Chapter 9

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

Rendering Pipeline
Triangle/Polygon Rasterization

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

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 (P, x + 1, y, C);
    FloodFill (P, x, y + 1, C);
    FloodFill (P, x, y - 1, C);
    FloodFill (P, 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

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:

\[ L(x, y) = \frac{(y_e - y_s)(x_t - x_e)}{(x_e - x_t)} (x - x_e) - (y - y_e) = 0 \]

- see Bresenham algorithm
- \( L(x,y) \) positive on one side of edge, negative on the other

- Question:
  - What happens for vertical lines?

Edge Equations

- Multiply with denominator

\[ L(x,y) = (y_e - y_s)(x - x_e) - (y - y_e)(x_e - x_s) = 0 \]

- Avoids singularity
- Works with vertical lines

- What about the sign?
  - Which side is in, which is out?
**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

- Counter-Clockwise Triangles
  - The equation \( L(x, y) \) as specified above is *negative inside, positive outside*
    - Flip sign:
      \[
      L(x, y) = -(y_e - y_s)(x - x_s) + (y - y_s)(x_e - x_s) = 0
      \]
  - *Clockwise triangles*
    - Use original formula
      \[
      L(x, y) = (y_e - y_s)(x - x_s) - (y - y_s)(x_e - x_s) = 0
      \]
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

General Algorithm
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity to determine in/out
- Fill the ‘in’ pixels
- Efficiency improvement:
  - Exploit row-to-row coherence using “edge table”
Scan Conversion - Polygons

Edge Walking

- Special case: Scan-converting a trapezoid
  - Exploit continuous L and R edges
    - Predict intersections from one line to next

\[ \text{scanTrapezoid}(x_L, x_R, y_B, y_T, \Delta x_L, \Delta x_R) \]

for (y = yB; y <= yT; y++) {
  for (x = xL; x <= xR; x++) {
    setPixel(x, y);
    xL += DxL;
    xR += DxR;
  }
}

Copyright 2011. Alla Sheffer, UBC
**Edge Walking Triangles**

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

\[
\text{scanTrapezoid}( x_3, x_m, y_3, y_1, \frac{1}{m_1}, \frac{1}{m_2})
\]

\[
\text{scanTrapezoid}( x_2, x_3, y_2, y_3, \frac{1}{m_2}, \frac{1}{m_1})
\]

**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
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

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
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

Assigning colors inside triangle interior
Shading

Input to Scan Conversion:
- Vertices of triangles (lines, quadrilaterals...)
- Color (per vertex)
  - Specified with glColor
  - Or: computed with lighting
- World-space normal (per vertex)
  - Left over from lighting stage

Shading Task:
- Determine color of every pixel in the triangle

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

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

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

Gouraud Shading Artifacts

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

*this interior shading missed!
*this vertex shading spread over too much area
Gouraud Shading Artifacts

- Mach bands
  - Eye enhances discontinuity in first derivative
  - Very disturbing, especially for highlights

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

\[
I_{total} = k_a I_{ambient} + \sum_{i=1}^{\#\text{lights}} I_i \left( k_d (\mathbf{n} \cdot \mathbf{l}_i) + k_s (\mathbf{v} \cdot \mathbf{r}_i)^{n_{\text{shiny}}} \right)
\]

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

Interpolate between vertices:
- z
- r, g, b - colour components
- u, v - texture coordinates
- N_x, N_y, N_z - surface normals

Equivalent
- Barycentric coordinates
- Bilinear interpolation
- Plane Interpolation
Barycentric Coordinates

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

- Barycentric coordinates
  \[ a_1 = \frac{A_{P_3P_2P}}{A}, \quad a_2 = \frac{A_{P_3P_1P}}{A}, \quad a_3 = \frac{A_{P_2P_1P}}{A} \]
  \[ P = a_1P_1 + a_2P_2 + a_3P_3 \]

Barycentric Coordinates

- Weighted combination of vertices
  \[ P = a_1 \cdot P_1 + a_2 \cdot P_2 + a_3 \cdot P_3 \]
  \[ a_1 + a_2 + a_3 = 1 \]
  \[ 0 \leq a_1, a_2, a_3 \leq 1 \]
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

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!
**Bi-Linear interpolation**

Formulation

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

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

\[ P_k = \frac{b_2}{b_1 + b_2} P_2 + \frac{b_1}{b_1 + b_2} P_1 \]

\[ P = \frac{c_2}{c_1 + c_2} \left( \frac{d_2}{d_1 + d_2} P_2 + \frac{d_1}{d_1 + d_2} P_3 \right) + \frac{c_1}{c_1 + c_2} \left( \frac{b_2}{b_1 + b_2} P_2 + \frac{b_1}{b_1 + b_2} P_1 \right) \]

---

**Another Alternative:**

**Plane Equation**

- 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:
    - \( r_1 = Ax_1 + By_1 + C \)
    - \( r_2 = Ax_2 + By_2 + C \)
    - \( r_3 = Ax_3 + By_3 + C \)
  - 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

Validation

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

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