?

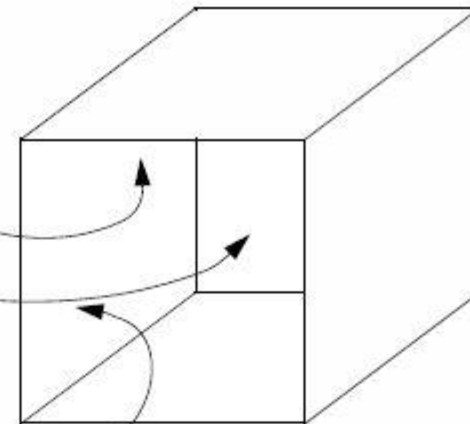
Leonard McMillan

Texture Mapping - Large Walls



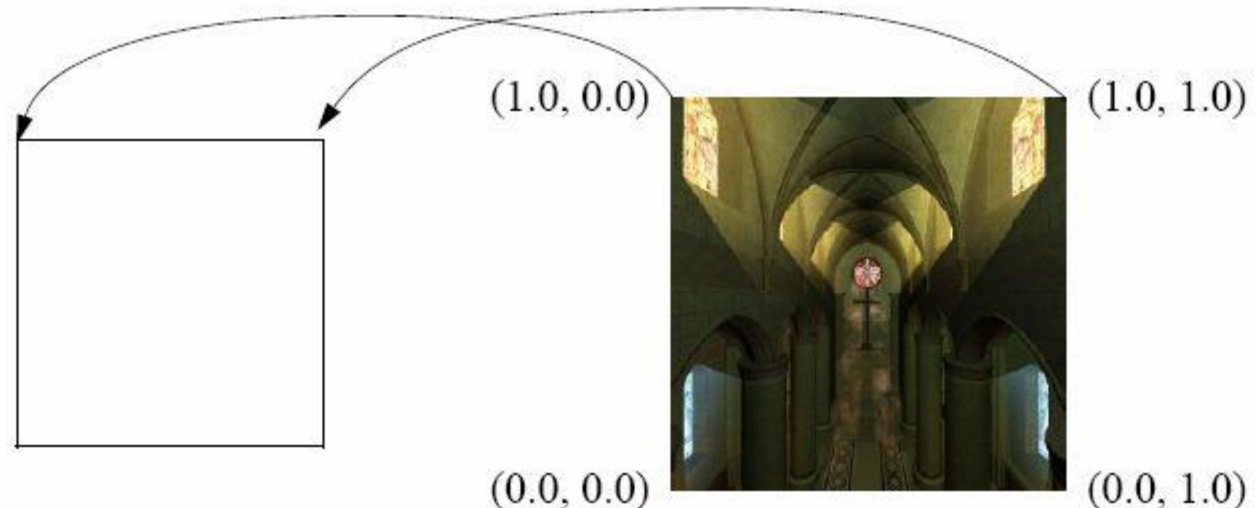


Take pictures, map as textures onto large polygon

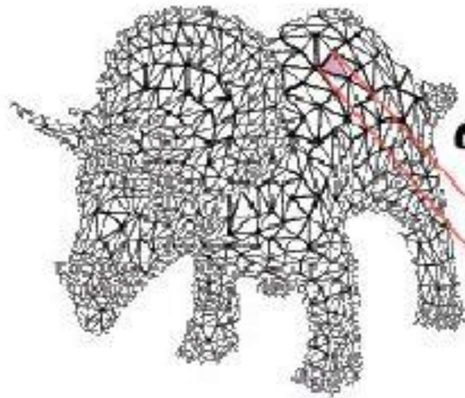


Texture Mapping Large Walls - OpenGL Program

```
glEnable(GL_TEXTURE_2D);
for each polygon
    glBindTexture(textureName);
    glBegin(GL_QUAD);
        glColor3fv(c1);  glVertex3fv(v1);   glTexCoord2D(0.0, 0.0);  /* vertex 1 */
        glColor3fv(c2);  glVertex3fv(v2);   glTexCoord2D(0.0, 1.0); /* vertex 2 */
        glColor3fv(c3);  glVertex3fv(v3);   glTexCoord2D(1.0, 1.0); /* vertex 3 */
        glColor3fv(c4);  glVertex3fv(v4);   glTexCoord2D(1.0, 0.0); /* vertex 4 */
    glEnd();
```



Texture Mapping - Small Facets

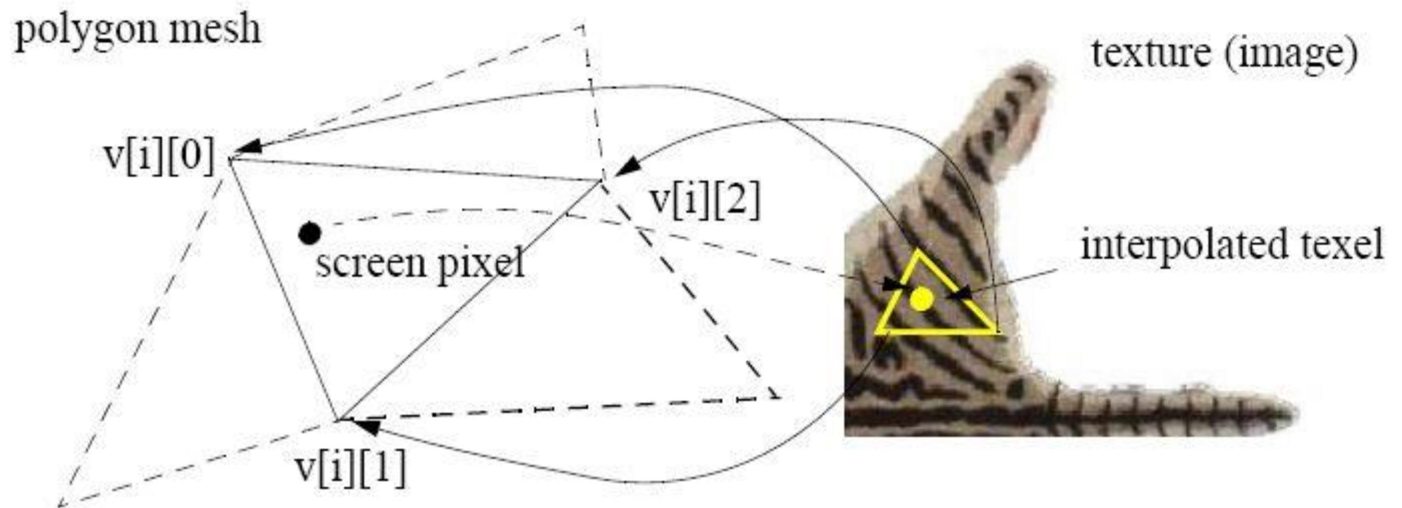


For each triangle in the model establish a corresponding region in a "texture map"

During rasterization interpolate the coordinate indices within the texture map

Texture Mapping Small Facets - OpenGL Program

```
glEnable(GL_TEXTURE_2D);
glBindTexture(textureName);
for each polygon i in the mesh
    glBegin(GL_QUAD);
        glColor3fv(c[i][0]);  glVertex3fv(v[i][0]);  glTexCoord2fv(t[i][0]);  /* vertex 1 */
        glColor3fv(c[i][1]);  glVertex3fv(v[i][1]);  glTexCoord2fv(t[i][1]); /* vertex 2 */
        glColor3fv(c[i][2]);  glVertex3fv(v[i][2]);  glTexCoord2fv(t[i][2]); /* vertex 3 */
    glEnd();
```



GPU-Based Image Processing

Textures are images, and images are textures

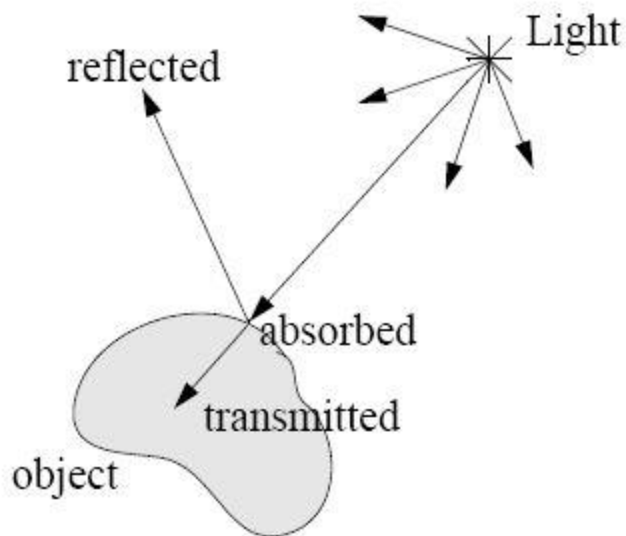
Can perform image processing by texture processing

Simply do this:

- map image onto a polygon
- render polygon at parallel projection
- run processing operation in a GPU fragment program (see next)

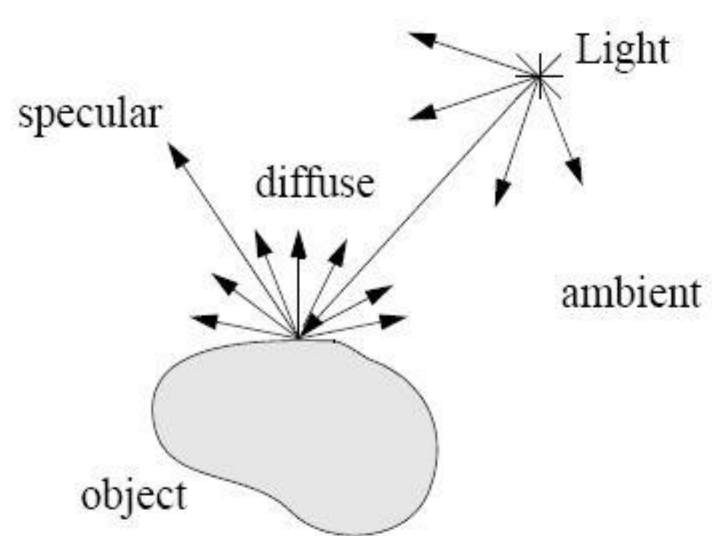
Illumination (Shading)

Total light decomposition



Light = reflected + transmitted + absorbed

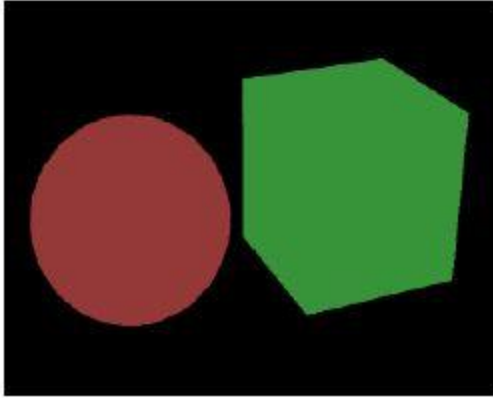
Reflected light



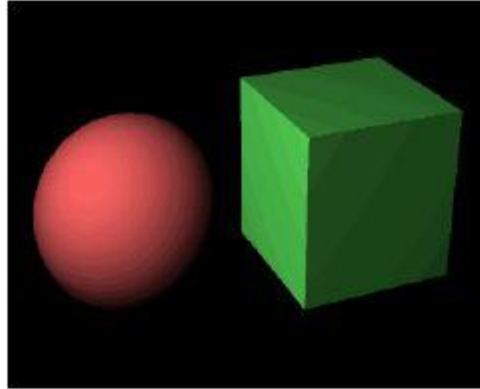
Reflected light = ambient + diffuse + specular

$$I = I_a + I_d + I_s$$

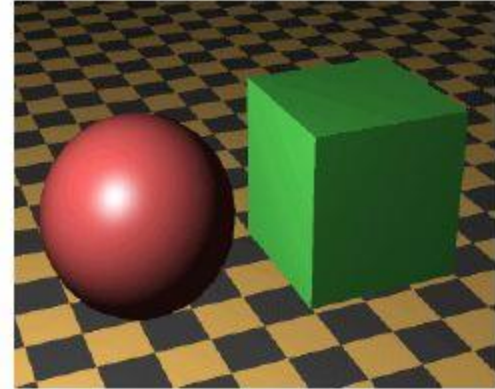
Illumination Examples



ambient



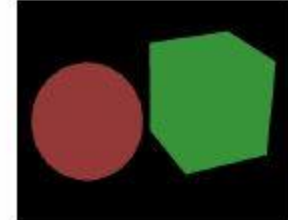
ambient + diffuse



ambient + diffuse + specular
(and a checkerboard)

Ambient Reflection

- Uniform background light
- $I_a = k_a I_A$
  - I_A: ambient light
  - k_a: material's ambient reflection coefficient
- Models general level of brightness in the scene
- Accounts for light effects that are difficult to compute (secondary diffuse reflections, etc)
- Constant for all surfaces of a particular object and the directions it is viewed at

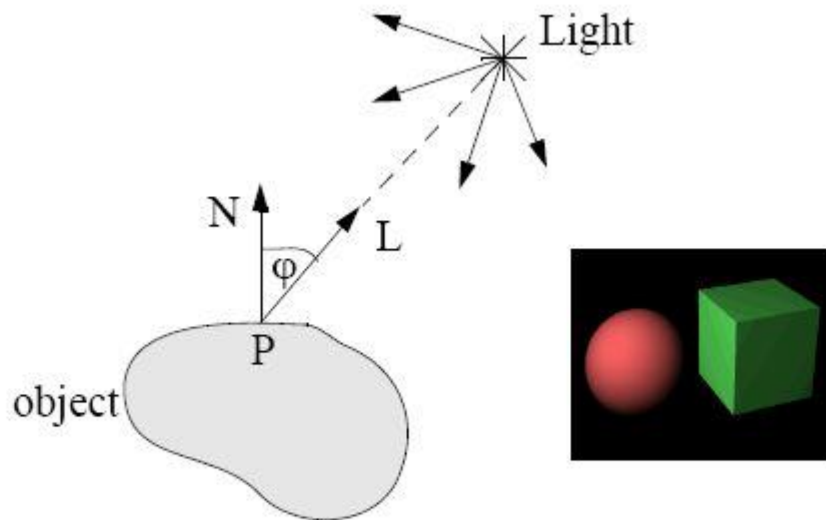


Diffuse Reflection

- Models dullness, roughness of a surface
- Equal light scattering in all directions
- For example, chalk is a diffuse reflector

Dot product:

$N \cdot L = (N_x L_x + N_y L_y + N_z L_z)$



$$L = \frac{Light - P}{|Light - P|} = \frac{(Light_x - P_x)}{|L'|}, \frac{(Light_y - P_y)}{|L'|}, \frac{(Light_z - P_z)}{|L'|}$$

$$|L'| = \sqrt{(Light_x - P_x)^2 + (Light_y - P_y)^2 + (Light_z - P_z)^2}$$

Lambertian cosine law:

$I_d = k_d \, I_L \cos \varphi = k_d \, I_L \, N \cdot L$

I_L: intensity of lightsource

N: surface normal vector

L: light vector (unit length)

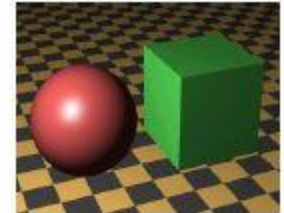
φ: angle of light incidence

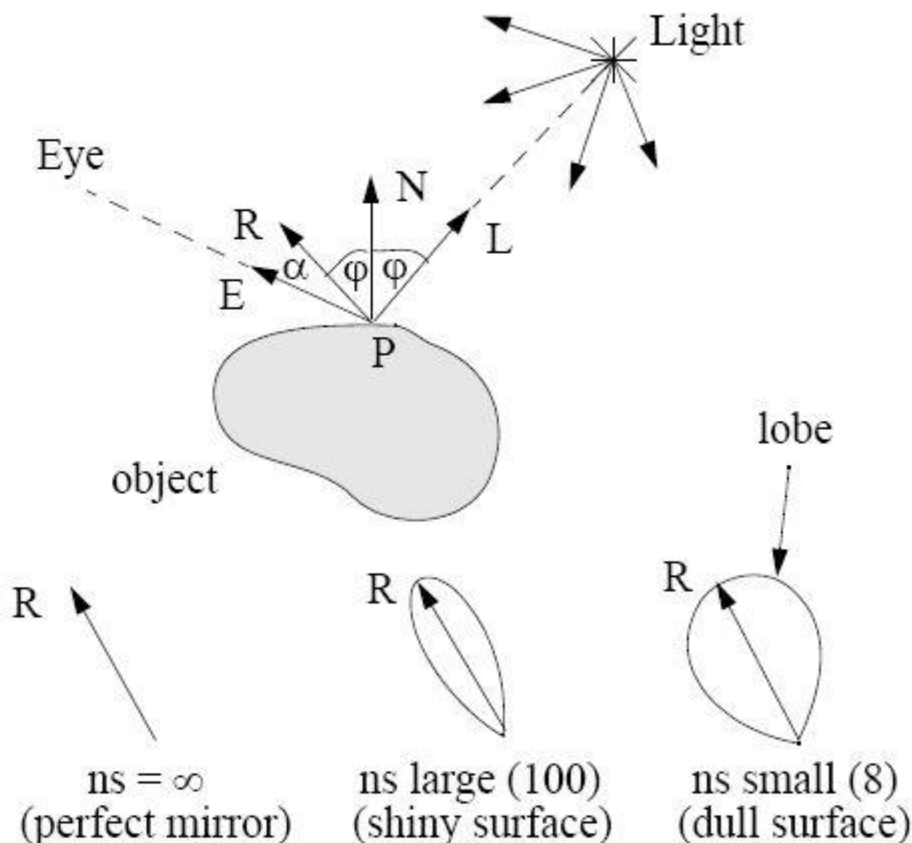
k_d: diffuse reflection coefficient (material constant)

Note: $I_d = 0$ for $N \cdot L < 0$

Specular Reflection - Fundamentals

- Models reflections on shiny surfaces (polished metal, chrome, plastics, etc.)
- Ideal specular reflector (perfect mirror) reflects light only along reflection vector R
- Non-ideal reflectors reflect light in a lobe centered about R
  - cos(α) models this lobe effect
  - the width of the lobe is modeled by Phong exponent ns, it scales cos(α)





Phong specular reflection model:

$I_s = k_s \, I_L \, (\cos \alpha)^{ns} = k_s \, I_L \, (E \cdot R)^{ns}$

I_L: intensity of lightsource

L: light vector

R: reflection vector = 2 N (N·L) - L

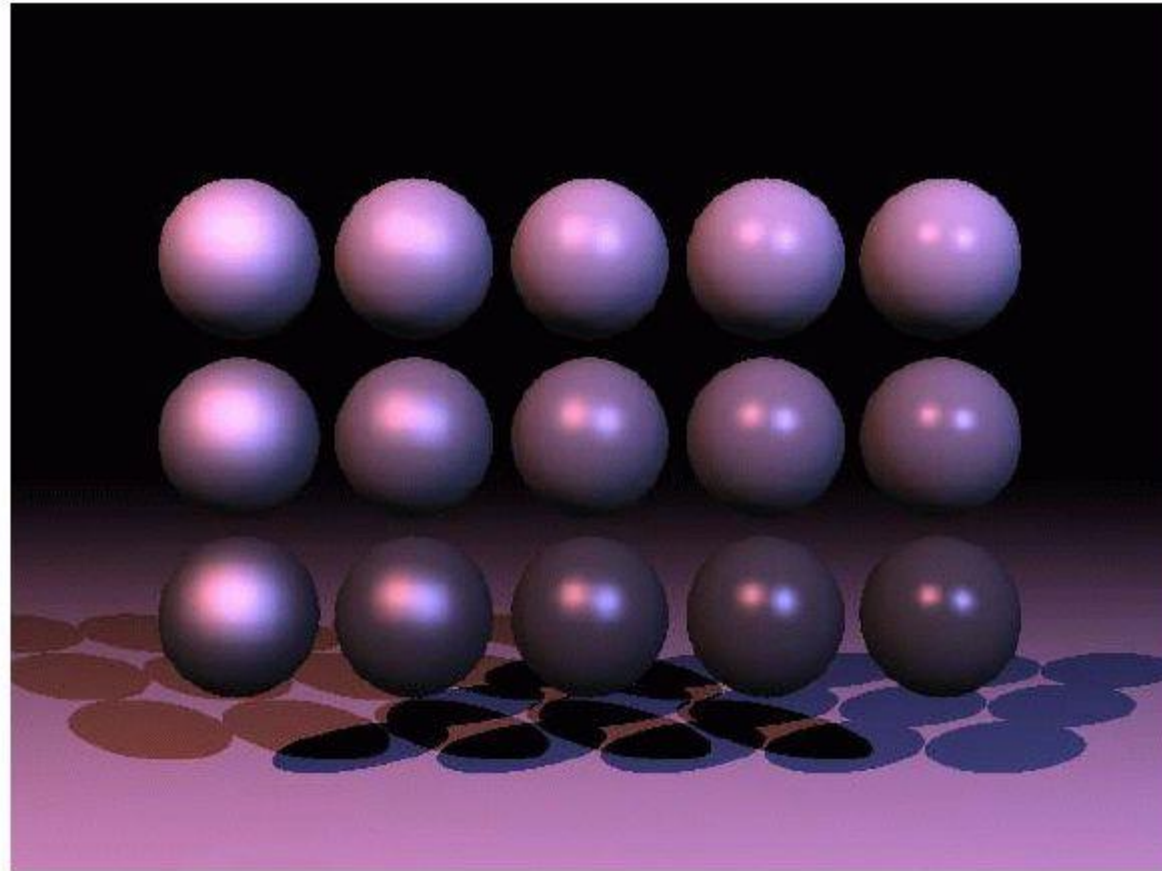
E: eye vector = (Eye-P) / |Eye-P|

α: angle between E and R

ns: Phong exponent

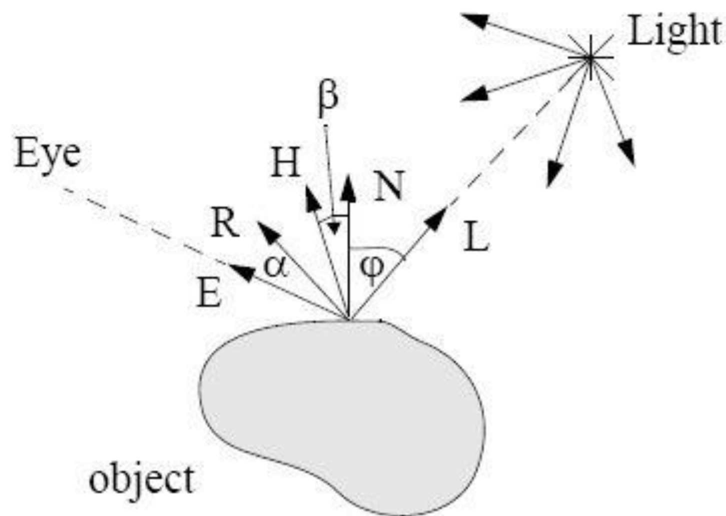
k_s: specular reflection coefficient

Specular and Diffuse Reflection: Varying the Coefficients



Specular Reflection: Using The Half-Vector

- Sometimes the half vector H is used instead of R in specular lighting calculation
- Both alternatives have similar effects



Phong specular reflection model:

$$I_s = k_s\, I_L\, (\cos \beta)^{ns} = k_s\, I_L\, (H \cdot N)^{ns}$$

I_L: intensity of lightsource

L: light vector

H: half vector = (L + E) / |L + E|

R: reflection vector

E: eye vector

Total Reflected Light

- Total reflected light (for a white object):

$$I = k_a I_A + k_d I_L N \cdot L + k_s I_L (H \cdot N)^{ns}$$

- Multiple lightsources:

$$I = k_a I_A + \Sigma (k_d I_i N \cdot L_i + k_s I_i (H_i \cdot N)^{ns})$$

- Usually, I is a color vector of (R=red, G=green, B=blue)
- Object has a color vector $C_{obj} = (R_{obj}, G_{obj}, B_{obj})$
- Object reflects I, modulated (multiplied) by C_{obj}

k_s, k_d, k_a values in [0.0, 1.0]
R, G, B values in [0.0, 1.0]

- Color C reflected by object:

$$C = C_{obj} (k_a I_A + \Sigma (k_d I_i N \cdot L_i)) + \Sigma (k_s I_i (H_i \cdot N)^{ns})$$

- In many applications, the specular color is not modulated by object color
  - specular highlight has the color of the lightsource
- Note: (R, G, B) cannot be larger than 1.0 (later scaled to [0, 255] for display)
  - either set a maximum for each individual term or clamp final colors to 1.0