Filling: Scan Conversion - Intro

• 2 aspects to filling:
  1. Id pixels contained by region
  2. Determine what to fill with

• Our primary concern is the id of which pixels to color

• Basic approach is to fill successive pixels in region as process intersecting scan line

• Based on spatial coherence:
  – Adjacent pixels tend to have the same characteristics
    * Therefore, only need to id points of change
  – Several types of coherence are particularly relevant to filling:
    1. Span
      * Adjacent pixels have same color
    2. Scan line
      * Adjacent lines tend to be the same
    3. Edge
      * Edges tend to intersect adjacent scan lines

• Span coherence enables efficient memory writes to frame buffer
  – Write whole word at 1X (if aligns at word boundary)
  – If not, mask ends

• Rectangles are simplest case
  – Assume upright rectangle aligned with principle axes
  – Basic algorithm:
    
    ```
    for (y = yMin to yMax)
      for (x = xMin to xMax)
        writePixel(x, y, color);
    ```
Filling: Scan Conversion - Intro (2)

• Shared edges problematic
  – Do not want to write a pixel 2X
  – Use arbitrary convention:
    * Edge pixels are not part of polygon if
      · Rectangle lies in half plane below non-vertical edge, or
      · Rectangle lies in half plane to left of vertical edge

  – Problems:
    * Pixels of lower left vertex written 2X
    * Pixels omitted in top and right sides of primitive
Filling: Scan Conversion - Polygons

• 3 step process:
  1. Find intersections of edges with scan lines
  2. Sort edges based on increasing X-intersections
  3. Fill between pairs of intersections (using parity)
     (a) Start at left end of scan line with parity even
     (b) When encounter an edge, invert parity
     (c) Fill only when parity odd

• Issues to be addressed:
  1. Given a non-integral intersection, what is interior extreme?
     - When pass from interior to exterior, round down
     - When pass from exterior to interior, round up
  2. Given an integral intersection, what is interior extreme?
     - Use half plane rule discussed earlier
  3. How should shared vertices be handled?
     - Write \( y_{min} \) vertex, ignore \( y_{max} \) vertex of an edge in parity check
     - \( y_{max} \) only written if also represents the min value of a shared edge
  4. How should horizontal edges be handled?
     - They are ignored (since endpoints are neither minima nor maxima)
Filling: Scan Conversion - Polygons (2)

- Problems with above approach:
  1. Can omit pixels that should be written (top and right edges)
  2. Does not deal with vertices shared by more than 2 edges
  3. Shared vertices may be written multiple times (unless keep a history)
     - E.g., edges shared by more than one primitive
  4. Slivers
     - Very thin primitive oriented vertically
     - May have disconnected pixels
Filling: Scan Conversion - Calculating intersections

• Could use standard midpoint algorithm and fill during scan conversion
• Problem is that algorithm has no sense of interior/exterior

• Alternative
  – Based on edge coherence
    * If edge $a$ intersects scan line $i$, it probably intersects scan line $i + 1$
  – Incremental, using integer arithmetic
  – Like midpoint algorithm,
    $$x_{i+1} = x_i + 1/m$$
  – Formulation for left edges, $m > 1$
    * End point at $(x_{min}, y_{min})$
    * Next $x$ value at $x_{min} + 1/m$
    * Since
      $$m = \frac{y_{max} - y_{min}}{x_{max} - x_{min}}$$
      Denominator of $x$ increment = $y_{max} - y_{min}$
    * Consequently, $x$ values consist of an integral part and a fractional part
    * Successive $x$ values are generated by accumulating the fractional part
      · When the fractional part $\geq 1$,
        1. Increment the integral part
        2. Decrement the fractional part
    * To avoid real arithmetic, work with numerator and denominator independently
      · Fractional part numerator = fractional part numerator + $(x_{max} - x_{min})$
      · Compare fractional part numerator to denominator ($y_{max} - y_{min}$)
  – If fractional part = 0, draw pixel
  – If fractional part $> 0$, round up and draw pixel
• Implementationally

1. Initialize numerator to \( \Delta y \)
2. Repeat until \( y = y_{\text{max}} \)
   (a) Draw pixel at \((x, y)\)
   (b) Increment \( y \)
   (c) Increment numerator by \( \Delta x \)
   (d) If \( \text{numerator} > \text{denominator} \)
      i. Increment integral part of \( x \)
      ii. Decrement numerator by \( \Delta y \)

• Example:

\( x_{\text{min}} = 5 \)
\( m = 7/3 \)
\( \Delta x = 3 \)
\( \Delta y = 7 \)

Sequence:

<table>
<thead>
<tr>
<th>scan line</th>
<th>integral part</th>
<th>numerator</th>
<th>denominator</th>
<th>draw pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>((5, i))</td>
</tr>
<tr>
<td>( i + 1 )</td>
<td>6</td>
<td>3</td>
<td></td>
<td>((6, i + 1))</td>
</tr>
<tr>
<td>( i + 2 )</td>
<td>6</td>
<td>6</td>
<td></td>
<td>((6, i + 2))</td>
</tr>
<tr>
<td>( i + 3 )</td>
<td>7</td>
<td>2</td>
<td></td>
<td>((7, i + 3))</td>
</tr>
<tr>
<td>( i + 4 )</td>
<td>7</td>
<td>5</td>
<td></td>
<td>((7, i + 4))</td>
</tr>
<tr>
<td>( i + 5 )</td>
<td>8</td>
<td>1</td>
<td></td>
<td>((8, i + 5))</td>
</tr>
<tr>
<td>( i + 6 )</td>
<td>8</td>
<td>4</td>
<td></td>
<td>((8, i + 6))</td>
</tr>
<tr>
<td>( i + 7 )</td>
<td>8</td>
<td>7</td>
<td></td>
<td>((8, i + 7))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Filling: Scan Conversion - Scan Line Fill Algorithm

• Based on 2 tables

1. Edge table
   – Holds 1 entry for each edge of each polygon
   – Stores
     (a) $x$ coordinate of end with smaller $y$ coordinate
     (b) $y$ coordinate of opposite end ($y_{\text{max}}$)
     (c) $\Delta x \ (1/m)$ from scan line to scan line
     (d) Id of polygon
   – Essentially a hash table
     * Entries indexed by smaller $y$ values
     * In a bucket, entries ordered by smaller $x$ value

2. Active edge table
   – Holds 1 entry for each edge that intersects current scan line
   – Edges ordered by $x$ intersection value

• Basic approach

\[
y = \text{smallest } y \text{ value in edgeTable} \\
\text{activeEdgeTable is empty} \\
\text{Do } \{ \\
\quad \text{Move edges with } y_{\text{Min}} = y \text{ from edgeTable to activeEdgeTable} \\
\quad \text{Delete edges with } y_{\text{Max}} = y \text{ from activeEdgeTable} \\
\quad \text{Sort activeEdgeTable on } x \\
\quad \text{Fill pixels on scan line using} \\
\quad \quad \text{pairs of } x \text{ values from activeEdgeTable} \\
\quad y += 1 \\
\quad \text{For (each non-vertical edge in activeEdgeTable)} \\
\quad \quad \text{Update } x \\
\quad \}\text{ until edgeTable and activeEdgeTable are empty}
\]

• Trapezoids and triangles can be treated as special cases: only 2 edges per scan line
• Arbitrary polygons can be treated as mesh of triangles
• Can apply algorithm 2 ways:
  1. Calculate all spans, then fill
  2. Fill on scan line by scan line basis
Filling: Scan Conversion - Circles

• Because of symmetry, only need to scan convert one arc

• To insure that pixel is interior, check \( F(P) \)
  – If \( > 0 \), choose next pixel over
  – Can be determined incrementally as discussed above

• Since scan line intersects circle exactly 2 times,
  – No need for edge table
  – Maintain current span only

• Can apply algorithm 2 ways:
  1. Calculate all spans, then fill
  2. Fill on scan line by scan line basis
Filling: Scan Conversion - Patterns

- Pattern represented as a bitmap
- Color used controlled by mode
  1. Transparent mode
     1 ⇒ draw
     0 ⇒ don’t draw
  2. Opaque mode
     1 ⇒ draw using foreground color
     0 ⇒ draw using background color
- Main issue is how pattern is anchored
  - Anchored wrt primitive
    * Pattern is ”glued” to primitive
    * Moves with primitive
    * No definitive vertex to which to anchor pattern
      · Decision left to programmer
  - Anchored wrt screen or window
    * Pattern tiled to fill area
    * Primitive ”swims” through pattern (ala OpenGL)
    * Primitive’s coordinates used as indices into pattern, using modular arithmetic
      · Given $n \times m$ pattern, coordinate $(x, y)$
        \[(x \% n, y \% m)\]
      · Pixel value ANDed with pattern
Filling: Scan Conversion - Repeated Pattern Fill

• For primitives that are scan converted multiple times (e.g., icons, characters)
  – Alternative approach is called for

• Scan convert primitive into a secondary storage area (bitmap) one time
  – Use map to draw into raster when needed
  – Drawing based on mode, etc., as discussed

• Patterns complicate this approach
  – 0’s interpreted 2 ways
    1. If outside of primitive
       * Transparent mode should be used
    2. If inside of primitive
       * Background color (of pattern) should be used

• Following technique uses 2 bitmaps
  – Silhouette mask
    * Map has 1’s for all interior pixels of primitive
    * 0’s elsewhere
  – Pattern mask
    * Start with copy of silhouette
    * Apply pattern to silhouette by ANDing
Filling: Scan Conversion - Repeated Pattern Fill (2)

- To create patterned image

1. Draw silhouette to screen using transparent mode
   - Use background color of primitive where 1’s appear
2. Draw pattern mask to screen using transparent mode
   - Use foreground color of primitive where 1’s appear
In this approach, want to fill an existing 2D shape with a color

Main issue is determining when a boundary pixel has been reached

4 basic components to these algorithms:

1. Propagation method
   - Determines next pixel to color
2. Start procedure
   - Initializes algorithm - id’s start pixel
3. Inside procedure
   - Determines whether pixel is in area to be filled
4. Set procedure
   - Changes color of pixel

2 general types of regions based on pixel connectivity:

1. 4-connected
   - Any pair of pixels can be connected by using only vectors
     (a) [ 1 0 ]
     (b) [ −1 0 ]
     (c) [ 0 1 ]
     (d) [ 0 −1 ]

2. 8-connected
   - Any pair of pixels can be connected by using only vectors
     (a) [ 1 0 ]
     (b) [ −1 0 ]
     (c) [ 0 1 ]
     (d) [ 0 −1 ]
     (e) [ 1 1 ]
     (f) [ 1 −1 ]
     (g) [ −1 1 ]
     (h) [ −1 −1 ]
   - Every 4-connected region is also 8-connected, but not vice-versa
Filling: Seed Fill Algorithms - Intro (2)

- In the following figure
  
  * The entire blue region is 8-connected
  * The 2 square regions are 4-connected

- Region boundaries can be defined in terms of colors in 2 ways:

  1. Interior-defined
     
     - Given pixel $P$, region is largest set of pixels connected to $P$ with same color as $P$

        ![Interior-defined diagram](image)

  2. Boundary-defined
     
     - Given pixel $P$, region is largest set of pixels connected to $P$ with color different from a specified boundary color

        ![Boundary-defined diagram](image)
• 3 basic types of algorithms for filling 2D regions:
  – Flood fill
    * For use with interior-defined regions
  – Boundary fill
    * For use with boundary-defined regions
  – Tint fill
    * For use with regions in which foreground color is blended with background
    * Boundary represented by pixels where foreground color fades to 0
    * Sometimes referred to as *soft fill* algorithms

• In general, these algorithms referred to as *seed fill* algorithms
  – Based on a seed pixel
Filling: Seed Fill Algorithms - Flood Fill and Boundary Fill

• Basic approach is to recurse 4 (or 8) times with neighbors of current pixel
  – Same basic approach for both flood and boundary fill
  – Only difference is boundary check

• Code p 981

• Easy to implement, but may result in stack overflow

• To avoid this problem, use spans
  – Basic algorithm:
    find rightmost pixel (r) of span containing seed
    push(r, stack)
    do {
      r <- pop(stack)
      from r, fill span from right to left
      for row above current span {
        find rightmost pixel (r) of each span connected to current span
        push(r, stack)
      }
      for row below current span {
        find rightmost pixel (r) of each span connected to current span
        push(r, stack)
      }
    } until empty(stack)
### Filling: Seed Fill Algorithms - Flood Fill and Boundary Fill (2)

<table>
<thead>
<tr>
<th>Initial</th>
<th>Seed pixel ✧</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Initial状态下像素分布图" /></td>
<td><img src="image2.png" alt="Seed pixel状态下像素分布图" /></td>
</tr>
<tr>
<td><img src="image3.png" alt="Flood Fill过程中像素分布图" /></td>
<td><img src="image4.png" alt="Boundary Fill过程中像素分布图" /></td>
</tr>
</tbody>
</table>
Filling: Seed Fill Algorithms - Flood Fill and Boundary Fill (3)

– Improvements:

1. If fill span \( b \) above another \( (a) \), and all of \( b \)'s pixels lie between end points of \( a \)
   * Do not need to check for new spans below \( b \)
   * And vice versa for spans below \( a \)
   * Note: Extent of current span line called its *shadow*

   ![Diagram](image)

   When \( b \) is filled sometime after \( a \), will not need to check below \( b \).
   But will need to in this case

2. Combine scan of new line with filling to avoid multiple passes

3. After filling current span (*parent*)
   (a) Find seed pixel in line above
   (b) Push seed, along with endpoints of *parent*, onto stack
   (c) After span containing seed filled
      i. Check if it extends beyond *parent*
      ii. If it does *not*, scan parent for another seed to left
Filling: Seed Fill Algorithms - Soft Fill

• This deals with filling a region in which the foreground has been blended with the background
  – Usually the result of antialiasing
  – Often based on alpha values

• Following assumptions are made:
  1. Region is result of blending foreground color $F$ with background color $C$
     – Pixel color $P$ is linear combination of $F$ and $C$:
       $P = tF + (1 - t)C$
     – $t$ usually alpha value, but not restricted to be
  2. Algorithm visits each pixel 1X
  3. $F$, $C$ are known, $F \neq C$

• Basic algorithm (code p 983):

  ```
  for (each pixel L in region) {
    P = color(L)
    t = (Pi - Ci)/(Fi - Ci)
    replace P with tN + (1 - t) * C
  }
  ```

• To solve for $t$, use any of

  $\begin{align*}
  P_r &= tF_r + (1 - t)C_r \\
  P_g &= tF_g + (1 - t)C_g \\
  P_b &= tF_b + (1 - t)C_b
  \end{align*}$

• Potential problems in solving for $t$

  1. $F_i = C_i$
     – Since $F \neq C$ by initial assumption, at least 2 pairs will be different
  2. $t$ values differ for different $F_i, C_i$ pairs
     – Can result from rounding errors
     – Use the $t$ value that produces the largest difference in $F_i, C_i$
     – Revised code p984
Filling: Seed Fill Algorithms - Soft Fill and Multicolored Backgrounds

- Discussion so far has addressed backgrounds of a single color
  - Would like to be able to handle backgrounds containing several colors
- Consider $n + 1$ "sufficiently general" points
  - They define an $n$-space, where
    \[ p = v_0 + t_1(v_1 - v_0) + t_2(v_2 - v_0) + \ldots + t_n(v_n - v_0) \]
    $v_0$ represents origin of coordinate system
    $v_i - v_0$ are the basis of the system
- Given that a blended color is a linear combination of foreground and background colors
  - Can think of the color as a point in $n$-space defined by the foreground and background colors
    1. Given $F$ and $C$
      * Subspace is a line in RGB space representing all colors between $F$ and $C$
    2. Given $F$, $C_1$, and $C_2$
      * Subspace is a plane in RGB space representing all colors in the plane defined by $F$, $C_1$, and $C_2$
    3. Given $F$, $C_1$, $C_2$, and $C_3$
      * Subspace is all colors in RGB space
        - Note that cannot go any farther than 4 colors
- Consider $F$, $C_1$, and $C_2$
  \[ P = sF + tC_1 + (1 - s - t)C_2 \]
  - Want to find values of $s$ and $t$ to calculate new color based on $N$
  - Any pair of $P_i = sF_i + tC_i + (1 - s - t)C_i$ provide 2 equations in 2 unknowns
  - Note that $F$, $C_1$, and $C_2$ cannot be colinear
  - As above, problems arise due to roundoff error
    * Most pronounced for most closely-parallel vectors (between colors)
    * Use pair that generates greatest difference
      \[ P' = sN + tC_1 + (1 - s - t)C_2 \]
Filling: Seed Fill Algorithms - Soft Fill and Multicolored Backgrounds (2)

• If have 2 foreground and 2 background colors, then

\[ P = rF_1 + sF_2 + tC_1 + (1 - r - s - t)C_2 \]

• Using all 3 of \( P_r, P_g, P_b \) provides 3 equations in 3 unknowns

\[ P' = rN_1 + sN_2 + tC_1 + (1 - r - s - t)C_2 \]