CSE 564: Computer Graphics

Graphics Foundation

Klaus Mueller

Computer Science Department
Stony Brook University
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
Polygon Mesh Definitions

v1, v2, v3: vertices (3D coordinates)
e1, e2, e3: edges
e1 = v2 - v1 and e2 = v3 - v2
f1: polygon or face

n1: face normal \[ n1 = \frac{e1 \times e2}{|e1 \times e2|} \]

n1 = \frac{e_{11} \times e_{12}}{|e_{11} \times e_{12}|}

n2 = \frac{e_{21} \times e_{22}}{|e_{21} \times e_{22}|}, e_{21} = -e_{12}

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: \( vp \cdot n > 0 \)
Polygons Mesh Data Structure

- **Vertex list** \((v1, v2, v3, v4, \ldots)\):
  \[(x1, y1, z1), (x2, y2, z2), (x3, y3, z3), (x4, y4, z4), \ldots\]

- **Edge list** \((e1, e2, e3, e4, e5, \ldots)\):
  \[(v1, v2), (v2, v3), (v3, v1), (v1, v4), (v4, v2), \ldots\]

- **Face list** \((f1, f2, \ldots)\):
  \[(e1, e2, e3), (e4, e5, -e1), \ldots\ \text{or}\ \ (v1, v2, v3), (v1, v4, v2), \ldots\]

- **Normal list** \((n1, n2, \ldots)\), one per face or per vertex
  \[(n1x, n1y, n1z), (n2x, n2y, n2z), \ldots\]

- Use Pointers or indices into vertex and edge list arrays, when appropriate
Basic Transformations - Translation and Scale

Translation:
translate by $T_x$ along the $x$-axis
translate by $T_y$ along the $y$-axis

$$x' = x + T_x$$
$$y' = y + T_y$$

Scale:
scale by $S_x$ along the $x$-axis
scale by $S_y$ along the $y$-axis

$$x' = S_x \cdot x$$
$$y' = S_y \cdot y$$

If $S_x = S_y$ then scaling is uniform
$S < 1$ shrinks, $S > 1$ enlarges the object

Note: we always scale about the origin

Translate (4, 2)
Scale (0.5, 2)
Basic Transformations - Rotation

A point is represented by polar coordinates \((r, \varphi)\):

\[
\begin{align*}
    x &= r \cos(\varphi) \\
    y &= r \sin(\varphi)
\end{align*}
\]

In this notation, a point after rotation is at:

\[
\begin{align*}
    x' &= r \cos(\varphi + \theta) \\
    y' &= r \sin(\varphi + \theta)
\end{align*}
\]

Using trigonometric identities we get:

\[
\begin{align*}
    x' &= r \cos(\varphi) \cos(\theta) - r \sin(\varphi) \sin(\theta) \\
    y' &= r \sin(\varphi) \cos(\theta) + r \cos(\varphi) \sin(\theta)
\end{align*}
\]

We know that:

\[
\begin{align*}
    x &= r \cos(\varphi) \quad \text{and} \quad y = r \sin(\varphi)
\end{align*}
\]

We can plug this expression into the previous ones:

\[
\begin{align*}
    x' &= x \cos(\theta) - y \sin(\theta) \\
    y' &= x \sin(\theta) + y \cos(\theta)
\end{align*}
\]

Note: If \(\theta > 0\) then the rotation is counter-clockwise.
Matrix Notation and Extension to 3D

• Scale: 
\[
\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} = 
\begin{bmatrix}
sx & 0 & 0 \\
0 & sy & 0 \\
0 & 0 & sz
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
\]

• Rotation about the z-axis: 
\[
\begin{bmatrix}
x' \\
y' \\
z'
\end{bmatrix} = 
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z
\end{bmatrix}
\]

• What about translation?
  - recall, we’re adding Tx, Ty, and Tz ..... without multiplying by a coordinate

• Solution: use homogenous coordinates 
\[
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]
Transformations in Homogenous Coordinates

- Translation (T):
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} =
  \begin{bmatrix}
  1 & 0 & 0 & Tx \\
  0 & 1 & 0 & Ty \\
  0 & 0 & 1 & Tz \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

- Scale (S):
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} =
  \begin{bmatrix}
  sx & 0 & 0 & 0 \\
  0 & sy & 0 & 0 \\
  0 & 0 & sz & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

- Rotation about the z-axis (Rz):
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} =
  \begin{bmatrix}
  \cos \theta & -\sin \theta & 0 & 0 \\
  \sin \theta & \cos \theta & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

- Rotation about the x-axis (Rx):
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} =
  \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos \theta & -\sin \theta & 0 \\
  0 & \sin \theta & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]

- Rotation about the y-axis (Ry):
  \[
  \begin{bmatrix}
  x' \\
  y' \\
  z' \\
  1
  \end{bmatrix} =
  \begin{bmatrix}
  \cos \theta & 0 & \sin \theta & 0 \\
  0 & 1 & 0 & 0 \\
  -\sin \theta & 0 & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z \\
  1
  \end{bmatrix}
  \]
Combining Transformations

- When an object is transformed, all its vertices $v_i$ need to be transformed to $v'_i$:
  \[ v'_i = T \cdot R_z \cdot S \cdot v_i = [T \cdot R_z \cdot S] \cdot v_i = M_t \cdot v_i \]

Combining the transformations into composite matrix $M_t$ minimizes the matrix-vector calculations.
Transformation About an Arbitrary Point in Space

- The standard matrices given in the past few slides only allow you to rotate and scale an object about the (world) origin (Note: translation is an exception)

- What if you wanted to rotate or scale an object around an arbitrary point in space, say its center?

\[ v_i' = T_2 \cdot R_z \cdot T_1 \cdot v_i = [T_2 \cdot R_z \cdot T_1] \cdot v_i = M_{r\text{-arbitrary\_point}} \cdot v_i \]
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 ($T_{\text{view}}$)
  - *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 ($R_{\text{view}}$)

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

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

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

• 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 *perspective divide* by the homogenous coordinate $w$:

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

$$
\begin{align*}
  x_p &= \frac{x_h}{w} \\
  y_p &= \frac{y_h}{w}
\end{align*}
$$

- So the entire *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):

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

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

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

- 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
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:

```c
// 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)
```

![Diagram](image_url)
Illumination

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

\[
I_d = k_d I_L \cos \varphi = k_d I_L \mathbf{N} \cdot \mathbf{L}
\]

Lambertian cosine law:

\[
I_d = k_d I_L \cos \varphi = k_d I_L \mathbf{N} \cdot \mathbf{L}
\]

\[\mathbf{L} = \frac{\mathbf{Light} - \mathbf{P}}{|\mathbf{Light} - \mathbf{P}|} = \frac{\mathbf{Light}_x - P_x}{|\mathbf{L'}|}, \frac{\mathbf{Light}_y - P_y}{|\mathbf{L'}|}, \frac{\mathbf{Light}_z - P_z}{|\mathbf{L'}|}
\]

\[|\mathbf{L'}| = \sqrt{(\mathbf{Light}_x - P_x)^2 + (\mathbf{Light}_y - P_y)^2 + (\mathbf{Light}_z - P_z)^2}
\]

\[
\mathbf{N} \cdot \mathbf{L} = (N_x L_x + N_y L_y + N_z L_z)
\]

Dot product:

\[
\mathbf{N} \cdot \mathbf{L} = (N_x L_x + N_y L_y + N_z L_z)
\]
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 $R$
  - $\cos(\alpha)$ models this lobe effect
  - the width of the lobe is modeled by Phong exponent $n_s$, it scales $\cos(\alpha)$

### Phong specular reflection model:

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

- $I_L$: intensity of lightsource
- $L$: light vector
- $R$: reflection vector $= 2 N (N \cdot L) - L$
- $E$: eye vector $= (\text{Eye}-P) / |\text{Eye}-P|$
- $\alpha$: angle between $E$ and $R$
- $n_s$: Phong exponent
- $k_s$: specular reflection coefficient

- $n_s = \infty$ (perfect mirror)
- $n_s$ large (100) (shiny surface)
- $n_s$ small (8) (dull surface)
Specular and Diffuse Reflection - Varying the Coefficients

diffuse coefficient $k_d$

Phong exponent $n_s$
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:

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

- \( I_L \): 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:

\[ I = k_a I_A + \sum (k_d I_i N \cdot L_i + k_s I_i (H_i \cdot N)^{ns}) \]

- Usually, I is a color vector of (R=red, G=green, B=blue)
- Object has a color vector \( C_{\text{obj}} = (R_{\text{obj}}, G_{\text{obj}}, B_{\text{obj}}) \)
- Object reflects I, modulated by \( C_{\text{obj}} \)
- Color C reflected by object:

\[ C = C_{\text{obj}} (k_a I_A + \sum (k_d I_i N \cdot L_i)) + \sum (k_s I_i (H_i \cdot N)^{ns}) \]

- 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
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: 
    \[
    \mathbf{N}_v = \frac{n}{\sum_{k=1}^{n} N_k} \left( \sum_{k=1}^{n} \frac{N_k}{N_k} \right)
    \]
    \(n\): 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
• `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

```
gluLookat(...)  
glTranslate(x,y,z)  
glRotate(\phi_y,0,1,0)  
glRotate(\phi_z,0,0,1)  
glRotate(\phi_x,1,0,0)  
```

order of execution

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

OpenGL rendering pipeline

```
Vertex  
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>w</th>
</tr>
</thead>
</table>
object coordinates → Modelview Matrix → Projection Matrix → Perspective Division → Viewport Transformation

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 */

```c
    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 */
Example: Scene Graph Bike

\[ T_d = \text{glTranslate}(\text{dist}) \] // translate bike

\[ \text{glPush()} \] // duplicate \( T_d \) on the stack
\[ T_f = \text{glTranslate}(+w_1 \rightarrow O) \]
\[ R = \text{glRotate}(\text{angle}) \]
\[ T_b = \text{glTranslate}(-w_1 \rightarrow O) \]
\[ \text{Render}(w_1) \] // \( T_d T_b R T_f w_1 \)
\[ \text{glPop()} \] // expose \( T_d \)

\[ \text{glPush()} \] // duplicate \( T_d \)
\[ \text{glTranslate}(+w_2 \rightarrow O) \]
\[ \text{glRotate}(\text{angle}) \]
\[ \text{glTranslate}(-w_2 \rightarrow O) \]
\[ \text{Render}(w_2) \] // \( T_d T_b R T_f w_1 \)
\[ \text{glPop()} \] // expose \( T_d \)

\[ \text{Render}(\text{frame}) \] // \( T_d f \)