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:
    - 200 polys
    - 1,000 polys
    - 15,000 polys
  - a mesh of spline patches:
    - an “empty” foot
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: $\mathbf{v} \cdot \mathbf{n} > 0$
Polygon Mesh Data Structure

• Vertex list \((v_1, v_2, v_3, v_4, \ldots)\):

\[(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4), \ldots\]

• Edge list \((e_1, e_2, e_3, e_4, e_5, \ldots)\):

\[(v_1, v_2), (v_2, v_3), (v_3, v_1), (v_1, v_4), (v_4, v_2), \ldots\]

• Face list \((f_1, f_2, \ldots)\):

\[(e_1, e_2, e_3), (e_4, e_5, -e_1), \ldots \text{ or} \]
\[(v_1, v_2, v_3), (v_1, v_4, v_2), \ldots\]

• Normal list \((n_1, n_2, \ldots)\), one per face or per vertex

\[(n_{1x}, n_{1y}, n_{1z}), (n_{2x}, n_{2y}, n_{2z}), \ldots\]

• Use Pointers or indices into vertex and edge list arrays, when appropriate
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 origin (T<sub>view</sub>)
  - *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<sub>view</sub>)

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

- We transform all objects (points, vertices) by $M_{\text{view}}$:

$$
\begin{bmatrix}
x' \\
y' \\
z' \\
1
\end{bmatrix} =
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\begin{bmatrix}
u_x & u_y & u_z & 0 \\
v_x & v_y & v_z & 0 \\
n_x & n_y & n_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & -\text{Cop}_x \\
0 & 1 & 0 & -\text{Cop}_y \\
0 & 0 & 1 & -\text{Cop}_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
$$

- 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:

\[
y_p = y' \cdot \frac{\text{eye}}{\text{eye} - z'}
\]

\[
x_p = x' \cdot \frac{\text{eye}}{\text{eye} - z'}
\]

- \(x_p\) 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 $M_{\text{proj}}$ with a subsequent \textit{perspective divide} by the homogenous coordinate $w$:

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

$$
x_p = \frac{x_h}{w}
$$

$$
y_p = \frac{y_h}{w}
$$

- So the entire \textit{world-to-screen} transform is:

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

with a subsequent divide by the homogenous coordinate

- $M_{\text{trans}}$ 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):

\[
x_s = \left( x_p + \frac{W}{2} \right) \cdot \frac{Nx - 1}{W} \quad y_s = \left( y_p + \frac{H}{2} \right) \cdot \frac{Ny - 1}{H}
\]
Step 3: Window Transform (2)

- The window transform can be written as the matrix \( M_{\text{window}} \):

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

- After the perspective divide, all object points (vertices) are multiplied by \( M_{\text{window}} \)

- Note: we could figure the window transform into \( M_{\text{trans}} \)
  - 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
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

- 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:

```plaintext
// 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 \( z_{poly}(x, y) > z\text{-buffer}(x, y) \)
        z-buffer(x, y) = z_{poly}(x, y)
color-buffer(x, y) = color_{poly}(x, y)
```
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., $N \cdot L, H \cdot E = \text{const.}$

- 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:
    \[
    N_v = \frac{\sum_{k=1}^{n} N_k}{\sum_{k=1}^{n}}
    \]
    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
  - `glTranslate()`, `glRotate()`, `glScale()`, ...

Modelview Matrix Stack

<table>
<thead>
<tr>
<th>Order of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>glRotate(φ_x,1,0,0)</code></td>
</tr>
<tr>
<td><code>glRotate(φ_y,0,1,0)</code></td>
</tr>
<tr>
<td><code>glRotate(φ_z,0,0,1)</code></td>
</tr>
<tr>
<td><code>gluLookat(...)</code></td>
</tr>
</tbody>
</table>

OpenGL rendering pipeline

- Vertex coordinates
- Modelview Matrix
- Projection Matrix
- Perspective Division
- Viewport Transformation

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

ModelviewTransformation

Object coordinates

Eye coordinates

Clip coordinates

Window coordinates

Normalized device coordinates

look also in www.opengl.org
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 */

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

At what point do things start looking realistic?
Texture Mapping - Large Walls

Take pictures, map as textures onto large polygon
Texture Mapping Large Walls - OpenGL Program

```c
void TextureMappingFunction_OpenGL(float x, float y)
{
    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();
Complete Graphics Pipeline

polygon primitives

transformed into screen space

vertex transformation

vertex shader (vertex engine)

transformed vertices

rasterization shading texture mapping

fragment shader (rasterizers)

shaded and texture fragments

fragments are interpolated from texture image

screen display (framebuffer)
Graphics Hardware - Peeking Under The Hood

- Graphics hardware accelerates vertex and fragment shaders
  - (almost) fully programmable
  - enables accurate physics and visuals
  - realistic games
  - latest: real-time movie production on the PC
  - accelerate even general purpose, scientific
    and numerical computations (GPGPU)