CSE 690: GPGPU

Lecture 2: Understanding the Fabric - Intro to Graphics

Klaus Mueller

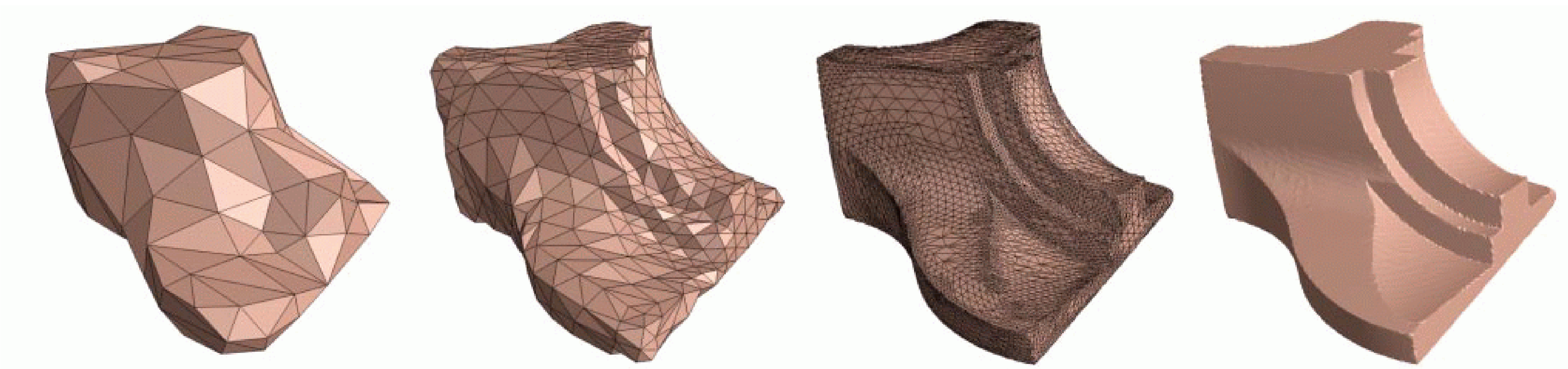
Stony Brook University

Computer Science Department

Surface Graphics

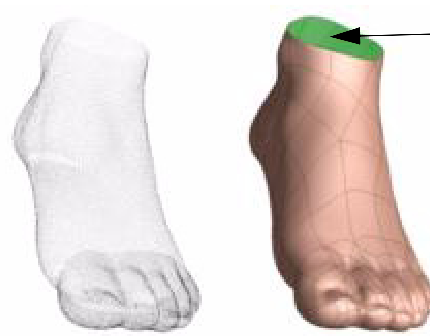
- Objects are explicitely defined by a surface or boundary representation (explicit inside vs outside)
- This boundary representation can be given by:

  - a mesh of polygons:



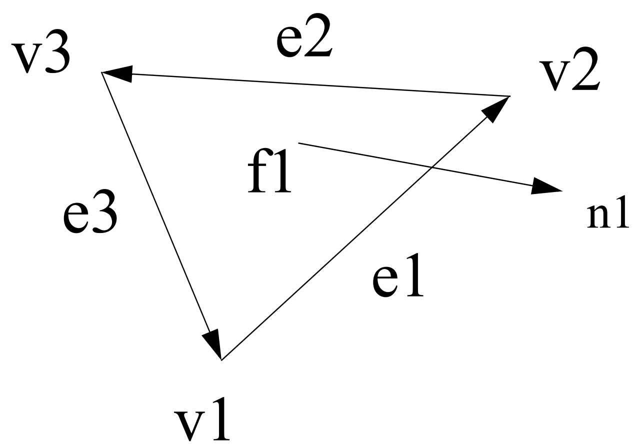
200 polys 1,000 polys 15,000 polys

  - a mesh of spline patches:



an “empty” foot

Polygon Mesh Definitions



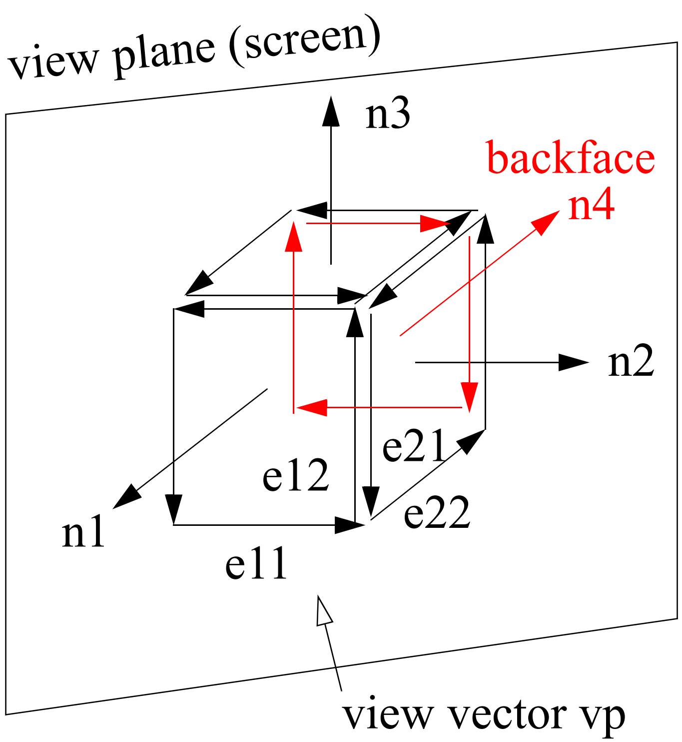
v1, v2, v3: vertices (3D coordinates)

e1, e2, e3: edges

$$e1 = v2 - v1 \quad \text{and} \quad e2 = v3 - v2$$

f1: polygon or *face*

n1: face normal $n1 = \dfrac{e1 \times e2}{|e1 \times e2|}$



$$n1 = \frac{e11 \times e12}{|e11 \times e12|}$$

$$n2 = \frac{e21 \times e22}{|e21 \times e22|}, \quad e21 = -e12$$

Rule: if all edge vectors in a face are ordered counter-clockwise, then the face normal vectors will always point towards the outside of the object.

This enables quick removal of *back-faces* (back-faces are the faces hidden from the viewer):

- back-face condition: $vp \bullet n > 0$

Polygon Mesh Data Structure

- Vertex list (v1, v2, v3, v4, ...):

  (x1, y1, z1), (x2, y2, z2), (x3, y3, z3), (x4, y4, z4), ...

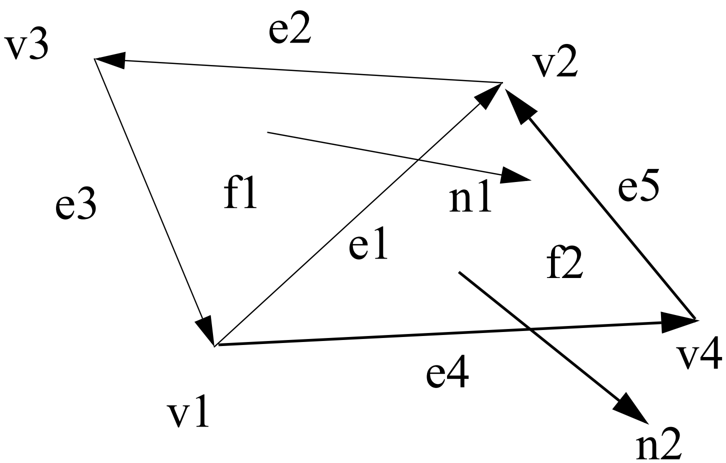
- Edge list (e1, e2, e3, e4, e5, ...):

  (v1, v2), (v2, v3), (v3, v1), (v1, v4), (v4, v2), ...

- Face list (f1, f2, ...):

  (e1, e2, e3), (e4, e5, -e1), ... or

  (v1, v2, v3), (v1, v4, v2), ...

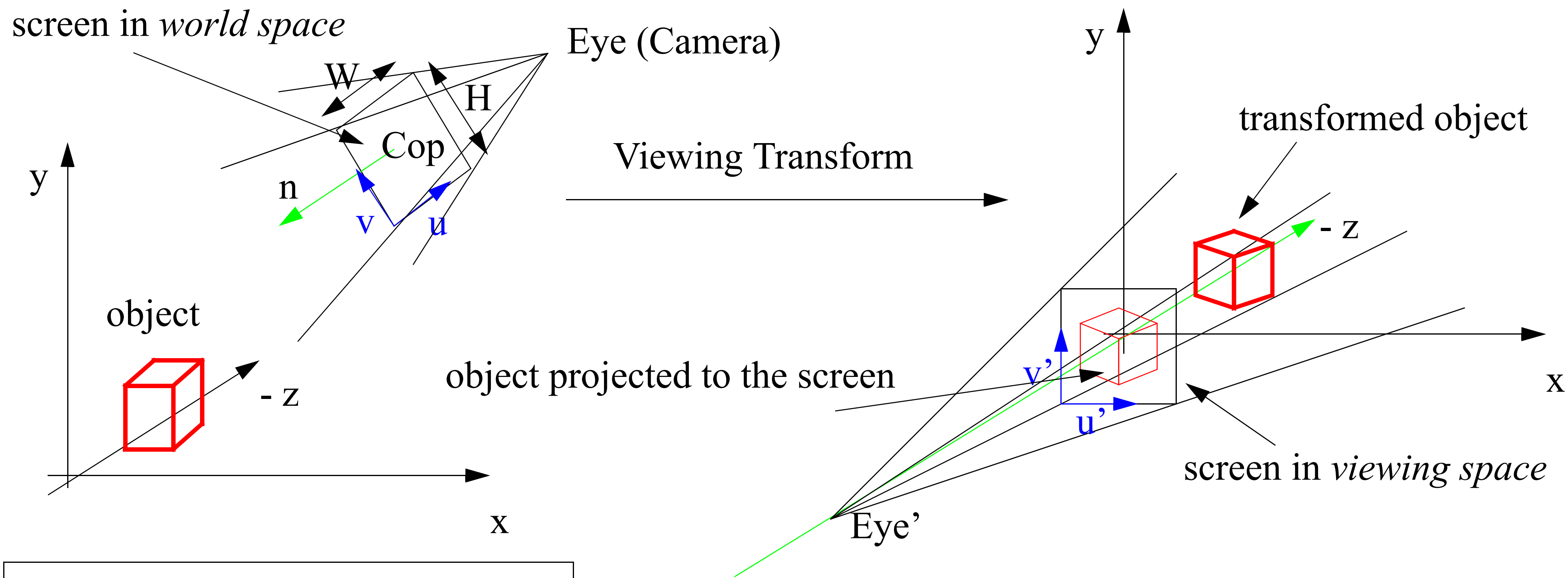


- Normal list (n1, n2, ...), one per face or per vertex

  (n1x, n1y, n1z), (n2x, n2y, n2z), ...

- Use Pointers or indices into vertex and edge list arrays, when appropriate

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 $M_{view} = R_{view} \cdot T_{view}$

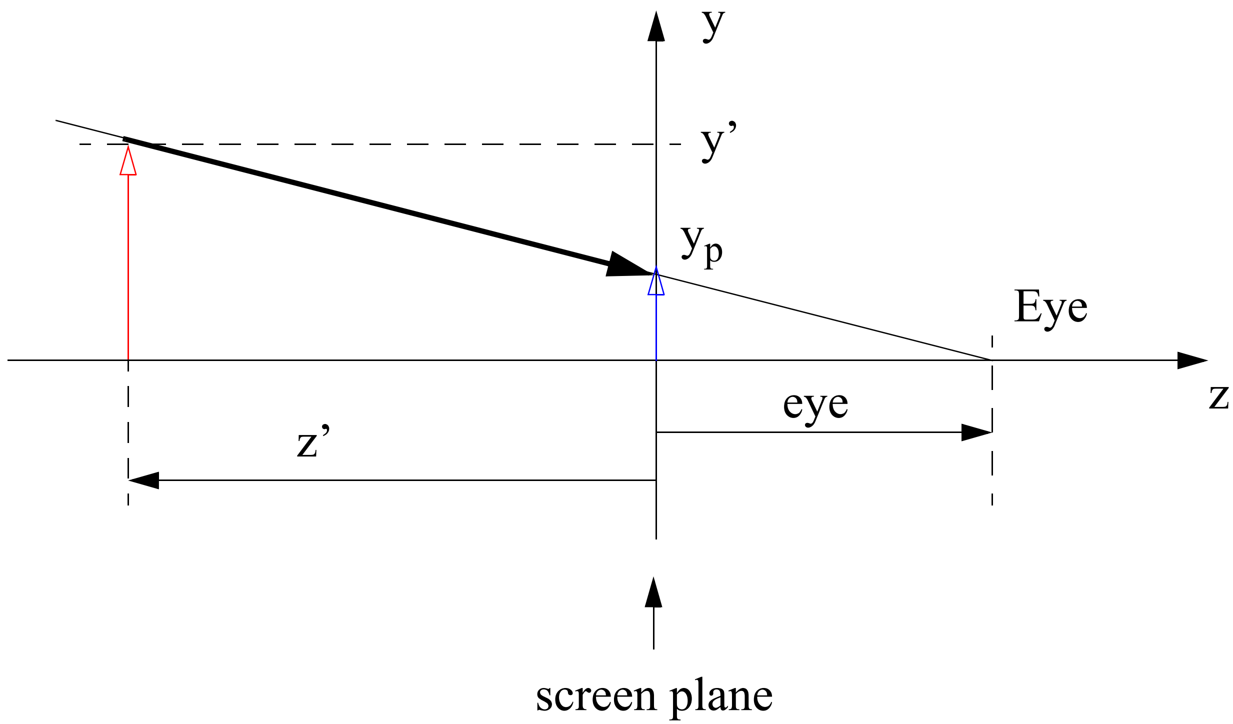
- We transform all object (points, vertices) by M_{view}:

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

- 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)

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

- Perspective projection can also be captured in a matrix M_{proj} with a subsequent *perspective divide* by the homogenous coordinate *w*:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ w \end{bmatrix} = \begin{bmatrix} eye & 0 & 0 & 0 \\ 0 & eye & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & eye \end{bmatrix} \cdot \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

$$x_p = \frac{x_h}{w} \qquad y_p = \frac{y_h}{w}$$

- So the entire *world-to-screen* transform is:

$$M_{trans} = M_{proj} \cdot M_{view} = M_{proj} \cdot R_{view} \cdot T_{view}$$

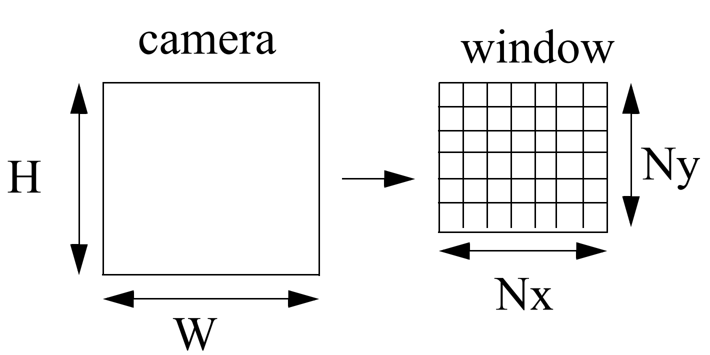
with a subsequent divide by the homogenous coordinate

- M_{trans} is composed only once per view and all object points (vertices) are multiplied by it

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)

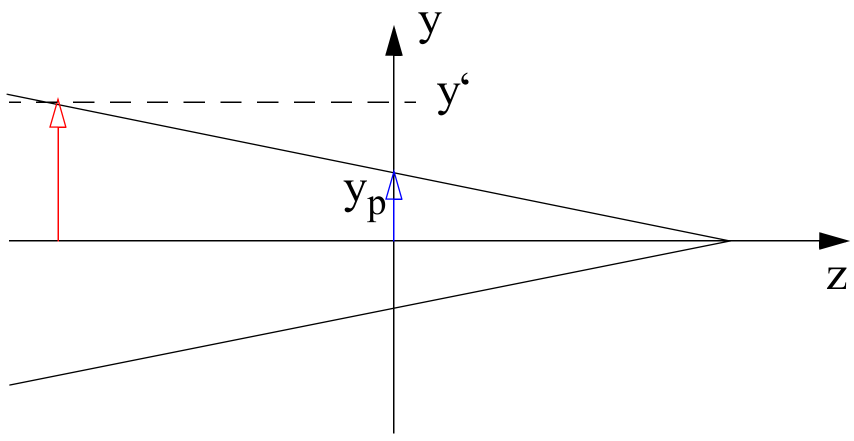
- The window transform can be written as the matrix M_{window}:

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

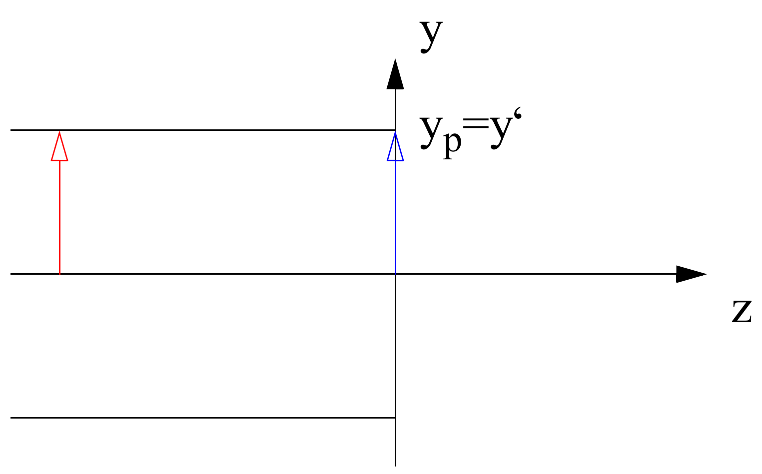
- After the perspective divide, all object points (vertices) are multiplied by M_{window}
- Note: we could figure the window transform into M_{trans}
  - in that case, there is only one matrix multiply per object point (vertex) with a subsequent perspective divide
  - the OpenGL graphics pipeline does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

Orthographic (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



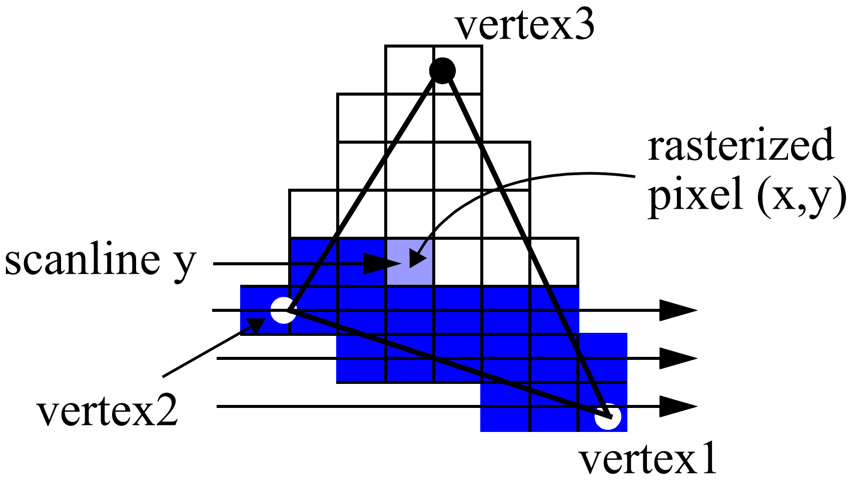
perspective projection



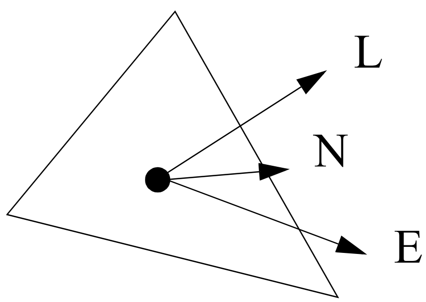
orthographic projection

Polygon Shading Methods - Faceted Shading

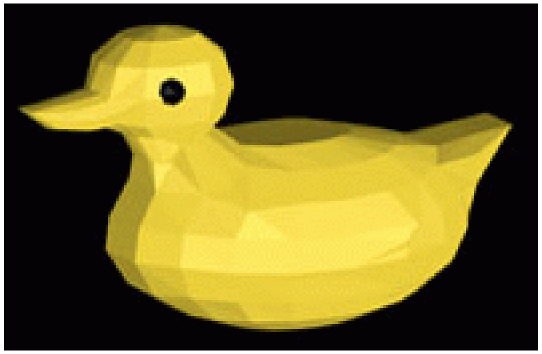
- How are the pixel colors determined in z-buffer?

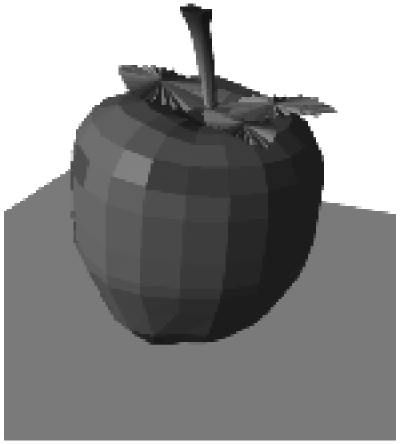


- The simplest method is *flat or faceted shading:*
    - each polygon has a constant color
    - compute color at one point on the polygon (e.g., at center) and use everywhere
    - assumption: lightsource and eye is far away, i.e., N·L, H·E = const.



- Problem: discontinuities are likely to appear at face boundaries

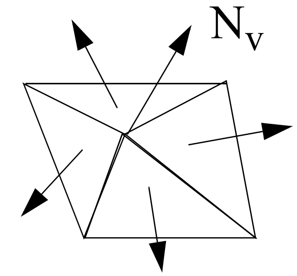




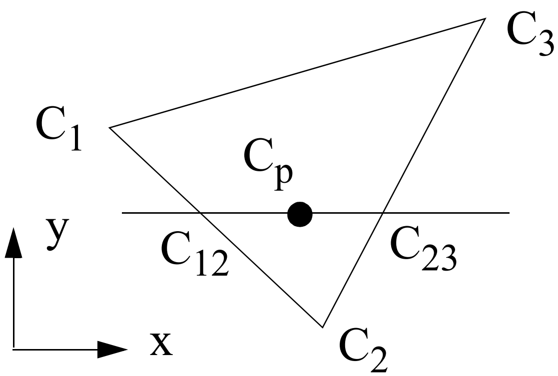
Polygon Shading Methods - Gouraud Shading

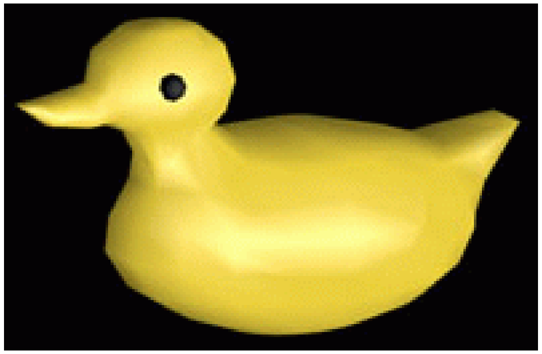
- Colors are averaged across polygons along common edges $\rightarrow$ no more discontinuities
- Steps:
  - determine average unit normal at each poly vertex: $N_v = \sum_{k=1}^{n} N_k \Big/ \left|\sum_{k=1}^{n} N_k\right|$

    n: number of faces that have vertex v in common



  - apply illumination model at each poly vertex $\rightarrow C_v$
  - linearly interpolate vertex colors across edges
  - linearly interpolate edge colors across scan lines





- Downside: may miss specular highlights at off-vertex positions or distort specular highlights

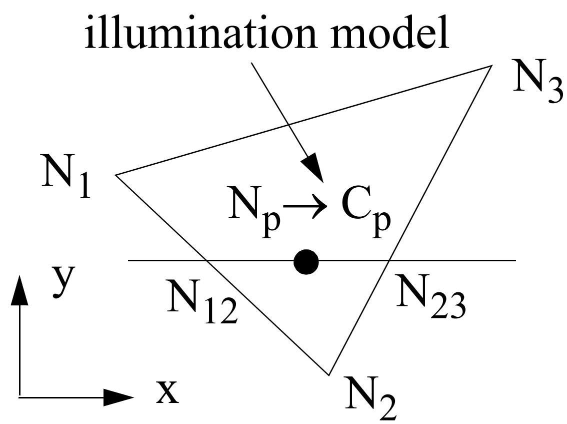
Polygon Shading Methods - Phong Shading

- Phong shading linearly interpolates normal vectors, not colors

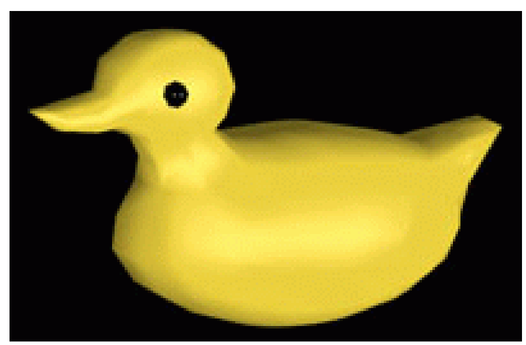
  → more realistic specular highlights

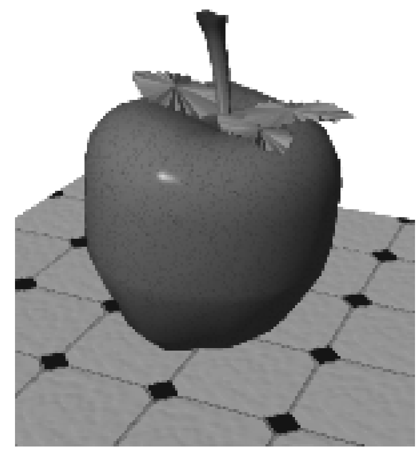
- Steps:

  - determine average normal at each vertex
  - linearly interpolate normals across edges
  - linearly interpolate normals across scanlines



  - apply illumination model at each pixel to calculate pixel color





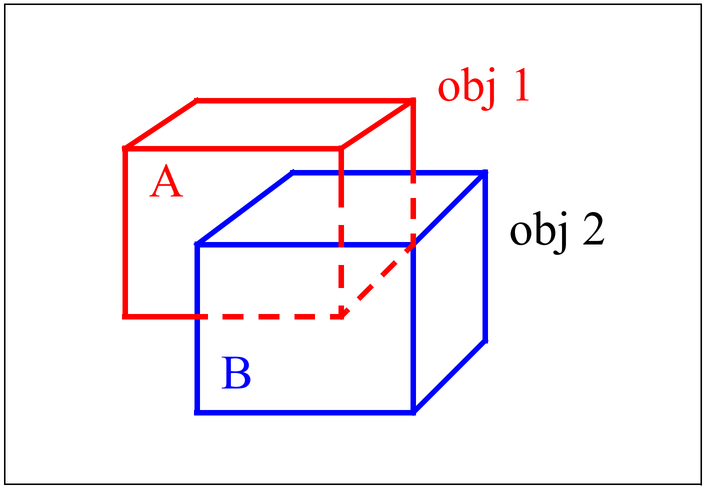
- Downside: need more calculations since need to do illumination model at each pixel

Rendering the Polygonal Objects - The Hidden Surface Removal Problem

- We have removed all faces that are *definitely* hidden: the back-faces
- But even the surviving faces are only *potentially* visible

  - they may be obscured by faces closer to the viewer

face A of object 1 is partially obscured by face B of object 2



screen

- Problem of identifying those face portions that are visible is called the *hidden surface problem*
- Solutions:

  - pre-ordering of the faces and subdivision into their visible parts before display (expensive)
  - the z-buffer algorithm (cheap, fast, implementable in hardware)

The Z-Buffer (Depth-Buffer) Scan Conversion Algorithm

- Two data structures:

  - z-buffer: holds for each image pixel the z-coordinate of the closest object so far

  - color-buffer: holds for each pixel the closest object's color

- Basic z-buffer algorithm:

// initialize buffers

for all (x, y)

    z-buffer(x, y) = -infinity;

    color-buffer(x, y) = $color_{background}$

// scan convert each front-face polygon

for each front-face poly

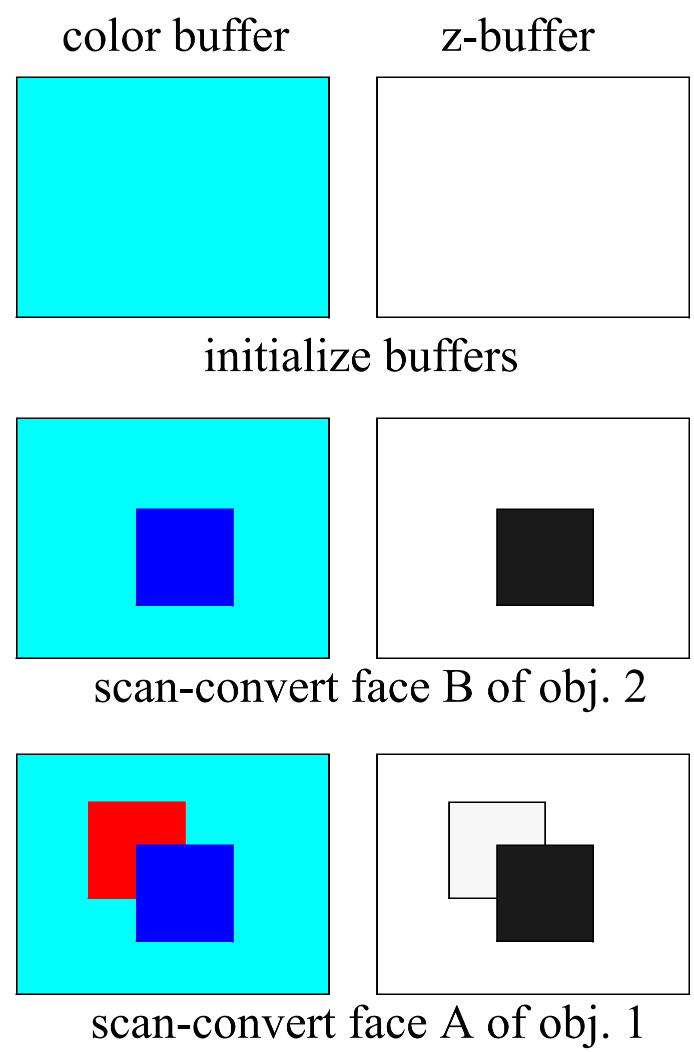
    for each scanline y that traverses projected poly

        for each pixel x in scanline y and projected poly

            if $z_{poly(x, y)}$ > z-buffer(x, y)

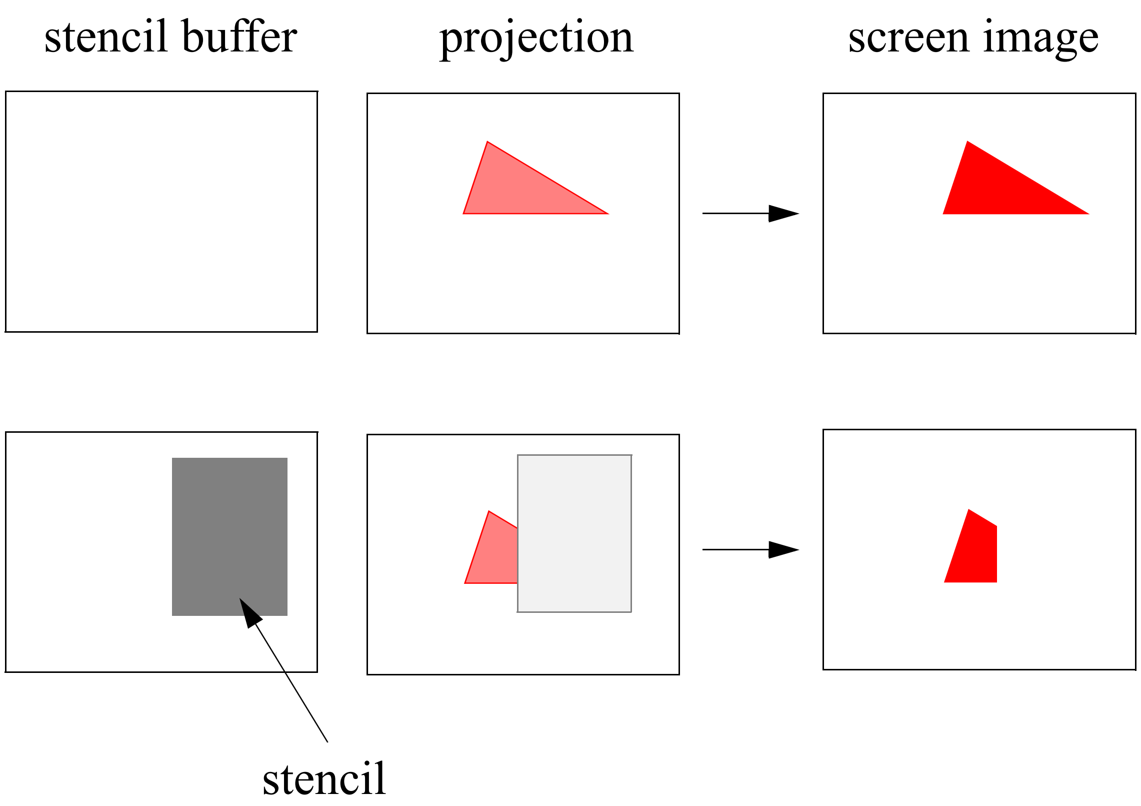
                z-buffer(x, y) = $z_{poly(x, y)}$

                color-buffer(x, y) = $color_{poly(x, y)}$



Stencil Buffer

- Allows a screen area to be “stenciled out”
- No write will occur in these areas on rasterization



Rendering With OpenGl (1)

look also in www.opengl.org

- glMatrixMode(GL_PROJECTION)
- Define the viewing window:

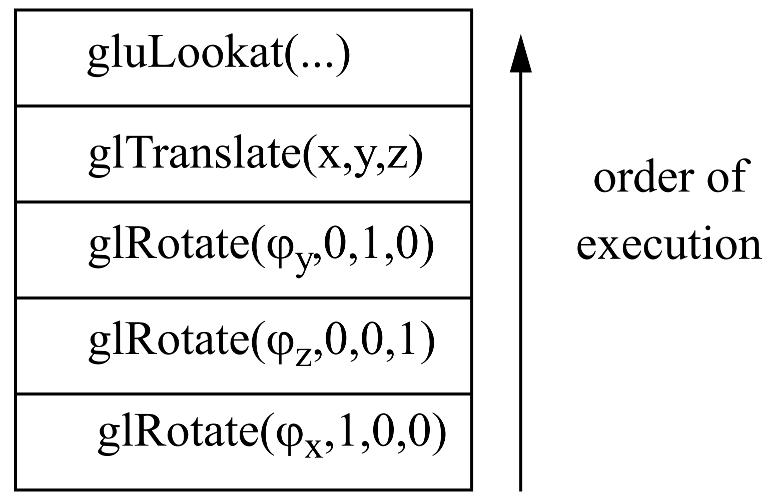
  glOrtho() for parallel projection

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

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

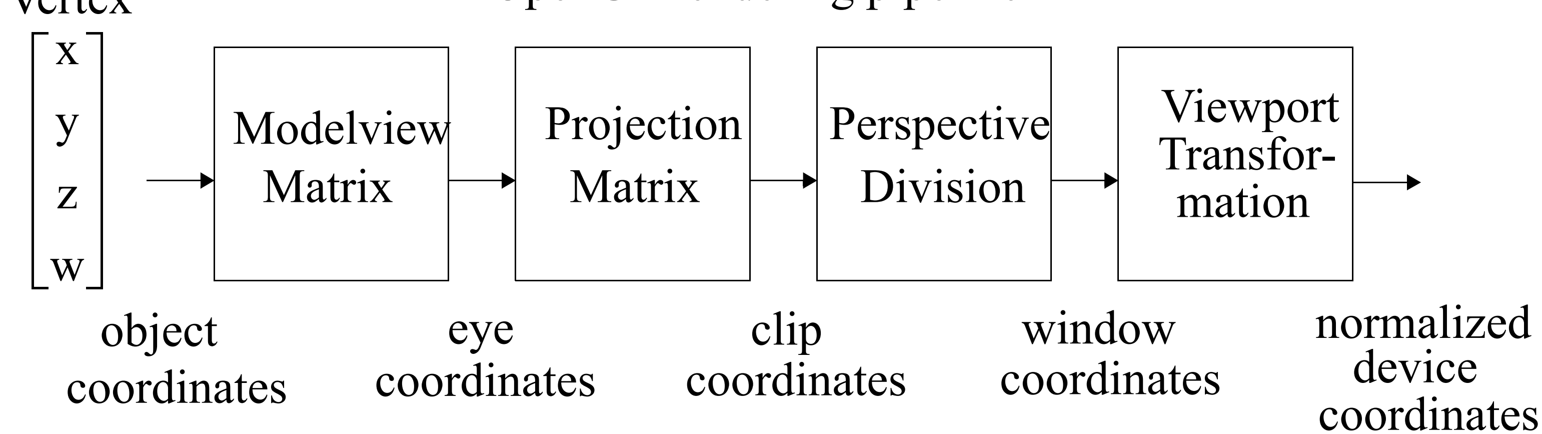
  glTranslate(), glRotate(), glScale(), ...

Modelview Matrix Stack



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

OpenGL rendering pipeline



Rendering With OpenGl (2)

Specify the light sources: glLight()

Enable the z-buffer: glEnable(GL_DEPTH_TEST)

Enable lighting: glEnable(GL_LIGHTING)

Enable stencil test (GL_STENCIL_TEST)

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 */

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 */
```