Scan Conversion & Shading

Courtesy of Adam Finkelstein, Princeton University
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Scan Conversion & Shading

3D Modeling Coordinates

3D World Coordinates

3D World Coordinates

3D Camera Coordinates

2D Screen Coordinates

2D Screen Coordinates

2D Image Coordinates

2D Image Coordinates

Image
Overview

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel
Rasterization

- After projection, we work in a 2-D space again
- Screen coordinate representation
- Generate a set of pixels through rasterization or scan conversion
- OpenGL allows us to manipulate pixels directly, need not worry about writing to frame buffer directly
Scan-Line Algorithm

- Most efficient method when memory was limited
- Move from left to right
- Draw polygons as we come to them
- For intersecting polygons determine $z$ value
- Comparison with z-Buffer: 
  *One scan line at a time not one polygon at a time*
Scan Conversion

- Rasterization of primitives
- Assume frame buffer is an $n \times m$ array of pixels, $(0,0)$ in lower-left corner
  - write_pixel(int ix, int iy, int value);
- Frame buffers are inherently discrete
- Screen coordinates are real numbers
Differential Digital Analyzer

FIG. 1. Vannevar Bush shown with the M.I.T. differential analyzer.
DDA Algorithm

- Simplest scan-conversion algorithm
- Digital Differential Analyzer – an electromechanical device to simulate differential equations
- Simple differential equation: \( dy/dx = m \)

\[
m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} \quad \text{where} \quad 0 \leq m \leq 1
\]
DDA Algorithm
for(x=x1, ix <= x2, x++)
{
    y+=m;
    write_pixel(x, round(y), line_color);
}

DDA Algorithm
DDA Algorithm

Slope greater than 1

Swap the roles of x and y
Bresenham’s Algorithm

\[ y = mx + h \]

\[ j + \frac{3}{2} \]

\[ j + \frac{1}{2} \]

\[ j2 \]

\[ i + \frac{1}{2} \]

\[ i + \frac{3}{2} \]
Bresenham’s Algorithm

- **Decision variable**
  - $d = a - b$
  - $d > 0$ then lower pixel,
    otherwise upper pixel
Flood Fill

- Start off with Bresenham’s algorithm for the edges
- Find a seed point \((x,y)\)

```
flood_fill(int x, int y)
{
    if(read_pixel(x,y)==WHITE)
    {
        write_pixel(x,y,BLACK);
        flood_fill(x-1,y);
        flood_fill(x+1,y);
        flood_fill(x,y-1);
        flood_fill(x,y+1);
    }
}
```
Scan Conversion

• Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel(int x, int y, Color rgba)
```

• Example: Filling the inside of a triangle
Scan Conversion

• Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel(int x, int y, Color rgba)
```

• Example: Filling the inside of a triangle
Triangle Scan Conversion

- Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Antialiased 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);
    }
}
```
Line defines two halfspaces

- Implicit equation for a line
  - On line: $ax + by + c = 0$
  - On right: $ax + by + c < 0$
  - On left: $ax + by + c > 0$
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
Inside Triangle Test

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
t 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
Triangle Sweep-Line Algorithm

```c
void ScanTriangle(Triangle T, Color rgba){
    for each edge pair {
        initialize x_L, x_R;
        compute dx_L/dy_L and dx_R/dy_R;
        for each scanline at y
            for (int x = x_L; x <= x_R; x++)
                SetPixel(x, y, rgba);
            x_L += dx_L/dy_L;
            x_R += dx_R/dy_R;
    }
}
```

Bresenham’s algorithm works the same way, but uses only integer operations!
Polygon Scan Conversion

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

What problems do we encounter with arbitrary polygons?
Polygon Scan Conversion

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

Convex Polygon

Concave Polygon
Inside Polygon Rule

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

Concave  Self-Intersecting  With Holes
Inside Polygon Rule

- Odd-parity rule
  - Any ray from P to infinity crosses odd number of edges

Concave  Self-Intersecting  With Holes
Polygon Sweep-Line Algorithm

• Incremental algorithm to find spans, and determine insideness with odd parity rule
  ◦ Takes advantage of scanline coherence
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($x_i, x_{i+1}, y, rgba$);
    }
}
Hardware Scan Conversion

• Convert everything into triangles
  ◦ **Scan convert the triangles**
Display

- All is not perfect on the screen
- Numerous quality problems
- Image oriented (pixels) vs. object oriented rendering
- Jaggedness on raster displays
- Introduce aliasing
Antialiasing

• Aliasing – an attempt to go from the continuous representation of an object, which has infinite resolution, to a discrete approximation, which has limited resolution
Antialiasing
Antialiasing

- Mathematical lines are one-dimensional entities that have length but no width.
- Rasterized lines must have width in order to be visible.
Antialiasing

- Shade each box by the percentage of the ideal line that crosses it
- Antialiasing by area averaging
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?

Emphasis on methods that can be implemented in hardware

Angel Figure 6.34
Ray Casting

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

\[
I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i)
\]
Polygon Shading

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

\[
I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^{n} I_i)
\]
Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- Phong Shading
Polygon 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

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
Flat Shading

• One illumination calculation per polygon
  ◦ Assign all pixels inside each polygon the same color
Flat Shading

• Objects look like they are composed of polygons
  ◦ OK for polyhedral objects
  ◦ Not so good for 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?

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
Gouraud Shading

- Method 1: One lighting calculation per vertex
  - Assign pixels inside polygon by interpolating colors computed at vertices
Gouraud Shading

- Bilinearly interpolate colors at vertices down and across scan lines

\[ A = \alpha l_1 + (1-\alpha) l_3 \]

\[ B = \beta l_2 + (1-\beta) l_3 \]

\[ I = \phi A + (1-\phi) B \]
Gouraud Shading

- Smooth shading over adjacent polygons
  - Curved surfaces
  - Illumination highlights
  - Soft shadows

Mesh with shared normals at vertices
Gouraud Shading

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

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
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

\[ A = \alpha N_1 + (1-\alpha)N_3 \]
\[ B = \beta N_2 + (1-\beta)N_3 \]
\[ I = \phi A + (1-\phi)B \]
Polygon Shading Algorithms

Wireframe  Flat

Gouraud  Phong

Watt Plate 7
Shading Issues

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