Chapter 9: Scan Conversion (part 2) - Drawing Polygons on Raster Display

Flood Fill Algorithm

- Input
  - polygon \( P \) with rasterized edges
  - \( P = (x, y) \in P \) point inside \( P \)
- Goal: Fill interior with specified color on graphics display

Flood Fill

- \( \text{FloodFill} \) (Polygon \( P \), int \( x \), int \( y \), Color \( C \))
- if not (OnBoundary \( (x, y, P) \) or Colored \( (x, y, C) \))
  - begin
  - PlotPixel \( (x, y, C) \);
  - FloodFill \( (x, x + 1, y, C) \);
  - FloodFill \( (x, x, y + 1, C) \);
  - FloodFill \( (x, x, y - 1, C) \);
  - FloodFill \( (x, x - 1, y, C) \);
  - end;

Flood Fill - Drawbacks

- How do we find a point inside?
- Pixels visited up to 4 times to check if already set
- Need per-pixel flag indicating if set already
  - clear for every polygon!
**Modern Rasterization: Implicit Formulation**

- Triangle (convex polygon) = intersection of edge half-spaces
- Defined by set of implicit line equations

**Edge Equations**

- Multiply with denominator
  \[ L(x,y) = (y_v - y_s)(x - x_s) - (y - y_v)(x_v - x_s) = 0 \]
- Avoids singularity
- Works with vertical lines
- What about the sign?
  - Which side is in, which is out?

**Using Implicit Edge Equations**

Usage:
- Go over each pixel in bounding rectangle
- Check if pixel is inside/outside of triangle
- Use sign of edge equations

**Computing Edge Equations**

- Implicit equation of a triangle edge:
  \[ L(x,y) = \frac{(y_v - y_s)(x - x_s) - (y - y_v)(x_v - x_s)}{(x_v - x_s)^2} = 0 \]
- see Bresenham algorithm
- \( L(x,y) \) positive on one side of edge, negative on the other
- Question:
  - What happens for vertical lines?

**Edge Equations**

- Determining the sign
  - Which side is “in” and which is “out” depends on order of start/end vertices...
  - Convention: specify vertices in counter-clockwise order

**Edge Equations**

- Counter-Clockwise Triangles
  - The equation \( L(x,y) \) as specified above is negative inside, positive outside
    - Flip sign:
      \[ L(x,y) = -(y_v - y_s)(x - x_s) + (y - y_v)(x_v - x_s) = 0 \]
- Clockwise triangles
  - Use original formula
    \[ L(x,y) = (y_v - y_s)(x - x_s) - (y - y_v)(x_v - x_s) = 0 \]
Scan Conversion of Polygons

- Implicit formulation doesn’t work for non-convex polygons
- Require per pixel, per edge computation
- Observation:
  - Straight line intersection
  - with polygon = set of segments
- Alternative: algorithm based on scan-line/edge intersections
  - Works for general polygons
  - Less per pixel computations

Edge Walking

scanTrapezoid(x_L, x_R, y_B, y_T, Δx_L, Δx_R)

for (y=y_B; y<=y_T; y++) {  
  for (x=x_L; x<=x_R; x++)  
    setPixel(x,y);
    xL += Δx_L;
    xR += Δx_R;
  }

Edge Walking Triangles

- Split triangles into two “trapezoids” with continuous left and right edges

scanTrapezoid(x1, x_m, y1, 1/m1, 1/m2,)

Issues

- Many applications have small triangles
  - Setup cost is non-trivial
- Clipping triangles produces non-triangles
  - Can be avoided through re-triangulation
**Discussion**

- Old hardware:
  - Use edge-walking algorithm
  - Scan-convert edges, then fill in scanlines
  - Compute interpolated values by interpolating along edges, then scanlines
  - Requires clipping of polygons against viewing volume
  - Faster if you have a few, large polygons
  - Possibly faster in software

**Discussion:**

- Modern GPUs:
  - Use edge equations
    - Plus plane equations for attribute interpolation
    - No clipping of primitives required
  - Faster with many small triangles
  - Additional advantage:
    - Can control the order in which pixels are processed
    - Allows for more memory-coherent traversal orders
      - E.g., tiles or space-filling curve rather than scanlines

**Triangle Rasterization Issues**

- **Shared Edge Ordering**
  - Need a consistent (if arbitrary) rule
  - Example: draw pixels on left or top edge, but not on right or bottom edge

**Triangle Rasterization Issues**

- **Sliver**

**Triangle Rasterization Issues**

- **Moving Slivers**

---

**Rasterization Issues (Independent of Algorithm)**

- Exactly which pixels should be lit?
  - Those pixels inside the triangle edge (of course)
  - But what about pixels exactly on the edge?
    - Don't draw them: gaps possible between triangles
    - Draw them: order of triangles matters
These are ALIASING Problems

- Problems associated with representing continuous functions (triangles) with finite resolution (pixels)
- More on this problem when we talk about sampling...

Shading

- How can we assign pixel colors using this information?
  - Easiest: flat shading
    - Whole triangle gets one color (color of 1st vertex)
  - Better: Gouraud shading
    - Linearly interpolate color across triangle
  - Even better: Phong shading
    - Linearly interpolate the normal vector
    - Compute lighting for every pixel
    - Note: not supported by rendering pipeline as discussed so far

Flat Shading

- Simplest approach: calculate illumination at one point per polygon (e.g. center)

- Obviously inaccurate for smooth surfaces

Shading Approximations

- If an object really is faceted, is this accurate?

Flat Shading Approximations

- If an object really is faceted, is this accurate?
Flat Shading Approximations

- If an object really is faceted, is this accurate?
  - no!
    - For point sources, direction to light varies across the facet
    - For specular reflectance, direction to eye varies across the facet

Gouraud Shading Artifacts

- Often appears dull, chalky
- Lacks accurate specular component
  - if included, will be averaged over entire polygon

Improving Flat Shading

- What if we evaluate Phong lighting model at each pixel of the polygon?
  - Better, but result still clearly faceted
- Gouraud Shading: For smoother-looking surfaces introduce vertex normals at each vertex
  - Usually different from facet normal
  - Used only for shading
  - Think of as a better approximation of the real surface that the polygons approximate

Vertex Normals

- Vertex normals may be
  - Provided with the model
  - Computed from first principles
  - Approximated by averaging the normals of the facets that share the vertex

Phong Shading

- linearly interpolating surface normal across the facet, applying Phong lighting model at every pixel
  - Same input as Gouraud shading
  - Pro: much smoother results
  - Con: considerably more expensive

  - Not the same as Phong lighting
    - Common confusion
    - Phong lighting: empirical model to calculate illumination at a point on a surface
**Phong Shading**
- Linearly interpolate the vertex normals
- Compute lighting equations at each pixel
- Can use specular component

\[ I_{\text{total}} = k_a I_{\text{ambient}} + \sum_{i=1}^{n \text{ lights}} \left( k_d (n \cdot l_i) + k_s (v \cdot r_i)^s\right) \]

*remember: normals used in diffuse and specular terms*

**Phong Shading Difficulties**
- Computationally expensive
- Per-pixel vector normalization and lighting computation!
- Floating point operations required
- Lighting after perspective projection
- Messes up the angles between vectors
- Have to keep eye-space vectors around
- No direct support in standard rendering pipeline
  - But can be simulated with texture mapping, procedural shading hardware (see later)

**Shading Artifacts: Silhouettes**
- Polygonal silhouettes remain

**Interpolation - access triangle interior**
- Interpolate between vertices:
  - \( z \)
  - \( r,g,b \) - colour components
  - \( u,v \) - texture coordinates
  - \( N_1, N_2, N_3 \) - surface normals

**Barycentric Coordinates**
- Area
  \[ A = \frac{1}{2} \left\| P_1P_2 \times P_1P_3 \right\| \]
  - Barycentric coordinates
  \[ a_1 = A_{P_1P_2P_3} / A, a_2 = A_{P_1P_3P} / A, \]
  \[ a_3 = A_{P_1P_2P} / A, \]
  \[ P = a_1 P_1 + a_2 P_2 + a_3 P_3 \]

**Barycentric Coordinates**
- Weighted combination of vertices
  \[ P = a_1 \cdot P_1 + a_2 \cdot P_2 + a_3 \cdot P_3 \]
  \[ a_1 + a_2 + a_3 = 1 \]
  \[ 0 \leq a_i, a_j, a_k \leq 1 \]
Alternative formula: Bi-Linear Interpolation

- Interpolate quantity along L and R edges
- (as a function of y)
- Then interpolate quantity as a function of x

**Observation:** Values vary linearly in image plane

- E.g.: \( r = Ax + By + C \)
- \( r \) = red channel of the color
- Same for \( g, b, Nx, Ny, Nz, z \... \)

From info at vertices we know:

\[
\begin{align*}
    r_1 &= Ax_1 + By_1 + C \\
    r_2 &= Ax_2 + By_2 + C \\
    r_3 &= Ax_3 + By_3 + C
\end{align*}
\]

- Solve for \( A, B, C \)
- One-time set-up cost per triangle & interpolated value

Discussion

- Which algorithm (formula) to use when?
  - Bi-linear interpolation
    - Together with trapezoid scan conversion
  - Plane equations
    - Together with implicit (edge equation) scan conversion
  - Barycentric coordinates
    - Too expensive in current context
    - But: method of choice for ray-tracing
      - Whenever you only need to compute the value for a single pixel

Bi-Linear Interpolation

- Most common approach, and what OpenGL does
  - Perform Phong lighting at the vertices
  - Linearly interpolate the resulting colors over faces
    - Along edges
    - Along scanlines
- Equivalent to Barycentric Coordinates!
  - interior: mix of \( c_1, c_2, c_3 \)
  - edge: mix of \( c_2, c_3 \)

Bi-Linear interpolation

- Formulation

\[
P = \frac{c_2}{c_1 + c_2} P_2 + \frac{c_1}{c_1 + c_2} P_1
\]

\[
P_i = \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_1
\]

\[
P_s = \frac{b_1 + b_2}{b_1 + b_2} P_s + \frac{b_1}{b_1 + b_2} P_1
\]

\[
P = \frac{c_2}{c_1 + c_2} \left( \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_1 \right) + \frac{c_1}{c_1 + c_2} \left( \frac{b_1}{b_1 + b_2} P_s + \frac{b_1}{b_1 + b_2} P_1 \right)
\]

Validation

- All formulations should provide same value
- Can verify barycentric properties

\[
a_1 + a_2 + a_3 = 1
\]

\[
0 \leq a_1, a_2, a_3 \leq 1
\]