Sweep fill
Sweep fill

• Reduces to filling many spans
• Inside/outside parity is relatively straightforward
• Need to compute the spans, then fill
• Need to update the spans for each scan
• Need to implement “inside” rule for ambiguous cases.
Spans

- Process - fill the bottom horizontal span of pixels; move up and keep filling
- have xmin, xmax.
- define:
  - floor(x):= if x integer, x else truncate(x)
  - ceiling(x):= truncate(x)+1
- fill from ceiling(xmin) up to but not including floor(xmax)
- consistent with convention
Algorithm

• For each row in the polygon:
  – Throw away irrelevant edges
  – Obtain newly relevant edges
  – Fill spans
  – Update spans

• Issues:
  – what aspects of edges need to be stored?
  – when is an edge relevant/irrelevant?
The next span - 1

- for an edge, have $y = mx + c$
- hence, if $y_n = m x_n + c$, then $y_{n+1} = y_n + 1 = m (x_n + 1/m) + c$
- hence, if there is no change in the edges, have:
  - $x_{\text{max}} > x_{\text{max}} + (1/m)(x_{\text{max}})$
  - $x_{\text{min}} > x_{\text{min}} + (1/m)(x_{\text{min}})$
The next span - 2

- Horizontal edges are irrelevant
- Edge is irrelevant - when $y \geq y_{\text{ymax}}$ of edge (note appeal to convention)
- Similarly, edge is relevant when $y \geq y_{\text{ymin}}$ of edge
Filling in details

- maintain a list of active edges in case there are multiple spans of pixels - known as Active Edge List.
- for each edge on the list, must know: x-value, maximum y value of edge, 1/m
- Keep edges in a table, indexed by minimum y value - Edge Table

- For row = min to row=max
  - AEL=append(AEL, ET(row));
  - remove edges whose ymax=row
  - sort AEL by x-value
  - fill spans
  - update each edge in AEL
Compute the edge table (ET) to begin. Then fill polygon and update active edge list (AEL) row by row.

Format of AEL entries

<table>
<thead>
<tr>
<th>xmin</th>
<th>l/m</th>
<th>ymax</th>
</tr>
</thead>
</table>

ET

```
3 0 5
2
1
0
```

1 0 3 → 5 0 5
### AEL just before filling

<table>
<thead>
<tr>
<th>Row=5</th>
<th>3 0 5</th>
<th>→</th>
<th>5 0 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row=4</td>
<td>3 0 5</td>
<td>→</td>
<td>5 0 5</td>
</tr>
<tr>
<td>Row=3</td>
<td>1 0 3</td>
<td>→</td>
<td>5 0 5</td>
</tr>
<tr>
<td>Row=2</td>
<td>1 0 3</td>
<td>→</td>
<td>5 0 5</td>
</tr>
<tr>
<td>Row=1</td>
<td>1 0 3</td>
<td>→</td>
<td>5 0 5</td>
</tr>
<tr>
<td>Row=0</td>
<td>1 0 3</td>
<td>→</td>
<td>5 0 5</td>
</tr>
</tbody>
</table>
Format of AEL entries

x_{\text{min}} \quad 1/m \quad y_{\text{max}}

ET

\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
4
\end{array}

\rightarrow \begin{array}{c|c|c}
1 & 1 & 4 \\
4 & -1 & 4
\end{array}
AEL just before filling

Row=4

Row=3

Row=2

Row=1

Row=0

\[
\begin{array}{ccc}
2 & -1 & 4 \\
2 & 1 & 4 \\
1 & 1 & 4 \\
1 & 1 & 4 \\
\end{array}
\rightarrow
\begin{array}{ccc}
3 & 1 & 4 \\
3 & -1 & 4 \\
4 & -1 & 4 \\
\end{array}
\]
Comments

• Sort is quite fast, because AEL is usually almost in order.
• Nonetheless, OpenGL limits to convex polygons, so two and only two elements in AEL at any time, and no sorting.
• With additional logic to keep track of what color to use, can fill in many polygons at a time.
• Can be done without floating point
Dodging floating point

• for edge, $1/m = Dx / Dy$, which is a rational number.
• store $x$ as $x_{\text{int}}$, $x_{\text{num}}$, $x_{\text{denom}} = Dy$
• then $x \rightarrow x + 1/m$ is given by:
  – $x_{\text{num}} = x_{\text{num}} + Dx$
  – if $x_{\text{num}} \geq x_{\text{denom}}$
    • $x_{\text{int}} = x_{\text{int}} + 1$
    • $x_{\text{num}} = x_{\text{num}} - x_{\text{denom}}$

• Advantages:
  – no floating point
  – can tell if $x$ is an integer or not (check $x_{\text{num}} = 0$), and get $\text{truncate}(x)$ easily, for the span endpoints.
Aliasing/Anti-Aliasing

- Analogous to the case of lines
- Anti-aliasing is done using graduated gray levels computed by by smoothing and sampling
- Problem with “slivers” (page 90) is really an aliasing problem.
Boundary fill

• Basic idea: fill in pixels inside a boundary

• Recursive formulation:
  – to fill starting from an inside point
    • if point has not been filled,
      – fill
      – call on all neighbours that are not boundary pixels.
Choice of neighbours is important

4-connected

4 connected fill of a four connected boundary doesn’t work

8 connected

Start Position
• Using spans for boundary fill means a less messy stack.
Pattern fill

• Use index into screen as index into pattern
Clipping

- 2D elements are laid out in a convenient coordinate system--perhaps km for a map--and then transformed to a frame buffer coordinate system.
- Objects that are to be drawn must lie inside frame buffer, and may have to lie inside particular region - e.g. viewport.
- We may also want to dodge additional expensive operations on objects or parts of objects that won’t be displayed.
- How do we ensure line/polygon lies inside a region?

Element in modelling coordinates

\[ \text{Transform into frame buffer coordinates} \]

\[ \text{Clip} \]

\[ \text{Convert to pixels in frame buffer} \]
Clipping lines against rectangles

Have

Compute
Cohen-Sutherland clipping (lines)

- Clip line against convex region.
- For each edge of the region, clip line against that edge:
  - line all on wrong side of some edge? throw it away (trivial reject--e.g. red line with respect to bottom edge)
  - line all on right side of all edges? doesn’t need clipping (trivial accept--e.g. green line).
  - line crosses edge? replace endpoint on wrong side with crossing point.
Cohen Sutherland - details

• Only need to clip line against edges where one endpoint is outside.
• The state (e.g., in or out) of that endpoint changes due to clipping--need to track this.
• Use “outcode” to record endpoint in/out wrt each edge. One bit per edge, 1 if out, 0 if in.

• Trivial reject:
  – outcode(p1)&outcode(p2)! = 0
• Trivial accept:
  – outcode(p1)|outcode(p2)==0
• Clipping line against edge is easy: e.g line has endpoints \( (x_s, y_s) \) and \( (x_e, y_e) \), clip against \( x=a \) gives the point:
  \[
  (a, y_s + (a - x_s)((y_e - y_s)/(x_e - x_s)))
  \]
Cohen Sutherland - Algorithm

• Compute outcodes for endpoints
• While not trivial accept and not trivial reject:
  – clip against a problem edge (i.e. one for which an outcode bit is not 0)
  – compute outcodes again
• Return appropriate data structure
Cyrus-Beck/Liang-Barsky clipping

- Parametric clipping - view line in parametric form and reason about the parameter values
- More efficient, as not computing the coordinate values at irrelevant vertices
- Line is: \[ \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + u \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \]

- Clipping conditions on parameter:
  \[ x_{\min} \leq x_1 + u\Delta x \leq x_{\max} \]
  \[ y_{\min} \leq y_1 + u\Delta y \leq y_{\max} \]
Conditions become: $u = \frac{q_k}{p_k}$

- when $p_k < 0$, as $u$ increases line goes from outside to inside
- when $p_k > 0$, line goes from inside to outside
- infinite line:
  - parallel to an edge (in which case $p_k = 0$ for some $k$, and clipping is easy)
  - or:
    - if there is a segment, parameter goes inside-inside-outside-outside.

where

$$p_1 = -\Delta x \quad q_1 = x_1 - x_{\text{min}}$$

$$p_2 = \Delta x \quad q_2 = x_{\text{max}} - x_1$$

$$p_3 = -\Delta y \quad q_3 = y_1 - y_{\text{min}}$$

$$p_4 = \Delta y \quad q_4 = y_{\text{max}} - y_1$$
Consider infinite line extension of segment. There are 3 cases:

- parallel to an edge (in which case $p_k=0$ for some $k$, and clipping is easy)
- no intersection with the clipping rectangle
- or, we go inside-inside-outside-outside (must get inside a corner, and leave the opposite corner).

- but to be on the *segment*, we need $0 \leq u \leq 1$
Cyrus-Beck/Liang-Barsky--Algorithm

- compute incoming u values, which are $q_k/p_k$ for each $p_k<0$
- compute outgoing u values, which are $q_k/p_k$ for each $p_k>0$
- parameter value for small u end of line is: $u_{\text{small}} = \max(0, \text{incoming values})$
- parameter value for large u end of line is: $u_{\text{large}} = \min(1, \text{outgoing values})$
- if $u_{\text{small}} < u_{\text{large}}$, there is a line segment - compute endpoints by substituting u values.
- Improvement (Liang-Barsky):
  - identify some rejects early as u’s are computed for each edge in turn
Nicholl-Lee-Nicholl clipping

- Some edges are irrelevant to clipping, particularly if one vertex lies inside region.
- Endpoints are: a, b
- Cases:
  - a inside
  - a in edge region
  - a in corner region
- For each case, we generate specialised test regions for b, which use simple tests (slope, >, <), and tell which edges to clip against.
- Fast, but specialized