Computer Graphics: 7-Polygon Rasterization, Clipping

Prof. Dr. Charles A. Wüthrich,
Fakultät Medien, Medieninformatik
Bauhaus-Universität Weimar
caw AT medien.uni-weimar.de
Filling polygons (and drawing them)

- In general, except if we are dealing with wireframes, we would want to draw a filled polygon on our screen.
- The advantage is clear: the polygon acquires thickness and can be used to render surfaces.
- The simplest way one would do that is to draw the polygon border and then fill the region delimited by the polygon.
- In fact, this is the start point for the real algorithm, the scanline algorithm.
- The scanline algorithm combines the advantages of filling algorithms and of line tracing at the borders in a complex but very fast way.
- As input one takes an ordered list of points representing the polygon.
Scanline algorithm

- The basic idea is very simple:
  - A polygon can be filled one scanline at a time, from top to bottom
  - Order therefore polygon corners according to their highest y coordinate
  - Order each horizontal line according to the x coordinate of the edge intersections
  - Fill between pairs of edges, stop drawing until the next edge, and then restart filling again till the next one
  - Once finished the edges at current line, restart at next y value
  - Of course, one can also draw upwards
Scanline algorithm

- Notice that the number of edges remains constant between starting and ending points in the horizontal bands.
- Notice also that segments have only a limited contiguous range where they are active.
- Notice that while proceeding downwards, borders can use a mirrored DDA to be drawn.
- In this way, one can draw line borders and fill between them, after having ordered the border intersections with the current line WRT current coordinate.
Scanline algorithm

- Polygon drawing starts at the bottom.
- Out of the edges list the ones with lowest starting point are chosen.
- These will remain part of the "active edge" list until their end is met.
- When they end, they are removed and replaced by new starting edges.
- This until there is no edge left among the active edge.
- At each value of the y variable, the edge rasterization is computed, and edges are ordered by growing x.
- Colour is then filled between sorted pairs of edge rasterizations.
Triangle rasterization

• Modern graphics cards accept only triangles at the rasterization step
• Polygons with more edges are simply triangularized
• Obviously, the rasterization of a triangle is much easier
• This because a triangle is convex, and therefore a horizontal line has just the left and the right hand borders
• Filling is then done between the left side and the right side
Clipping: motivation

- Often in 2D we have drawings that are bigger than a screen
- To save drawing complexity, it is good to be able to cut the drawings so that only screen objects are drawn
- Also, one needs to protect other (invisible) regions while working on a complex drawing
- The question is how is this done
- Problem: Given a segment in the plane, clip it to a rectangular segment
Line clipping

- Let \( B \) be the screen, and let \( P_1P_2 \) be the endpoints of the segment to be drawn.
- There are four possible cases available:
  a) Whole line is visible
     \( P_1, P_2 \in B \)
  b) Line is partially visible
     \( P_1 \in B, P_2 \notin B, P_1P_2 \) intersects screen borders
  c) Line partially visible
     \( P_1, P_2 \notin B, \) but \( P_1P_2 \) intersects screen borders
  d) Line not visible
     \( P_1, P_2 \notin B \)
Line clipping Algorithm

IF \((P_1, P_2 \in B)\) /* a */
  DrawLine(P_1, P_2)
ELSE IF /* b */
  
  \(((P_1 \in B) \text{AND NOT}(P_2 \in B)) \text{ OR } \\
  ((P_2 \in B) \text{AND NOT}(P_1 \in B)))\n
  \text{compute } I = (P_1 P_2 \cap \text{borders})

  \text{IF}(P_1 \hat{B})

  \text{Drawline}(I, P_1)

  ELSE

  \text{DrawLine}(I, P_2)

ELSE /* c, d */

  \text{compute } I_1, I_2 = \\
  (P_1 P_2 \cap \text{borders})

  \text{IF } I_1, I_2 \text{ exist}

  \text{Drawline} (I_1, I_2)

END
Examples: Cohen-Sutherland algo.

Code points according to characteristics:

- Bit 0=1 if $x_p < x_{\text{min}}$ else 0
- Bit 1=1 if $x_p > x_{\text{max}}$ else 0
- Bit 2=1 if $y_p < y_{\text{min}}$ else 0
- Bit 3=1 if $y_p > y_{\text{max}}$ else 0

Use bitwise operations:

- $\text{code}(P_1) \ \text{AND} \ \text{code}(P_2) \neq 0$
  - trivial case, line not on screen
- $\text{code}(P_1) \ \text{OR} \ \text{code}(P_2) = 0$
  - trivial case, line on screen

ELSE

- compute line-borders intersection (one at time) and set their code as above
- redo clipping with shortened line

Note: before new intersection, at least one endpoint is outside WRT the border you clipped against, thus one subseg is trivially out (all left or right or up or down of screen)
Algorithm Examples
Algorithm examples

\[ P_1 P_2: P_1 = 0001, P_2 = 1000 \]
\[ P_1 \text{ AND } P_2 = 0000 \]
\[ P_1 \text{ OR } P_2 = 1001 \]
Subdivide against left,
Pick \( P_2 \), find \( P_4 \)
new line \( P_2P_4 \)
\[ P_2 P_4: P_2 = 1000, P_4 = 1000 \]
\[ P_2 \text{ AND } P_4: 1000 \text{ outside!} \]
Draw nothing

\[ Q_1 Q_2: Q_1 = 0100, Q_2 = 0000 \]
\[ Q_1 \text{ AND } Q_2 = 0000 \]
\[ Q_1 \text{ OR } Q_2 = 0100 \]
Subdivide, Pick \( Q_2 \), find \( Q_3 \)
new line \( Q_2Q_3 \)
\[ Q_2 Q_3: Q_2 = 0000, Q_3 = 0000 \]
\[ Q_2 \text{ AND } Q_3 = 0000 \]
\[ Q_1 \text{ OR } Q_3 = 0000 \text{ inside!} \]
Draw \( Q_3Q_2 \)
\[ Q_3 Q_2: Q_3 = 0100 \]

\[ R_1 R_2: R_1 = 0100, R_2 = 0010 \]
\[ R_1 \text{ AND } R_2 = 0000 \]
\[ R_1 \text{ OR } R_2 = 0110 \]
Subdivide, Pick \( R_1 \), find \( R_4 \)
new line \( R_1 R_4 \)
\[ R_1 = 0100, R_4 = 0000 \]
\[ R_1 \text{ AND } R_4 = 0000 \]
\[ R_1 \text{ OR } R_4 = 0100 \]
Subdivide, Pick \( R_4 \), find \( R_3 \)
new line \( R_3 R_4 \)
\[ R_3 = 0000, R_4 = 0000 \]
\[ R_3 \text{ AND } R_4 = 0000 \]
draw \( R_3R_4 \)
Clipping polygons

- The task is similar, but it is more complicated to achieve
- Polygon clipping may result into disjunct polys
Sutherland Hodgeman Algorithm

- Clearly, drawing polygons is a more complicated issue.
- Idea: one could follow the polygon border, and switch to following the border when the polygon leaves the screen until it re-enters it.
- This means creating a new polygon, which is trimmed to the screen.
- While following an edge, four cases are possible:
Sutherland-Hodgeman Algorithm

- The algorithm works considering polygons as lists of edges
- Input is a list $L$ of polygon edges
- Output will be a new list $L'$ of polygon edges
- The polygon is clipped against ALL screen borders one at a time

FOR all screen borders DO:
  FOR all lines in polygons DO:
    FOR all points $P$ in $L$ DO
      Compute intersection $I$ of line with current border
      IF (case 1): Do Nothing
      IF (case 2): Add $(I, \text{Succ}(P))$ to $L'$
      IF (case 3): Add $(I)$ to $L'$
      IF (case 4): Add $(\text{succ}(P))$ to $L'$
    END
  END
END
Example

• Left border
  Input: \{V_1, V_2, V_3, V_4, V_5\}
  Output: \{I_1, V_2, V_3, V_4, I_2\}

• Top Border
  Input: \{I_1, V_2, V_3, V_4, I_2\}
  Output: \{I_1, I_3, I_4, V_3, V_4, I_2\}
Clipping in 3D

- Remember the near and far clipping planes of the view frustum?
- How do I clip a polygon against them?
Clipping in 3D

- Remember the near and far clipping planes of the view frustum?
- How do I clip a polygon against them?
- As a matter of fact, it is not so different!
- The problem can be reduced to the same as in 2D, with a few differences
Clipping in 3D

• Let us consider a the far plane and a polygon
• Substitute the coordinates of the vertices of the triangle into the plane equation:
  - Front: <0
  - Back: >0
  - Plane: =0
• So we can follow the vertices exactly like in Cohen-Sutherland to clip against the plane
• A similar method can be applied for an arbitrary plane
• For the frustum planes one can do clipping one plane at a time, like in 2D (except they are 6 now)
End