Surface Graphics

- Objects are explicitly defined by a surface or boundary representation (explicit inside vs outside)
- This boundary representation can be given by:
  - a mesh of polygons:
    - back-face condition: $vp \cdot n > 0$
    - Rule: if all edge vectors in a face are ordered counterclockwise, 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: $vp \cdot n > 0$

- a mesh of spline patches:

Polygon Mesh Definitions

$\begin{align*}
  v_1, v_2, v_3: & \text{ vertices (3D coordinates)} \\
  e_1, e_2, e_3: & \text{ edges} \\
  f_1: & \text{ polygon or face} \\
  n_1: & \text{ face normal}
\end{align*}$

$n_1 = \frac{e_1 \times e_2}{|e_1 \times e_2|}$

Rule: if all edge vectors in a face are ordered counterclockwise, 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):

$\begin{align*}
  n_1 \times n_2 = e_2 \times e_2 - e_1 \times e_1
\end{align*}$

Polygon Mesh Data Structure

- Vertex list ($v_1, v_2, v_3, v_4, ...$):
  $(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4), ...$

- Edge list ($e_1, e_2, e_3, e_4, e_5, ...$):
  $(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_1, v_4), (v_4, v_2), ...$

- Face list ($f_1, f_2, ...$):
  $(v_1, v_2, v_3), (v_1, v_4, v_2), ...$

- Normal list ($n_1, n_2, ...$), one per face or per vertex
  $(n_1x, n_1y, n_1z), (n_2x, n_2y, n_2z), ...$

- 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

Object Display

- See the previous lecture on transforms to learn about object display with GL
- We shall now discuss the parts missing in that lecture:
  - hidden surface removal
  - lighting and illumination
  - shading

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

- 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)
Enabling Lighting and Light Specification (1)

• Lighting in general must be enabled:
  * glEnable(GL_LIGHTING);
• Each individual light must be enabled (OpenGL supports at least 8 lightsources)
  * glEnable(GL_LIGHT0);
• Directional light given by “position” vector
  * GLfloat light_position[] = {-1.0, 1.0, -1.0, 0.0};
  * glLightfv(GL_LIGHT0, GL_POSITION, light_position);
• Point source is given by “position” point
  * GLfloat light_position[] = {-1.0, 1.0, -1.0, 1.0};
  * glLightfv(GL_LIGHT0, GL_POSITION, light_position);
• Set ambient intensity for entire scene (the following is the default setting)
  * GLfloat al[] = {0.2, 0.2, 0.2, 1.0};
  * glLightModelfv(GL_LIGHT_MODEL_AMBIENT, al);

Enabling Lighting and Light Specification (2)

• Use vectors {R,G,B,A} for light source properties
  remember, the light source will be transformed (using the current matrix stack), so it will be as
  if you carried the light in your hand
  * GLfloat light_ambient[] = {0.2, 0.2, 0.2, 1.0};
  * GLfloat light_diffuse[] = {1.0, 1.0, 1.0, 1.0};
  * GLfloat light_specular[] = {1.0, 1.0, 1.0, 1.0};
  * GLfloat light_position[] = {-1.0, 1.0, -1.0, 0.0};
  * glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
  * glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
  * glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
  * glLightfv(GL_LIGHT0, GL_POSITION, light_position);

Material Specifications

• Set material properties (ambient, diffuse, specular, shininess)
  * GLfloat mat_a[] = {0.1, 0.5, 0.8, 1.0};
  * GLfloat mat_d[] = {0.1, 0.5, 0.8, 1.0};
  * GLfloat mat_s[] = {1.0, 1.0, 1.0, 1.0};
  * GLfloat low_sh[] = {5.0};
  * glMaterialfv(GL_FRONT, GL_AMBIENT, mat_a);
  * glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_d);
  * glMaterialfv(GL_FRONT, GL_SPECULAR, mat_s);
  * glMaterialfv(GL_FRONT, GL_SHININESS, low_sh);

Polygon Rasterization

• How are the pixel values determined in the color buffer and the z-buffer?
  Polygon is rasterized, using the vertex values to interpolate the values of the interior pixels
  * we can assign any 4-vector to the vertices: RGBA-color, xyzw-position, or any other 4-vector
  * these values are then linearly interpolated by the rasterizer hardware to create the pixel values
  * for example, assigning the vectors:
    * red = (1.0, 0.0, 0.0, 0.0),
    * green = (0.0, 1.0, 0.0, 0.0),
    * blue = (0.0, 0.0, 1.0, 0.0)
Polygon Shading Methods - Faceted Shading

- 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., $N \cdot L, H \cdot E = \text{const.}$

- Problem: discontinuities are likely to appear at face boundaries

----

Polygon Shading Methods - Gouraud Shading

- Colors are averaged across polygons along common edges → no more discontinuities

- Steps:
  - determine average unit normal at each poly vertex: $N_v = \sum_{k=1}^{n} N_k / \sum_{k=1}^{n} n_k$
    - $n$: number of faces that have vertex $v$ in common
  - apply illumination model at each poly vertex $C_v$
  - 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 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
  - `glTranslatef()`, `glRotatef()`, `glScalef()`, ...

---

Modelview Matrix Stack

- `glMatrixMode(GL_PROJECTION)`
- `glOrtho()` for parallel projection
- `glFrustum()` for perspective projection
- `glMatrixMode(GL_MODELVIEW)`
- `gluLookat()` /* need to have GLUT */
- `glTranslatef()`, `glRotatef()`, `glScalef()`, ...

Order of execution

- rotation first, then translate, then do viewing...

OpenGL rendering pipeline

- Vertex = object coordinates
- OpenGl rendering pipeline
  - Modelview Matrix
  - Projection Matrix
  - Perspective Division
  - Viewport Transformation
- normalized device coordinates
Specify the light sources: glEnable(GL_LIGHT0)
Enable the z-buffer: glEnable(GL_DEPTH_TEST)
Enable lighting: glEnable(GL_LIGHTING)
Enable light source i: glEnable(GL_LIGHTi) /* GL_LIGHTi 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 */