Scan Conversion & Shading

Overview

- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel

Scan Conversion

- Render an image of a geometric primitive by setting pixel colors
  - Example: Filling the inside of a triangle

Triangle Scan Conversion

- Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Anti-aliased edges
  - No cracks between adjacent primitives
  - MUST BE FAST!
**Triangle Scan Conversion**

- Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Antialiased edges
  - No cracks between adjacent primitives
  - MUST BE FAST!

**Simple Algorithm**

- Color all pixels inside triangle

```c
void ScanTriangle(Triangle T, Color rgba) {
  for each pixel P at (x,y) {
    if (Inside(T, P)) {
      SetPixel(x, y, rgba);
    }
  }
}
```

**Inside Triangle Test**

- A point is inside a triangle if it is in the positive halfspace of all three boundary lines
  - Triangle vertices are ordered counter-clockwise
  - Point must be on the left side of every boundary line

```c
Boolean Inside(Triangle T, Point P) {
  for each boundary line L of T {
    Scalar d = L.a*P.x + L.b*P.y + L.c;
    if (d < 0.0) return FALSE;
  }
  return TRUE;
}
```

**Simple Algorithm**

- What is bad about this algorithm?

```c
void ScanTriangle(Triangle T, Color rgba) {
  for each pixel P at (x,y) {
    if (Inside(T, P)) {
      SetPixel(x, y, rgba);
    }
  }
}
```

**Triangle Sweep-Line Algorithm**

- Take advantage of spatial coherence
  - Compute which pixels are inside using horizontal spans
  - Process horizontal spans in scan-line order
- Take advantage of edge linearity
  - Use edge slopes to update coordinates incrementally
void ScanTriangle(Triangle T, Color rgba){
    for each edge pair {
        initialize xL, xR;
        compute dxL/dyL and dxR/dyR;
        for each scanline at y
            for (i = xL; x <= xR; x++)
                SetPixel(x, y, rgba);
        xL += dxL/dyL;
        xR += dxR/dyR;
    }
}

• Fill pixels inside a polygon
  ○ Triangle
  ○ Quadrilateral
  ○ Convex
  ○ Star-shaped
  ○ Concave
  ○ Self-intersecting
  ○ Holes

What problems do we encounter with arbitrary polygons?

• Need better test for points inside polygon
  ○ Triangle method works only for convex polygons

Convex Polygon
Concave Polygon

• What is a good rule for which pixels are inside?

Inside Polygon Rule
• Odd-parity rule
  ○ Any ray from P to infinity crosses odd number of edges

Concave
Self-Intersecting
With Holes

• Incremental algorithm to find spans, and determine insideness with odd parity rule
  ○ Takes advantage of scanline coherence

Triangle
Polygon
**Polygon Sweep-Line Algorithm**

```c
void ScanPolygon(Triangle T, Color rgba){
    sort edges by maxy
    make empty "active edge list"
    for each scanline (top-to-bottom) {
        insert/remove edges from "active edge list"
        update x coordinate of every active edge
        sort active edges by x coordinate
        for each pair of active edges (left-to-right)
            SetPixels(xi, xi+1, y, rgba);
    }
}
```

**Hardware Scan Conversion**

- Convert everything into triangles
  - Scan convert the triangles

**Hardware Antialiasing**

- Supersample pixels
  - Multiple samples per pixel
  - Average subpixel intensities (box filter)
  - Trades intensity resolution for spatial resolution

**Overview**

- Scan conversion
  - Figure out which pixels to fill
- Shading
  - Determine a color for each filled pixel

**Shading**

- How do we choose a color for each filled pixel?
  - Each illumination calculation for a ray from the eyepoint through the view plane provides a radiance sample
    - How do we choose where to place samples?
    - How do we filter samples to reconstruct image?

**Ray Casting**

- Simplest shading approach is to perform independent lighting calculation for every pixel
  - When is this unnecessary?

\[
I = I_e + K_s I_{sc} + \sum (K_d (N \cdot L_i) I_s + K_a (V \cdot R_i)^\gamma I_s)
\]
**Polygon Shading**

- Can take advantage of spatial coherence
  - Illumination calculations for pixels covered by same primitive are related to each other

\[ I = I_e + K_d I_d + \sum_i (K_r (N \cdot L) I_i + K_g (V \cdot R)^l I_i) \]

**Polygon Shading Algorithms**

- Flat Shading
- Gouraud Shading
- Phong Shading

**Polyhedron Shading Algorithms**

- Flat Shading
- Gouraud Shading
- Phong Shading

**Flat Shading**

- What if a faceted object is illuminated only by directional light sources and is either diffuse or viewed from infinitely far away

\[ I = I_e + K_d I_d + \sum_i (K_r (N \cdot L) I_i + K_g (V \cdot R)^l I_i) \]

**Flat Shading**

- One illumination calculation per polygon
  - Assign all pixels inside each polygon the same color
  - Objects look like they are composed of polygons
    - OK for polyhedral objects
    - Not so good for ones with smooth surfaces
Polygon Shading Algorithms

• Flat Shading
• Gouraud Shading
• Phong Shading

Gouraud Shading

• What if smooth surface is represented by polygonal mesh with a normal at each vertex?

\[ I = I_0 + K_d I_{dL} + \sum (K_g (N \cdot L_i) I_i + K_r (V \cdot R_i)^d I_i) \]

Gouraud Shading

• Method 1: One lighting calculation per vertex
  o Assign pixels inside polygon by interpolating colors computed at vertices

Gouraud Shading

• Bilinearly interpolate colors at vertices down and across scan lines

\[ A = \alpha I_1 + (1 - \alpha) I_3 \]
\[ B = \beta I_2 + (1 - \beta) I_3 \]
\[ I = \alpha A + (1 - \alpha)B \]

Gouraud Shading

• Smooth shading over adjacent polygons
  o Curved surfaces
  o Illumination highlights
  o Soft shadows

Gouraud Shading

• Produces smoothly shaded polygonal mesh
  o Piecewise linear approximation
  o Need fine mesh to capture subtle lighting effects

Mesh with shared normals at vertices
Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- Phong Shading

Phong Shading

- What if polygonal mesh is too coarse to capture illumination effects in polygon interiors?

\[ I = I_e + K_e I_{d, e} + \sum (K_{le}(N \cdot L_i)I_i + K_{se}(V \cdot R_i)I_i) \]

Phong Shading

- Method 2: One lighting calculation per pixel
  - Approximate surface normals for points inside polygons by bilinear interpolation of normals from vertices

Phong Shading

- Bilinearly interpolate surface normals at vertices down and across scan lines

Shading Issues

- Problems with interpolated shading:
  - Polygonal silhouettes
  - Perspective distortion
  - Orientation dependence (due to bilinear interpolation)
  - Problems at T-vertices
  - Problems computing shared vertex normals
Summary

• 2D polygon scan conversion
  o Paint pixels inside primitive
  o Sweep-line algorithm for polygons

• Polygon Shading Algorithms
  o Flat
  o Gouraud
  o Phong
  o Ray casting

• Key ideas:
  o Sampling and reconstruction
  o Spatial coherence

Less expensive
More accurate