CS488

Polygon Filling, Point and Line Clipping

Luc RENAMBOT
Last time

• How line segments are drawn into the frame buffer
• Began to talk about how polygons are filled in the frame buffer.
• Today: more about filling polygons and clipping
Overall idea

- Moving from bottom to top up the polygon
- Starting at a left edge, fill pixels in spans until you come to a right edge
Filling Algorithm

• Moving from bottom to top up the polygon
  1. Find intersections of the current scan line with all edges of polygon
  2. Sort intersections by increasing x coordinate
  3. Moving through list of increasing x intersections
    • Parity bit = even (0)
    • Each intersection inverts the parity bit
    • Draw pixels when parity is odd (1)
Efficient Way

- Makes use of the fact that, similar to the *midpoint algorithm*, once we have an intersection we incrementally compute the next intersection from the current one.

- $X_{i+1} = X_i + 1/m$
Create a Global Edge Table

- Contains all edges sorted by Ymin
- Array of buckets - one bucket for each scan line in the polygon
- Each bucket has a list of entries stored in increasing X coordinate order (where X is the X value at the Ymin of the line)
- Each entry contains Ymax, Xmin (again X at Ymin), 1/m
List of Edges

A -> B -> E -> D -> C

F
### Edge Table

<table>
<thead>
<tr>
<th>Height-1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-2.5</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ymax - Xmin - 1/m - Next
Edge Table

- Entry contains
  - Ymax
  - Xmin
  - 1/m (step)
  - pointer to next edge
Algorithm

1. \( Y = \) minimum \( Y \) value in ET (index of first nonempty bucket)
2. Set Active Edge Table (AET) to be empty
3. Repeat until AET and ET are empty
   1. Move edges in ET with \( Y_{\text{min}} = y \) into AET (new edges)
   2. Sort AET on \( x \) (AET will eventually include new and old values)
   3. \textbf{Use list of \( x \) coordinates to fill spans}
   4. Remove entries in AET with \( y_{\text{max}} = y \) (edges you are finished with)
   5. Add 1 to \( y \) (to go to the next scan line)
   6. Update \( X \) values of remaining entries in AET (add \( 1/m \) to account for new \( y \) value)
Details for Step 3 (1)

- If intersection is at a non-integer value (say 4.6) is pixel 4 interior? is pixel 5 interior?
  - if we are outside the polygon (parity is even) then we round up (pixel 5 is interior)
  - if we are inside the polygon (parity is odd) then we round down (pixel 4 is interior)
Details for Step 3 (2)

• If intersection is at an integer value (say 4.0)
  • If we are at a left edge then that pixel is interior
  • If we are at a right edge then that pixel is not interior
Details for Step 3 (3)

- What if more than one vertex shares the same location? (this also handles the situation where you may end up with an odd number of edges in the AET)

- Bottom vertex of a line segment counts in parity calculation

- Top vertex of a line segment does not count in parity calculation
Details for Step 3 (4)

- What if 2 vertices define a horizontal edge?
- imperfect, but easy, solution is to ignore this edge
Example

- So the ET starts out being ...

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>⇒</td>
<td>50</td>
<td>40</td>
<td>-1</td>
<td>⇒</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>⇒</td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>⇒</td>
<td>40</td>
</tr>
</tbody>
</table>

- And the horizontal line from (10,10) to (70,10) is ignored
Why would this algorithm be much simpler if the polygon were guaranteed to be convex with no intersecting edges?
Clipping

- Since we have a separation between the models and the image created from those models, there can be parts of the model that do not appear in the current view when they are rendered.
- Pixels outside the clip rectangle are clipped, and are not displayed.
Clipping (1)

- Can clip analytically
- Knowing where the clip rectangle is
- Clipping can be done before scan-line converting a graphics primitive by altering the graphics primitive so the new version lies entirely within the clip rectangle
Clipping (2)

- Can clip by brute force
- Scissoring
- Scan convert the entire primitive but only display those pixels within the clip rectangle by checking each pixel to see if it is visible
Various Cases

• Clipping a point against a rectangle
  → Nothing or single point

• Clipping a line against a rectangle
  → Nothing or single line segment

• Clipping a rectangle against a rectangle
  → Nothing or single rectangle

• Clipping a convex polygon against a rectangle
  → Nothing or single convex polygon

• Clipping a concave polygon against a rectangle
  → Nothing or 1 or more concave polygons
Clipping Operation

- Very common operation
- Every pixel
- Every primitive / vertex
- Operation must be very efficient
Point Clipping

- Point \((X,Y)\)
- Clipping rectangle with corners
  - \((X_{\text{min}},Y_{\text{min}})\) \((X_{\text{max}},Y_{\text{max}})\)
- Point is within the clip rectangle if
  - \(X_{\text{min}} \leq X \leq X_{\text{max}}\)
  - \(Y_{\text{min}} \leq Y \leq Y_{\text{max}}\)
- \(\text{inside} = ( (X \geq X_{\text{min}}) \&\& (X \leq X_{\text{max}}) \&\& (Y \geq Y_{\text{min}}) \&\& (Y \leq Y_{\text{max}}) )\)
Line Clipping
Line Clipping

Diagram showing points P1 to P10 and line segments P1-P2, P2-P3, P3-P4, P4-P5, P5-P6, P6-P8, P8-P9, P9-P10.
After Clipping
Cohen-Sutherland Line Clipping

1. Given a line segment, repeatedly:
   - Check for trivial acceptance
   - Both endpoints within clip rectangle

2. Check for trivial rejection
   - Both endpoints outside clip rectangle **is not enough**
   - Both endpoints off the same side of clip rectangle **is enough**

3. Divide segment in two where one part can be trivially rejected
Encoding

• Clip rectangle extended into a plane divided into 9 regions
• Each region is defined by a unique 4-bit string
  • left bit = 1: above top edge (Y > Ymax)
  • 2nd bit = 1: below bottom edge (Y < Ymin)
  • 3rd bit = 1: right of right edge (X > Xmax)
  • right bit = 1: left of left edge (X < Xmin)
### Outcodes

<table>
<thead>
<tr>
<th>1001</th>
<th>1000</th>
<th>1010</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>0101</td>
<td>0100</td>
<td>0110</td>
</tr>
</tbody>
</table>

1st bit = \( Y > Y_{\text{max}} \)
2nd bit = \( Y < Y_{\text{min}} \)
3rd bit = \( X > X_{\text{max}} \)
4th bit = \( X < X_{\text{min}} \)
Algorithm

• For each line segment:
  1. Each end point is given the 4-bit code of its region
  2. Repeat until acceptance or rejection
     1. If both codes are 0000 → trivial acceptance
     2. If logical AND of codes is not 0000 → trivial rejection
     3. Divide line into 2 segments using edge of clip rectangle
        1. Find an endpoint with code not equal to 0000
        2. Move left to right across the code to find a 1 bit → the crossed edge
        3. Break the line segment into 2 line segments at the crossed edge
        4. Forget about the new line segment lying completely outside the clip rectangle

The full algorithm is given (in C) in the white book as figure 3.41 on p.116
Next Week

- Polygon Clipping
- Circles
C.S. Line Clipping