Review 2
Lines and Curves

- Explicit - one coordinate as function of the others

\[ y = f(x) \]
\[ z = f(x, y) \]

**line**
\[ y = mx + b \]
\[ y = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1) + y_1 \]

**circle**
\[ y = \pm \sqrt{r^2 - x^2} \]
Lines and Curves

- Parametric – all coordinates as functions of common parameter

\[(x, y) = (f_1(t), f_2(t))\]
\[(x, y, z) = (f_1(u, v), f_2(u, v), f_3(u, v))\]

**line**

\[x(t) = x_1 + t(x_2 - x_1)\]
\[y(t) = y_1 + t(y_2 - y_1)\]
\[t \in [0,1]\]

**circle**

\[x(\theta) = r \cos(\theta)\]
\[y(\theta) = r \sin(\theta)\]
\[\theta \in [0,2\pi]\]
Lines and Curves

- Implicit - define as “zero set” of function of all the parameters

\[
\{(x, y) : F(x, y) = 0\} \\
\{(x, y, z) : F(x, y, z) = 0\}
\]

- Defines partition of space

\[
\{(x, y) : F(x, y) > 0\}, \{(x, y) : F(x, y) = 0\}, \{(x, y) : F(x, y) < 0\}
\]
Lines and Curves - Implicits

**line**

\[ dy = y_2 - y_1 \]

\[ dx = x_2 - x_1 \]

\[ F(x, y) = (x - x_1)dy - (y - y_1)dx \]

\[ F(x, y) = 0 \quad \text{(x,y) is on line} \]

\[ F(x, y) > 0 \quad \text{(x,y) is below line} \]

\[ F(x, y) < 0 \quad \text{(x,y) is above line} \]

\[ F(x, y) = xdy - ydx + (y_1dx - x_1dy) \]

**circle**

\[ F(x, y) = x^2 + y^2 - r^2 \]

\[ F(x, y) = 0 \quad \text{(x,y) is on circle} \]

\[ F(x, y) > 0 \quad \text{(x,y) is outside} \]

\[ F(x, y) < 0 \quad \text{(x,y) is inside} \]
Basic Line Drawing

Assume \( x_1 < x_2 \) \& line slope absolute value is \( \leq 1 \)

\[
\textbf{Line} \ (x_1, y_1, x_2, y_2)
\begin{align*}
\text{begin} \\
\text{float} \ dx, \ dy, \ x, \ y, \ slope \ ; \\
\text{dx} \triangleq x_2 - x_1; \\
\text{dy} \triangleq y_2 - y_1; \\
\text{slope} \triangleq \frac{dy}{dx}; \\
y \triangleq y_1 \\
\text{for } x \text{ from } x_1 \text{ to } x_2 \text{ do} \\
\text{begin} \\
\textbf{PlotPixel} \ (x, \text{Round}(y)); \\
y \leftarrow y + \text{slope} \ ; \\
\text{end} \\
\text{end} ;
\end{align*}
\]

Questions:
Can this algorithm use integer arithmetic?
How do we draw other curves?
e.g. \( y = x^2 \) between \( x_1 \) and \( x_2 \)
Midpoint (Bresenham) Algorithm

**Assumptions:**

\[ x_2 > x_1, y_2 > y_1 \text{ and } \frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1} < 1 \]

**Idea:**

- Proceed along the line incrementally
- Have ONLY 2 choices
- Select one that minimizes error (distance to line)
Bresenham Algorithm

Distance (error):

\[ \tau = \{(x, y) \mid ax + by + c = xdy - ydx + c = 0\} \]

\[ d(x, y) = 2( xdy - ydx + c) \]

- Given point \( P = (x, y), d(x, y) \) is signed distance of \( P \) to \( \tau \) (up to normalization factor)
- \( d \) is zero for \( P \in \tau \)
Midpoint Line Drawing (cont’d)

- Starting point satisfies \( d(x_1, y_1) = 0 \)
- Each step moves right (east) or upper right (northeast)
- Sign of \( d(x + 1, y + \frac{1}{2}) \) indicates if to move east or northeast
Midpoint Line Algorithm (version 1)

```c
Line (x₁, y₁, x₂, y₂)
begin
int x, y, dx, dy, d;
x ← x₁;
y ← y₁;
dx ← x₂ - x₁;
dy ← y₂ - y₁;
PlotPixel (x, y);
while (x < x₂) do
    d = (2x + 2)dy - (2y + 1)dx + 2c; // 2((x + 1)dy - (y + .5)dx + c)
    if (d < 0) then
        begin
            x ← x + 1;
        end;
    else begin
        x ← x + 1;
        y ← y + 1;
    end;
    PlotPixel (x, y);
end;
end;
```

![Diagram of Bresenham's Line Algorithm](image)
Triangulation

- Convex polygons easily triangulated
- Concave polygons present a challenge
- Convexity - formal definition:

Object $S$ is convex iff for any two points $P, Q \in S$, $tP + (1-t)Q \subseteq S$, $t \in [0,1]$. 
Flood Fill Algorithm

Input

- polygon $P$ with rasterized edges
- $P = (x,y) \in P$ point inside $P$
Scanline Algorithm

- Observation: Each intersection of straight line with boundary moves it from/into polygon
- Detect (& set) pixels inside polygon boundary (simple closed curve) with set of horizontal lines (pixel apart)
for (y=yB; y<=yT; y++) {
    for (x=xL; x<=xR; x++)
        setPixel(x,y);
    xL += DxL;
    xR += DxR;
}

\[ \Delta x_L \quad \Delta x_R \]

\[ \Delta y \]

\[ (y_B, x_L, x_R) \]

\[ (y_T, x') \]
Define a triangle from implicit edge equations:
Barycentric Coordinates

- **Area**
  \[ A = \frac{1}{2} \left\| \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3} \right\| \]

- **Barycentric coordinates**
  \[ a_1 = \frac{A_{P_2P_3P}}{A}, \quad a_2 = \frac{A_{P_3P_1P}}{A}, \quad a_3 = \frac{A_{P_1P_2P}}{A}, \]
  \[ P = a_1 P_1 + a_2 P_2 + a_3 P_3 \]
Computing Barycentric Coords

combining

\[
P = \frac{c_2}{c_1 + c_2} \cdot P_L + \frac{c_1}{c_1 + c_2} \cdot P_R
\]

gives

\[
P_L = \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3
\]

\[
P_R = \frac{b_2}{b_1 + b_2} P_2 + \frac{b_1}{b_1 + b_2} P_1
\]
Cohen-Sutherland Algorithm (cont’d)

C-S-Clip( \( P_0 = (x_0, y_0), P_1 = (x_1, y_1), x_{\min}, x_{\max}, y_{\min}, y_{\max} \) )

\( C_0 \leftarrow \text{code}(P_0); \quad C_1 \leftarrow \text{code}(P_1); \)

if \( (C_0 \text{ and } C_1) \neq 0 \) then return;

if \( (C_0 \text{ or } C_1) = 0 \) then draw(\( P_0, P_1 \));

else if (OutsideWindow(\( P_0 \)) then

begin

Edge \( \leftarrow \) Window boundary of leftmost non-zero bit of \( C_0 \);

\( P_2 \leftarrow P_0, P_1 \cap Edge; \)

C-S-Clip( \( P_2, P_1, x_{\min}, x_{\max}, y_{\min}, y_{\max} \) );

end

derel

else

Edge \( \leftarrow \) Window boundary of leftmost non-zero bit of \( C_1 \);

\( P_2 \leftarrow P_0, P_1 \cap Edge; \)

C-S-Clip( \( P_0, P_2, x_{\min}, x_{\max}, y_{\min}, y_{\max} \) );

end

<table>
<thead>
<tr>
<th>bit</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( y &lt; y_{\min} )</td>
<td>( y \geq y_{\min} )</td>
</tr>
<tr>
<td>2</td>
<td>( y &gt; y_{\max} )</td>
<td>( y \leq y_{\max} )</td>
</tr>
<tr>
<td>3</td>
<td>( x &gt; x_{\max} )</td>
<td>( x \leq x_{\max} )</td>
</tr>
<tr>
<td>4</td>
<td>( x &lt; x_{\min} )</td>
<td>( x \geq x_{\min} )</td>
</tr>
</tbody>
</table>
C-S Algorithm for convex polygons – full version

C-S-Clip( poly = P₀,..., Pₙ, xₘᵣ₂, xₘᵢₓ, yₘᵢₙ, yₘᵃₓ )
for i = 1 to n Cᵢ ← code( Pᵢ );
if (( C₀ and C₁ and ... and Cₙ ) ! = 0 ) then return;
if (( C₀ or C₁ or ... or Cₙ ) == 0 ) then draw( poly );
else
for i = 1 to n if (OutsideWindow( Pᵢ ) ) then
begin
    Edge ← Window boundary of leftmost non-zero bit of Cᵢ;
    Pᵢ₋₁,i ← Pᵢ₋₁, Pᵢ ∩ Edge; /* if no intersection return Pᵢ₋₁ */
    Pᵢ,i₊₁ ← Pᵢ, Pᵢ₊₁ ∩ Edge; /* if no intersection return Pᵢ₊₁ */
    if ( Pᵢ₋₁,i == Pᵢ₋₁ and Pᵢ,i₊₁ == Pᵢ₊₁ )
        C-S-Clip( P₀,..., Pᵢ₋₁, Pᵢ₊₁,..., Pₙ, xₘᵣ₂, xₘᵢₓ, yₘᵢₙ, yₘᵃₓ )
    else if ( Pᵢ₋₁,i == Pᵢ₋₁ ) /* no intersection, or exactly on the end-vertex */
        C-S-Clip( P₀,..., Pᵢ₋₁, Pᵢ₊₁,..., Pₙ, xₘᵣ₂, xₘᵢₓ, yₘᵢₙ, yₘᵃₓ );
    else if ( Pᵢ,i₊₁ == Pᵢ₊₁ ) /* no intersection, or exactly on the end-vertex */
        C-S-Clip( P₀,..., Pᵢ₋₁, Pᵢ₋₁,i, Pᵢ₊₁,..., Pₙ, xₘᵣ₂, xₘᵢₓ, yₘᵢₙ, yₘᵃₓ );
    else
        C-S-Clip( P₀,..., Pᵢ₋₁, Pᵢ₋₁,i, Pᵢ,i₊₁,..., Pₙ, xₘᵣ₂, xₘᵢₓ, yₘᵢₙ, yₘᵃₓ );
end

<table>
<thead>
<tr>
<th>bit</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y &lt; yₘᵢₙ</td>
<td>y ≥ yₘᵢₙ</td>
</tr>
<tr>
<td>2</td>
<td>y &gt; yₘₐₓ</td>
<td>y ≤ yₘₐₓ</td>
</tr>
<tr>
<td>3</td>
<td>x &gt; xₘᵢₙ</td>
<td>x ≤ xₘᵢₙ</td>
</tr>
<tr>
<td>4</td>
<td>x &lt; xₘᵢₙ</td>
<td>x ≥ xₘᵢₙ</td>
</tr>
</tbody>
</table>
Back Face Culling (object space)

- In closed polyhedron you don’t see object “back” faces

- Assumption
  - Normals of faces point \textit{out} from the object
BSP Trees

- Convention: Right sibling in \( N_p \) direction
- BSP Tree is *view independent*
- Constructed using only object geometry
- Can be used in hidden surface removal from multiple views
- How to choose what is visible for given view?
Z-Buffer

ZBuffer(Scene)
For every pixel (x,y) do PutZ(x,y,MaxZ);
For each polygon P in Scene do
    Q := Project(P);
    For each pixel (x,y) in Q do
        z1 := Depth(Q,x,y);
        if (z1<GetZ(x,y)) then
            PutZ(x,y,z1);
            PutColor(x,y,Col(P));
        end;
    end;
end;
Transparency/Object Buffer

- A-buffer - extension to Z-buffer
- Save all pixel values
- At the end – have list of polygons & depths (order) for each pixel
- Simulate transparency by weighting different list elements
Light Sources

- **Point source**
  - Light originates at a point
  - Rays hit planar surface at different angles

- **Parallel source**
  - Light rays are parallel
  - Rays hit a planar surface at identical angles
  - May be modeled as point source at infinity

*Directional light*
Light

- Light has color
- Interacts with object color \((r, g, b)\)

\[
I = I_a k_a
\]

\[
I_a = (I_{ar}, I_{ag}, I_{ab})
\]

\[
k_a = (k_{ar}, k_{ag}, k_{ab})
\]

\[
I = (I_r, I_g, I_b) = (I_{ar} k_{ar}, I_{ag} k_{ag}, I_{ab} k_{ab})
\]

- Blue light on white surface?
- Blue light on red surface?
Illumination equation is now:

\[ I = I_a k_a + I_p k_d (N \cdot L) = I_a k_a + I_p k_d \cos \theta \]

- \( I_p \) - point/parallel source’s intensity
- \( k_d \) - surface diffuse reflection coefficient

Can we locate light source from shading?
Specular Reflection

- Shiny objects (e.g. metallic) reflect light in preferred direction $R$ determined by surface normal $N$.

- Most objects are not ideal mirrors - reflect in the immediate vicinity of $R$. 
Illumination Equation

- For multiple light sources:

\[
I = I_a k_a + \sum_p \frac{I_p}{d_p^2} \left( k_d (N \cdot L_p) + k_s (R_p \cdot V)^n \right)
\]

- \(d_p\) - distance between surface and light source
  + distance between surface and viewer
  (Heuristic atmospheric attenuation)
Flat Shading

- Illumination value depends only on polygon normal
  - each polygon colored with uniform intensity
- Not adequate for polygons approximating smooth surface
- Looks non-smooth
  - worsened by Mach bands effect
Gourard Shading

- Polyhedron - approximation of smooth surface
  - Assign to each vertex normal of original surface at point
  - If surface not available use estimate normal
- Compute illumination intensity at vertices using those normals
- Linearly interpolate vertex intensities over interior pixels of polygon projection

\[ n_1 \quad n_2 \quad n_3 \]
Phong Shading

- Interpolate (in image space) normal vectors instead of intensities
- Apply illumination equation for each interior pixel with its own normal

\[ n_4 = \alpha_1 n_1 + (1 - \alpha_1) n_2 \]
\[ n_5 = \alpha_2 n_1 + (1 - \alpha_2) n_3 \]
\[ n(x, y) = \alpha_3 n_4 + (1 - \alpha_3) n_5 \]
\[ c(x, y) = Ill(n(x, y)) \]
Texture Mapping

(u, v) parameterization in OpenGL
Example Texture Map

glTexCoord2d(4, 4);
glVertex3d (x, y, z);

(4,4)
(4,0)
(0,4)
(0,0)
(1,0)
(0,0)
(0,1)
(1,1)

glTexCoord2d(1, 1);
glVertex3d (x, y, z);

(1,0)
(1,1)
(0,0)
(0,1)
Texture Mapping

- Texture coordinate interpolation
  - Perspective foreshortening problem
  - Also problematic for color interpolation, etc.
MIP-mapping

without

with
Volumetric Texture - Principles

- 3D function $\rho$
  
  - $\rho = \rho(x,y,z)$

- Texture Space – 3D space that holds the texture (discrete or continuous)

- Rendering: for each rendered point $P(x,y,z)$ compute $\rho(x,y,z)$

- Volumetric texture mapping function/space transformed with objects
Texture Parameters

- In addition to color can control other material/object properties
  - Reflectance (either diffuse or specular)
  - Surface normal (bump mapping)
  - Transparency
  - Reflected color (environment mapping)
Environment Mapping: Cube Mapping

- 6 planar textures, sides of cube
- Point camera in 6 different directions, facing out from origin