Scan Conversion

CS5600 Computer Graphics

Spring 2013

Review

• Line rasterization
  – Basic Incremental Algorithm
  – Digital Differential Analyzer
    • Rather than solve line equation at each pixel, use evaluation of line from previous pixel and slope to approximate line equation
  – Bresenham
    • Use integer arithmetic and midpoint discriminator to test between two possible pixels (over vs. over-and-up)

Rasterizing Polygons

• In interactive graphics, polygons rule the world
  • Two main reasons:
    – Lowest common denominator for surfaces
      • Can represent any surface with arbitrary accuracy
      • Splines, mathematical functions, volumetric isosurfaces...
    – Mathematical simplicity lends itself to simple, regular rendering algorithms
      • Like those we’re about to discuss...
      • Such algorithms embed well in hardware

Rasterizing Polygons

• Triangle is the minimal unit of a polygon
  • All polygons can be broken up into triangles
    – Convex, concave, complex
  • Triangles are guaranteed to be:
    – Planar
    • Convex
  – What exactly does it mean to be convex?

Convex Shapes

• A two-dimensional shape is convex if and only if every line segment connecting two points on the boundary is entirely contained.

Triangularization

• Convex polygons easily triangulated
  • Concave polygons present a challenge

Even hippos are made of polygons!
Rasterizing Triangles

- Interactive graphics hardware sometimes uses edge walking or edge equation techniques for rasterizing triangles.
- Interactive graphics hardware more commonly uses barycentric coordinates for rasterizing triangles.

Scan Conversion

- In scanline rendering surfaces are projected on the screen and space filling 'rasterizing' algorithms are used to fill in the color.
- Color values from light are approximated.

Triangle Rasterization Issues

- Exactly which pixels should be lit?
- A: Those pixels inside the triangle edges
- What about pixels exactly on the edge?
  - Draw them: order of triangles matters (it shouldn't)
  - Don't draw them: gaps possible between triangles
- We need a consistent (if arbitrary) rule
  - Example: draw pixels on left and bottom edge, but not on right or top edge

Triangle Rasterization Issues

- Sliver

Triangle Rasterization Issues

- Moving Slivers

Triangle Rasterization Issues

- Shared Edge Ordering
### 2D Polygon
- Area “inside” a sequence of points
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting
  - Holes

**Notes:**
- Points are in counter-clockwise order

### 2D Rendering
- Create an image from a set of 2D geometric primitives

### Scan Conversion
- Render an image of a geometric primitive by setting pixel colors
  ```c
  void setPixel(int x, int y, Color rgb)
  ```
- 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!

### Simple Algorithm
- Color all pixels inside triangle
  ```c
  void ScanTriangle(Triangle T, Color rgb)
  for each pixel F at (x,y)
  if (inside(T, F))
      SetPixel(x, y, rgb);
  }
  ```
How do we know if it’s ‘inside’?

**Edge Equations**

- An edge equation is simply the equation of the line defining that edge
  
  - Q: *What is the implicit equation of a line?*
  - A: $Ax + By + C = 0$
  
  - Q: *Given a point $(x, y)$, what does plugging $x$ & $y$ into this equation tell us?*
  - A: Whether the point is:
    - On the line: $Ax + By + C = 0$
    - "Above" the line: $Ax + By + C > 0$
    - "Below" the line: $Ax + By + C < 0$

**Edge Equations**

- Edge equations thus define two *half-spaces*:
  
  - $Ax + By + C > 0$
  - $Ax + By + C = 0$
  - $Ax + By + C < 0$

**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

---

**Simple Algorithm**

- Color all pixels inside triangle

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

How do we know if it’s ‘inside’?
Sweep-line

• Basic idea:
  – Draw edges vertically
  – Interpolate colors up/down edges
  – Fill in horizontal spans for each scanline
  – At each scanline, interpolate edge colors across span

Sweep-line: Notes

• Order three triangle vertices in x and y
  – Find middle point in y dimension and compute if it is to the left or right of polygon. Also could be flat top or flat bottom triangle
• We know where left and right edges are.
  – Proceed from top scanline downwards (and other way too)
  – Fill each span
  – Until bottom/top vertex is reached
• Advantage: can be made very fast
• Disadvantages:
  – Lots of finicky special cases

Sweep line: Disadvantages

• Fractional offsets:
  – Be careful when interpolating color values!
  – Beware of gaps between adjacent edges
  – Beware of duplicating shared edges

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

void scanTriangle(Triangle T, Color rgbal)
    for each edge pair (e1, e2)
        initialize \( x_1, y_1 \)
        compute \( dx_1, dy_1 \) and \( dx_2, dy_2 \)
        for each scanline at \( y \)
            for (\( x \) = \( x_1 \); \( x < x_2 \); \( x++ \))
                setpixel(\( x \), \( y \), rgbal);
        \( x_2 \) = \( dx_1(dy_2 - dy_1)/dx_1 \);
    }

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 Scan Conversion

- Intersection Points
- Other points in the span
Determining Inside vs. Outside

- Use the odd-parity rule
  - Set parity even initially
  - Invert parity at each intersection point
  - Draw pixels when parity is odd, do not draw when it is even
- How do we count vertices, i.e., do we invert parity when a vertex falls exactly on a scan line?

Vertices and Parity

- How do we count the intersecting vertex in the parity computation?

Vertices and Parity

- We need to either count it 0 times, or 2 times to keep parity correct.
- What about:
  - We need to count this vertex once

Vertices and Parity

- If we count a vertex as one intersection, the second polygon gets drawn correctly, but the first does not.
- If we count a vertex as zero or two intersections, the first polygon gets drawn correctly, but the second does not.
- How do we handle this?
  - Count only vertices that are the ymin vertex for that line

Vertices and Parity

- Both cases now work correctly

Horizontal Edges

- How do we deal with horizontal edges?

Don’t count their vertices in the parity calculation!
Top Spans of Polygons

• Effect of only counting $y_{	ext{min}}$:
  – Top spans of polygons are not drawn
  – If two polygons share this edge, it is not a problem.
  – What about if this is the only polygon with that edge?

Shared Polygon Edges

• What if two polygons share an edge?
  – Solution:
    – Span is closed on left and open on right ($x_{	ext{min}} < x < x_{	ext{max}}$)
    – Scan lines closed on bottom and open on top ($y_{	ext{min}} < y < y_{	ext{max}}$)

General Pixel Ownership Rule

• Half-plane rule:
  A boundary pixel (whose center falls exactly on an edge) is not considered part of a primitive if the half plane formed by the edge and containing the primitive lies to the left or below the edge.

• Consequences:
  – Spans are missing the right-most pixel
  – Each polygon is missing its top-most span

General Polygon Rasterization

• Consider the following polygon:

  How do we know whether a given pixel on the scanline is inside or outside the polygon?

Polygon Rasterization

• Inside-Outide Points

  Scan line
General Polygon Rasterization

- Basic idea: use a parity test
  
  ```
  edgeCnt = 0;
  for each pixel on scanline (l to r)
    if (oldpixel->newpixel crosses edge)
      edgeCnt ++;
  // draw the pixel if edgeCnt odd
  if (edgeCnt % 2)
    setPixel(pixel);
  ```

General Polygon Rasterization

- Count your vertices carefully

Faster Polygon Rasterization

- How can we optimize the code?
  
  ```
  edgeCnt = 0;
  for each pixel on scanline (l to r)
    if (oldpixel->newpixel crosses edge)
      edgeCnt ++;
  // draw the pixel if edgeCnt odd
  if (edgeCnt % 2)
    setPixel(pixel);
  ```

- Big cost: testing pixels against each edge
- Solution: active edge table (AET)

Active Edge Table

- Idea:
  - Edges intersecting a given scanline are likely to intersect the next scanline
  - The order of edge intersections doesn't change much from scanline to scanline
**Active Edge Table**

Preprocess: Sort on Y

<table>
<thead>
<tr>
<th>Edge Table</th>
<th>$Y_{max}$, $x_{at\ y\ min}$, slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB:</td>
<td>3 7       -5/2</td>
</tr>
<tr>
<td>CB:</td>
<td>5 7       6/4</td>
</tr>
<tr>
<td>CD:</td>
<td>11 13     0</td>
</tr>
<tr>
<td>DE:</td>
<td>11 7      6/4</td>
</tr>
<tr>
<td>EF:</td>
<td>9 7       -5/2</td>
</tr>
<tr>
<td>FA:</td>
<td></td>
</tr>
</tbody>
</table>

**Active Edge Table**

Preprocess: Sort on Y

<table>
<thead>
<tr>
<th>Edge Table</th>
<th>$Y_{max}$, $x_{at\ y\ min}$, slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB:</td>
<td>3 7       -5/2</td>
</tr>
<tr>
<td>CB:</td>
<td>5 7       6/4</td>
</tr>
<tr>
<td>CD:</td>
<td>11 13     0</td>
</tr>
<tr>
<td>DE:</td>
<td>11 7      6/4</td>
</tr>
<tr>
<td>EF:</td>
<td>9 7       -5/2</td>
</tr>
<tr>
<td>FA:</td>
<td></td>
</tr>
</tbody>
</table>
Active Edge Table

Preprocess: Sort on Y

Edge Table

\[
Y_{\text{max}}, x_{\text{at } y_{\text{min}}}, \text{slope}
\]

<table>
<thead>
<tr>
<th>Edge</th>
<th>Y_{\text{max}}</th>
<th>x_{\text{at } y_{\text{min}}}</th>
<th>Slope</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>3</td>
<td>7</td>
<td>-5/2</td>
</tr>
<tr>
<td>CB</td>
<td>5</td>
<td>7</td>
<td>6/4</td>
</tr>
<tr>
<td>CD</td>
<td>11</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>DE</td>
<td>11</td>
<td>7</td>
<td>6/4</td>
</tr>
<tr>
<td>EF</td>
<td>9</td>
<td>7</td>
<td>-5/2</td>
</tr>
<tr>
<td>FA</td>
<td>9</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Active Edge Table

Preprocess: Sort on Y

Active Edge Table Example

Example of an AET containing edges (FA, EF, DE, CD) on scan line 8:

1. : \( y = 8 \) Get edges from ET bucket \( y \) (none in this case, \( y = 8 \) has no entry)
2. : Remove from the AET any entries where \( y_{\text{max}} < y \) (none here)
3. : sort by \( x \)
4. : Draw scan line. To handle multiple edges, group in pairs: \( \text{FA,EF} \), \( \text{DE,CD} \)
5. : \( y = 9 \), y_{\text{max}} = 9
6. : Update \( x \) for non-vertical edges, as in simple line drawing.

Active Edge Table

• Algorithm: scanline from bottom to top...
  • Sort all edges by their minimum y coord (last slide)
  • Starting at smallest y coord with in entry in edge table
  • For each scanline:
    • Add edges with \( y_{\text{min}} = y \) (move edges in edge table to AET)
    • Retire edges with \( y_{\text{max}} < y \) (completed edges)
    • Sort edges in AET by x intersection
    • Walk from left to right, setting pixels by parity rule
    • Increment scanline
    • Recalculate edge intersections (how?)
    • Stop when \( y > y_{\text{max}} \) for edge table and AET is empty

Active Edge Table

• Algorithm: scanline from bottom to top...
  • Sort all edges by their minimum y coord (last slide)
  • Starting at smallest y coord with in entry in edge table
  • For each scanline:
    1. Add edges with \( y_{\text{min}} = y \) (move edges in edge table to AET)
    2. Retire edges with \( y_{\text{max}} < y \) (completed edges)
    3. Sort edges in AET by x intersection
    4. Walk from left to right, setting pixels by parity rule
    5. Increment scanline
    6. Recalculate edge intersections (how?)

For every non-vertical edge in the AET update \( x \) for the new \( y \) (calculate the next intersection of the edge with the scan line).

• Stop when \( y > y_{\text{max}} \) for edge table and AET is empty
Active Edge Table Example (cont.)

1. \( y = 9 \) Get edges from ET bucket \( y \) (none in this case, \( y = 9 \) has no entry in ET)
   - “Scan line 9” shown in fig 3.28 below
2. Remove from the AET any entries with ymax = \( y \) (remove FA, EF)
3. Sort by \( x \)
4. Draw scan line between (DE, CD)
5. \( y = y + 1 = 10 \)
6. Update x in (DE, CD)
7. (\( y = 10 \)) (Scan line 10 shown in fig 3.28 below)
8. And so on…

(FvDFH pages 92, 99)

Why are Barycentric coordinates useful?

- For any point \( x \), if the barycentric representation of that point: \( x = \alpha x_a + \beta x_b + \gamma x_c \), \( x < 1 \)
- Also \( \alpha, \beta, \gamma \) can be used as a mass function across the surface of a triangle to be used for interpolation.
- This is used to interpolate normals across the surface of a triangle to make polygon surfaces look rounder.

Triangles (cont.)

- Rasterization algorithms can take advantage of triangle properties
- Graphics hardware is optimized for triangles
- Because triangle drawing is so fast, many systems will subdivide polygons into triangles prior to scan conversion

Barycentric Coordinates

- Consider a triangle defined by three points \( a, b, \) and \( c \).
- Define a new coordinate system in which \( a \) is the origin, \( b \) and \( c \) define the coordinate system basis vectors
- Note that the coordinate system will be non-orthogonal.

Barycentric Coordinates

- With this new coordinate system, any point can be written as:
  \[ p = a + \beta(b - a) + \gamma(c - a) \]
- Rearranging terms, we get:
  \[ p = (1 - \beta - \gamma)a + \beta b + \gamma c \]
- Let \( \alpha = (1 - \beta - \gamma) \)
- Then
  \[ p = \alpha a + \beta b + \gamma c \]
Barycentric Coordinates

- Now any point in the plane can be represented using its barycentric coordinates
- If
  \[ \alpha + \beta + \gamma = 1 \]
  0 < \alpha < 1
  0 < \beta < 1
  0 < \gamma < 1
- then the point lies somewhere in the triangle

Computing Barycentric Coordinates

- Implicit form between two points (a,b) and (a,c)
  \[
  f(a,b) = (y_a - y_b) x + (x_b - x_a) y + x_a y_b - x_b y_a
  \]
  \[
  f(a,c) = (y_a - y_c) x + (x_c - x_a) y + x_a y_c - x_c y_a
  \]
Computing Barycentric Coordinates

- To compute the barycentric coordinates of a point:

\[
\gamma = \frac{(y_a - y_c)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_c)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}
\]

\[
\beta = \frac{(y_a - y_b)x + (x_c - x_a)y + x_a y_c - x_c y_a}{(y_a - y_b)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a}
\]

\[
\alpha = 1 - \beta - \gamma
\]

Rasterization Pseudo Code

```plaintext
drawTriangle2D(x_a, y_a, x_b, y_b, x_c, y_c)
{
    for all x in screen_x
        for all y in screen_y
            compute(\alpha, \beta, \gamma) for (x, y)
                if (\alpha \in [0, 1] and \beta \in [0, 1] and \gamma \in [0, 1])
                    color = compute_color(\alpha, \beta, \gamma)
                    put_pixel(x, y, color)
}
Rasterization

Bounding Box

Barycentric Coordinates

• weighted combination of vertices
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \[ \alpha + \beta + \gamma = 1 \]
  \[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

"convex combination of points"

Barycentric Coordinates for Interpolation

• how to compute \( \alpha, \beta, \gamma \) ?
  – use bilinear interpolation or plane equations
  \[ \text{Interpolate } \alpha, \beta, \gamma \]
  \[ \alpha = a \cdot x + b \cdot y + c \cdot z + d \]
  \[ \beta = \ldots \]

  – once computed, use to interpolate any # of parameters from their vertex values
  \[ x = \alpha \cdot x_1 + \beta \cdot x_2 + \gamma \cdot x_3 \]
  \[ r = \alpha \cdot r_1 + \beta \cdot r_2 + \gamma \cdot r_3 \]
  \[ g = \alpha \cdot g_1 + \beta \cdot g_2 + \gamma \cdot g_3 \]
  etc.

Interpolation: Gouraud Shading

• need linear function over triangle that yields original vertex colors at vertices
• use barycentric coordinates for this
  – every pixel in interior gets colors resulting from mixing colors of vertices with weights corresponding to barycentric coordinates
  – color at pixels is affine combination of colors at vertices

\[
\text{Color}(\alpha \cdot \mathbf{x}_1 + \beta \cdot \mathbf{x}_2 + \gamma \cdot \mathbf{x}_3) =
\alpha \cdot \text{Color}(\mathbf{x}_1) + \beta \cdot \text{Color}(\mathbf{x}_2) + \gamma \cdot \text{Color}(\mathbf{x}_3)
\]

Gouraud Shading Scanline Alg

• algorithm
  – modify scanline algorithm for polygon scan-conversion :
  • linearly interpolate colors along edges of triangle to obtain colors for endpoints of span of pixels
  • linearly interpolate colors from these endpoints within the scanline

\[
\begin{align*}
X_{\text{min}} & < X_{\text{cur}} < X_{\text{max}} \\
\frac{X_{\text{cur}} - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} & \cdot c_1 + \left( 1 - \frac{X_{\text{cur}} - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \right) c_2
\end{align*}
\]
Filling Techniques

• Another approach to polygon fill is using a filling technique, rather than scan conversion
• Pick a point inside the polygon, then fill neighboring pixels until the polygon boundary is reached
• Boundary Fill Approach:
  – Draw polygon boundary in the frame buffer
  – Determine an interior point
  – Starting at the given point, do:
  • If the point is not the boundary color or the fill color
    Set this point to the fill color
    Propagate to the point’s neighbors and continue

Flood Fill Approach:
– Set all interior pixels to a certain color
– The boundary can be any other color
– Pick an interior point and set it to the polygon color
– Propagate to neighbors, as long as the neighbor is the interior color
– This is used for regions with multi-colored boundaries

Fill Problems

• Fill algorithms have potential problems
• E.g., 4-connected area fill:
  - Starting point
  - Fill complete

Propagating to Neighbors

• Most frequently used approaches:
  – 4-connected area
  – 8-connected area

Fill Problems

• Similarly, 8-connected can “leak” over to another polygon
  - Starting point
  - Fill complete
• Another problem: the algorithm is highly recursive
  – Can use a stack of spans to reduce amount of recursion
Pattern Filling

- Often we want to fill a region with a pattern, not just a color
- Define an n by m pixmap (or bitmap) that we wish to replicate across the region

![5x4 pixmap](image)

Object to be patterned

Final patterned object

Pattern Filling

- How do you determine the anchor point
  - A point on the polygon
  - Left-most point?
  - The pattern will move with the polygon
  - Difficult to decide the right anchor point
  - Screen (or window) origin
  - Easier to determine anchor point
  - The pattern does not move with the object

Pattern Filling

- How do we determine which color to color a point in the object?
- Use the MOD function to tile the pattern across the polygon
- For point \((x, y)\)
  - Use the pattern color located at \((x \mod m, y \mod n)\)

Pattern Example

- For the pattern shown, what color does the pixel at location \((235, 168)\) get colored, assuming the pattern is anchored at the lower left corner of the object?

![Pattern](image)

Need to find the relative distance to the point to draw:

\[ X = (235 - 225) = 10 \]
\[ Y = (168 - 163) = 5 \]

Next figure out which pattern pixel corresponds to this screen pixel:

\[ X_{\text{pattern}} = 10 \mod 4 = 2 \]
\[ Y_{\text{pattern}} = 5 \mod 5 = 0 \]

Pattern Example

- The pattern pixel \((2, 0)\) should map to screen location \((235, 168)\)

Let’s map the pattern onto the polygon and see

![Pattern Example](image)
The End

Scan Conversion