Chapter 9

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

Triangle/Polygon Rasterization

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

Using Implicit Edge Equations

Usage:
- Go over each pixel on screen
- To be efficient restrict to bounding rectangle
- Check if pixel is inside/outside of triangle
- Use sign of edge equations

Implicit Formulation

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

Computing Edge Equations

- Implicit equation of a triangle edge:
  \[ L(x, y) = \frac{(x_2 - y_1)(x - x_1) - (y - y_1)(x_1 - x_2)}{x_1 - x_2} = 0 \]
  - see Bresenham algorithm
  - \( L(x, y) \) positive on one side of edge, negative on the other
- Question:
  - What happens for vertical lines?
Edge Equations

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

Edge Equations

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

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

Scan Conversion of Polygons

- 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”

Edge Walking

- Special case: Scan-converting a trapezoid
  - Exploit continuous \( L \) and \( R \) edges
    - Predict intersections from one line to next
  - \( \text{scanTrapezoid}(x_L, x_R, y_L, y_U, \Delta x_L, \Delta x_R) \)
Scan Conversion - Polygons

**Edge Walking**

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

**Edge Walking Triangles**

```plaintext
split triangles into two “trapezoids”
with continuous left and right edges
```

**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
  - 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

- **Moving Slivers**

**Triangle Rasterization Issues**

- **Sliver**

**Triangle Rasterization Issues**

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

**Values in the interior**

- Barycentric coordinates

**Interpolation - access triangle interior**

- 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**
  \[ A = \frac{1}{2} |P_1P_2 \times P_3P_2|\]
- Barycentric coordinates
  \[ a_1 = A_{P_1P_2P} / A, \quad a_2 = A_{P_2P_3P} / A, \quad a_3 = A_{P_3P_1P} / A \]
  \[ P = a_1P_1 + a_2P_2 + a_3P_3 \]

**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

**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

---

**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 \( c_1, c_3 \)

**Bi-Linear Interpolation**

- **Formulation**
  \[ P = \frac{c_1}{c_1 + c_2} \cdot P_1 + \frac{c_1}{c_1 + c_2} \cdot P_3 \]
  \[ P = \frac{d_1}{d_1 + d_2} \cdot P_1 + \frac{d_1}{d_1 + d_2} \cdot P_3 \]
  \[ P = \frac{b_1}{b_1 + b_2} \cdot P_1 + \frac{b_1}{b_1 + b_2} \cdot P_3 \]

**Barycentric Coordinates**

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

Shading

Input to Scan Conversion:
- Vertices of triangles (lines, quadrilaterals...)
- Color (per vertex)
  - Specified with \texttt{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

Flat Shading

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

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

Validation

- All formulations should provide same value
- Can verify barycentric properties
  \[ a_1 + a_2 + a_3 = 1 \]
  \[ 0 \leq a_1, a_2, a_3 \leq 1 \]
**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

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

---

Copyright 2013. Alla Sheffer, UBC
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 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)

Phong Shading

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

Remember: normals used in diffuse and specular terms

Discontinuity in normal's rate of change harder to detect

Shading Artifacts: Silhouettes
- Polygonal silhouettes remain