CS 450: COMPUTER GRAPHICS

RASTERIZING POLYGONS

SPRING 2016

DR. MICHAEL J. REALE
OVERVIEW

• We will now discuss how we fill in the pixels inside a **polygon**

• Usually, we like to work with **triangles**; if we have an arbitrary polygon, we either:
  • Split it into triangles:
    • Faster rendering for the triangles themselves
    • Can interpolate attribute values more easily
  • Apply a more general polygon-filling algorithm

• In the course of this discussion, we will talk about:
  • **Filling triangles**
  • **Identifying and splitting concave polygons**
  • **General polygon-filling algorithms**
FILLING TRIANGLES
FILLING TRIANGLES

• Let’s assume we have three 2D points defining our triangle: $P_0$, $P_1$, $P_2$
  • NOTE: Assume these are in COUNTER-CLOCKWISE ORDER!

• We need to know:
  • What pixels are inside the triangle?
  • What are the attributes for each pixel we fill in?
BARYCENTRIC COORDINATES

• Basically, we can use barycentric coordinates to figure this out:
  • Loop through all (reasonable) pixels
  • Compute barycentric coordinates
  • If these conditions are true, pixel INSIDE triangle:
    • $0 < \alpha < 1$
    • $0 < \beta < 1$
    • $0 < \gamma < 1$
  • Can use barycentric coordinates directly to interpolate vertex attributes:
    • Example: Gouraud interpolation → interpolating color from each vertex:
      \[
      C = \alpha C_0 + \beta C_1 + \gamma C_2
      \]
BRUTE FORCE RASTERIZATION

- Let's assume we will interpolate the color for each pixel filled.
- What are some of the efficiency issues with the code on the left?

```plaintext
for all x:
    for all y:
        compute (α, β, γ) for (x, y)
        if (0 ≤ α ≤ 1 && 0 ≤ β ≤ 1 && 0 ≤ γ ≤ 1):
            C = αC₀ + βC₁ + γC₂
        drawPixel(x, y, C)
```
BRUTE FORCE RASTERIZATION

• Let’s assume we will interpolate the color for each pixel filled

• **What are some of the efficiency issues with the code on the left?**
  - Going through ALL possible pixel positions
  - Actually can just check that $\alpha$, $\beta$, and $\gamma$ are all positive
    - Since $(\alpha + \beta + \gamma) = 1$, if any of them are greater than 1 $\rightarrow$ one of them HAS to be negative to compensate
    - **NOTE**: If it is a STRICTLY positive check $\rightarrow$ only fills INTERIOR of triangle (no pixels on edges or vertices)

```python
for all x:
    for all y:
        compute $(\alpha, \beta, \gamma)$ for $(x,y)$
        if $(0 \leq \alpha \leq 1 \&\&
            0 \leq \beta \leq 1 \&\&
            0 \leq \gamma \leq 1)$:
            $C = \alpha C_0 + \beta C_1 + \gamma C_2$
            drawPixel(x,y,C)
```
• Improved code:
  • Find bounding rectangle for triangle → minimum and maximum X and Y values
  • Replaced with positive checks only
    • Again, this is excluding pixels on EDGES or VERTICES of the triangle, but we’ll get to that...

\[
\begin{align*}
\text{xmin} &= \text{floor} (x_i) \\
\text{xmax} &= \text{ceiling} (x_i) \\
\text{ymin} &= \text{floor} (y_i) \\
\text{ymax} &= \text{ceiling} (y_i)
\end{align*}
\]

for y = ymin to ymax:
  for x = xmin to xmax:
    compute \((\alpha, \beta, \gamma)\) for \((x, y)\)
    if \((\alpha > 0 \land \beta > 0 \land \gamma > 0)\):
      \(C = \alpha C_0 + \beta C_1 + \gamma C_2\)
      drawPixel(x, y, C)
COMPUTING BARYCENTRIC COORDINATES

- Remember, to compute the barycentric coordinates:
  - \((x_i, y_i) = i^{th} \) vertex of the triangle
  - \(f_{ij} = \) line from \(i^{th}\) vertex to \(j^{th}\) vertex

\[
\begin{align*}
\alpha &= f_{12}(x, y) / f_{12}(x_0, y_0) \\
\beta &= f_{20}(x, y) / f_{20}(x_1, y_1) \\
\gamma &= f_{01}(x, y) / f_{01}(x_2, y_2)
\end{align*}
\]

\[
\begin{align*}
f_{01}(x, y) &= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 \\
f_{12}(x, y) &= (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1 \\
f_{20}(x, y) &= (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2
\end{align*}
\]
SAVING COMPUTATION

\[
\begin{align*}
\alpha &= f_{12}(x, y) / f_{12}(x_0, y_0) \\
\beta &= f_{20}(x, y) / f_{20}(x_1, y_1) \\
\gamma &= f_{01}(x, y) / f_{01}(x_2, y_2)
\end{align*}
\]

• One thing we can notice is that the denominators can be computed once before we enter the loops:

\[
\begin{align*}
f_\alpha &= f_{12}(x_0, y_0) \\
f_\beta &= f_{20}(x_1, y_1) \\
f_\gamma &= f_{01}(x_2, y_2)
\end{align*}
\]
INCREMENTAL METHOD

- Let’s take a closer look at $\alpha$ for a minute:
  \[ \alpha = f_{12}(x, y) / f_a \]

- In the inner loop, we will increment $x$ by 1 to get the next point
- The value of the numerator, then, would be:
  \[
  f_{12}(x+1, y) = (y_1 - y_2)(x+1) + (x_2 - x_1)y + x_1y_2 - x_2y_1 \\
  = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1 + (y_1 - y_2) \\
  = f_{12}(x, y) + (y_1 - y_2)
  \]

- SO, before we enter the inner loop, we could compute:
  \[ \alpha_0 = f_{12}(x_{\text{min}}, y) / f_a \]

- And then, each iteration:
  \[ \alpha_{k+1} = \alpha_k + (y_1 - y_2) / f_a \]

- A similar procedure can be employed for the other coordinates AND for the outer loop
WHAT ABOUT EDGES?

• If two triangles share the same edge, obviously they also write to the same pixels on the edge
• So far, we’ve avoid this by just filling the interior
• HOWEVER, that gives us GAPS between the triangles
  • We also can’t just let them BOTH draw it, because triangles might be transparent → double-coloring

• Thus, we need a CONSISTENT way to “award” a pixel to one triangle or the other...
OFF-SCREEN POINT

• If we pick some arbitrary point off-screen (say, (-1,-1))

• Let’s also say we have two triangles that share the same edge
  • ...although to split hairs, the winding order of the triangles should be different, so technically it wouldn’t be the same line equation, but still...

• Let’s further say, because of this, \( \alpha = 0 \) for the \((x,y)\) pixel we are considering

• We will award the pixel to the triangle WHOSE \( P_0 \) point is on the same side of the edge as the offscreen point
  • If the offscreen point is on the same side as \( P_0 \) \( \rightarrow \quad f_{12}(-1,-1)f_a > 0 \)

• Problem: we will have to check whether our edge actually goes THROUGH the offscreen point
  • We will not, however, do so in the code on the next slide...
MUCH IMPROVED RASTERIZATION CODE

\[
x_{\text{min}} = \text{floor}(x_i)
\]
\[
x_{\text{max}} = \text{ceiling}(x_i)
\]
\[
y_{\text{min}} = \text{floor}(y_i)
\]
\[
y_{\text{max}} = \text{ceiling}(y_i)
\]
\[
f_a = f_{12}(x_0,y_0)
\]
\[
f_b = f_{20}(x_1,y_1)
\]
\[
f_c = f_{01}(x_2,y_2)
\]

for \( y = y_{\text{min}} \) to \( y_{\text{max}} \):
  for \( x = x_{\text{min}} \) to \( x_{\text{max}} \):
    \[
    \alpha = f_{12}(x,y)/f_a\n    \]
    \[
    \beta = f_{20}(x,y)/f_b\n    \]
    \[
    \gamma = f_{01}(x,y)/f_c\n    \]
    if( \( \alpha \geq 0 \) && \( \beta \geq 0 \) && \( \gamma \geq 0 \)):
      if( \( \alpha > 0 \) || \( f_a f_{12}(-1,-1) > 0 \) &&
          \( \beta > 0 \) || \( f_b f_{20}(-1,-1) > 0 \) &&
          \( \gamma > 0 \) || \( f_c f_{01}(-1,-1) > 0 \) ):
        \[
        C = \alpha C_0 + \beta C_1 + \gamma C_2\n        \]
        drawPixel(x,y,C)
ADDITIONAL CONSIDERATIONS

• One COULD terminate early if one of the coordinates were less than 0
  • E.g., don’t bother computing $\beta$ and $\gamma$ if you already know $\alpha < 0$
  • However, extra branching $\rightarrow$ profile code to see what is actually optimizing it
• Should test to ensure not dealing with degenerate triangle (no area, all points collinear)
  • E.g, test whether ANY of these are true:
    $$f_\alpha = 0$$
    $$f_\beta = 0$$
    $$f_\gamma = 0$$
IDENTIFYING AND SPLITTING CONCAVE POLYGONS
If we decide that, for arbitrary polygons, we will first split them into triangles → need to know whether polygon is **concave** or **convex**

- **Convex** → easy to split
- **Concave** → need to make into convex polygons first
DETECTING CONCAVE POLYGONS

• We can do this one of two ways:
  • Check interior angles for one greater than 180°
  • For each edge line, check if vertices are all on the same side → if not, concave
CONVEX VS. CONCAVE: BY INTERIOR ANGLE

- **Interior angle** = angle inside polygon boundary formed by two adjacent edges

- **Convex** = all interior angles less than 180°

- **Concave** = one or more interior angles greater than or equal to 180°
CHECKING INTERIOR ANGLES

• Treat each edge as a vector (go counterclockwise): \[ E_k = V_{k+1} - V_k \]
• For each pair of consecutive edge vectors, get cross product
• If **convex** → signs of all cross products will be the same
• If **concave** → one or more will be different

\[
\begin{align*}
(E_1 \times E_2)_z &> 0 \\
(E_2 \times E_3)_z &> 0 \\
(E_3 \times E_4)_z &< 0 \\
(E_4 \times E_5)_z &> 0 \\
(E_5 \times E_1)_z &> 0
\end{align*}
\]
CHECKING INTERIOR ANGLES

\[(E_1 \times E_2)_z > 0\]
\[(E_2 \times E_3)_z > 0\]
\[(E_3 \times E_4)_z < 0\]
\[(E_4 \times E_5)_z > 0\]
\[(E_5 \times E_1)_z > 0\]

NOT THE SAME!
CHECKING INTERIOR ANGLES: 2D AND 3D

• In 2D → cross product will be all +Z or all –Z

• In 3D:
  • First transform to one of the axis-aligned planes (XY, XZ, or YZ)
  • All cross products should point the same way
    • E.g., YZ → all +X or all –X
• NOTE: If you have 3 successive collinear points anywhere → vectors parallel → cross product is ZERO VECTOR!

\[(E_1 \times E_2) = \vec{0}\]
CONVEX VS. CONCAVE: BY PICKING SIDES

• Another way to define convex vs. concave is to look at each line formed by each edge and see if all other vertices on one side or not
  • Could use implicit line formulas $f(x,y) = 0$ for each edge
  • Have to check ALL edges

• **Convex** = for ALL edge lines, all other vertices are on one side

• **Concave** = for one or more edge lines, some of the vertices are on one side and some are on the other
  • Also, one or more edge lines will intersect another edge
CONVEX VS. CONCAVE: CORNY MEMORY HOOK
SPLITTING A CONCAVE POLYGON

• There are 2 ways we can do this:
  • Vector method
  • Rotational method
SPLITTING BY VECTOR METHOD

- Transform to XY plane (if necessary)
- Get edge vectors in counterclockwise order: \[ E_k = V_{k+1} - V_k \]
- For each pair of consecutive edge vectors, get cross product
- If a cross product has a negative z component → split polygon along first vector in cross product pair
  - Have to intersect this line with other edges
SPLITTING BY ROTATIONAL METHOD

- Transform to XY plane (if necessary)
- For each vertex $V_k$:
  - Move polygon so that $V_k$ is at the origin
  - Rotate polygon so that $V_{k+1}$ is on the X axis
  - If $V_{k+2}$ is below X axis $\rightarrow$ polygon is concave $\rightarrow$ split polygon along X axis
    - Intersection calculations a little easier, because intersection with X axis
  - Repeat concave test for each of the two new polygons
- Stop when we’ve checked all vertices
Once we have a convex polygon(s), splitting into triangles is pretty straightforward:

- Every 3 consecutive vertices → make triangle
- Remove middle vertex
- Keep going until down to last 3 vertices
POLYGON FILLING ALGORITHMS
INTRODUCTION

• If, on the other hand, we decide to fill in the polygon as-is:
  • Our rasterization process is going to be more complicated
  • Interpolating attribute values will be non-trivial

• Two basic ways to fill in a polygon:
  • Scan-line approach
    • For each scan line the polygon touches, check which pixels on scan line are within polygon boundaries
  • Flood-fill or Boundary fill approaches
    • Start at interior pixel → “paint” outwards until we hit boundaries OR run out of interior pixels
GENERAL SCAN-LINE POLYGON FILLING ALGORITHM
BASIC IDEA

- General scan-line polygon filling algorithm
  - Find intersection points of polygon with scan lines
  - Sort intersections from left to right
  - For each scan line
    - Fill in interior pixels along scan line
    - Determine interior pixels using odd-even rule

- Polygons easiest to fill \(\rightarrow\) edges are linear equations \(\rightarrow\) intersecting with \(y = \text{constant}\)
PROBLEM: CROSSING VERTICES

• What happens if we cross a vertex?
  • In some cases, want to “count” it once, but in others you want it to count twice...
SOLUTION TO CROSSING VERTICES

- Look at y coordinates of previous vertex, middle vertex (the one at intersection), and the next vertex
- If y values monotonically increase or decrease → $y_5 > y_0 > y_1$
  - Edges on opposite sides of scan line →
  - Count vertex as one
- Otherwise → $y_1 < y_2 > y_3$
  - Edges on same side of scan line →
  - Vertex is a local max/min →
  - Count vertex twice
IMPLEMENTATION OF VERTEX-CROSSING SOLUTION

• **Before you start filling:**
  
  • Go through all edges, winding around polygon
  
  • Check if any set of vertices $v_{k-1}$, $v_k$, and $v_{k+1}$ have y values that monotonically increase/decrease
  
  • **Two possible options:**
    
    • *(Option 1)* Monotonically increasing/decreasing $\rightarrow$ should count as ONE crossing $\rightarrow$ shorten one of the edges
    
    • OR
    
    • *(Option 2)* NOT monotonically increasing/decreasing $\rightarrow$ should count as TWO crossings $\rightarrow$ separate vertex into TWO vertices at same location
**OPTION 1: SHORTEN ENDS**

- **(Option 1)** Monotonically increasing/decreasing → should count as ONE crossing → **shorten one of the edges**
  - Prevents you from hitting both edges that meet at vertex
  - Assumes polygon stored as *set of lines*, each with their own vertices

- **DOES NOT** consider crossing a vertex a special case
  - Basically checking whether you cross edges

- **So, when you go to fill later:**
  - ONE crossing case → only hit ONE edge/vertex
  - TWO crossings case → didn’t shorten any edges → cross two edges
OPTION 2: ADD VERTEX

- *(Option 2)* NOT monotonically increasing/decreasing → should count as TWO crossings → separate vertex into TWO vertices at same location
  - Basically add a new vertex and have one edge start/end there instead of original vertex
  - Assumes polygon stored as *edges sharing vertices*

- DOES consider crossing vertex as ONE crossing by default
  - I.e., crossing a vertex is a special case

- *So, when you go to fill later:*
  - ONE crossing case → only hit ONE vertex
  - TWO crossings case → added extra vertex → crossing TWO vertices
To compute the intersections with the scan-line and the edges:

- Plug in numbers directly → less efficient
- -OR-
- Use INCREMENTAL approach
  - Always adding +1 to y
  - Just need to figure out x → just add on inverse slope for each edge
IMPLEMENTING SCAN-LINE FILLING

• Let’s assume we already preprocessed our polygon edges → how do we actually EFFICIENTLY do the filling?
  • Don’t want to intersect with EVERY single edge for each scan-line

• Thus, we will have two lists:
  • **Sorted edge table**
    • Create this before we start fill
  • **Active edge table**
    • Create and update this as we fill the polygon
BUILDING THE SORTED EDGE TABLE

• Let’s say we have a structure to hold our edge information → EdgeInfo

• Create a list of lists for all possible y values
  • E.g., vector<EdgeInfo> *sortedEdgeTable = new vector<EdgeInfo>[windowHeight]

• For each (non-horizontal) edge E in the polygon:
  • Determine/compute the following:
    • Smallest y value (ymin) and corresponding x value
    • Largest y value
    • Inverse of slope (or Δx and Δy separately)
  • Insert edge E into SORTED scanList[ymin]
    • I.e., when the edge starts in x
  • sortedEdgeTable[y] → contains all edges STARTING at y, SORTED by x (left to right)
SORTED EDGE TABLE: EXAMPLE

- Let’s say we start with the polygon on the right
- We need to go through the following edges:
  - AB
  - BC
  - CD
  - DE
  - EA
First, we’ll create our sorted edge table
- Size = height of window
Initially, there are no edges in each list
- Again, sorted edge table = list OF lists OF EdgeInfo’s

An EdgeInfo struct will contain:
- Maximum Y
  - So we’ll know when an edge ENDS
- Corresponding X value for minimum Y
  - We will update this value as we go along
- Difference in X
- Difference in Y
SORTED EDGE TABLE: EXAMPLE

- Add edge AB to sorted edge table:
  - \( y_B \) = Maximum Y
  - \( x_A \) = corresponding X value for minimum Y
  - \( (x_B - x_A) \)
  - \( (y_B - y_A) \)
SORTED EDGE TABLE: EXAMPLE

- Add edge BC to sorted edge table:
  - \( y_B = \text{Maximum Y} \)
  - \( x_C = \text{corresponding X value for minimum Y} \)
  - \( (x_B - x_C) \)
  - \( (y_B - y_C) \)
- NOTE: Differences have to go UP in Y
  - i.e., going from C to B
Add edge CD to sorted edge table:
- \( y_C \) = Maximum Y
- \( x_D \) = corresponding X value for minimum Y
- \((x_C - x_D)\)
- \((y_C - y_D)\)

NOTE: Again, differences have to go UP in Y
- I.e., going from D to C
Add edge DE to sorted edge table:

- $y_E = \text{Maximum } Y$
- $x_D = \text{corresponding } X \text{ value for minimum } Y$
- $(x_E - x_D)$
- $(y_E - y_D)$

NOTE: We now have TWO edges that start at $y = y_D$
**SORTED EDGE TABLE: EXAMPLE**

- Add edge EA to sorted edge table:
  - \( y_E \) = Maximum \( Y \)
  - \( x_A \) = corresponding \( X \) value for **minimum** \( Y \)
  - \((x_E - x_A)\)
  - \((y_E - y_A)\)
- **NOTE:** We now have TWO edges that start at \( y = y_A \)
- **ALSO NOTE:** We MUST insert EA BEFORE AB, because it must be sorted in \( X \)!
  - If the \( x_{min} \) values are equal \( \rightarrow \) look at \( x_{max} \) OR at \( \Delta x \)

<table>
<thead>
<tr>
<th>( (\text{windowHeight} - 1) )</th>
<th>( y_B )</th>
<th>( x_C )</th>
<th>( \Delta x_{BC} )</th>
<th>( \Delta y_{CB} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_C )</td>
<td>( y_C )</td>
<td>( x_D )</td>
<td>( \Delta x_{DC} )</td>
<td>( \Delta y_{DC} )</td>
</tr>
<tr>
<td>( y_A )</td>
<td>( y_E )</td>
<td>( x_D )</td>
<td>( \Delta x_{AE} )</td>
<td>( \Delta y_{AE} )</td>
</tr>
</tbody>
</table>

**Diagram:**

- Scan line \( y_C \)
- Scan line \( y_D \)
- Scan line \( y_A \)
- Points A, B, C, D, E
- Triangle ABCDE
ACTIVE EDGE TABLE

- List also sorted by x values (left to right)
- For each scanline y:
  - Add corresponding list from sorted edge table to active list
    - I.e., add any new edges
    - Again, must add new edges so that the active list is still sorted by x!
  - Fill in appropriate areas along scanline using active list for x coordinate boundaries
    - I.e., odd-even rule
  - Remove any edges that are complete
    - I.e., y = ymax for that edge
  - Update x values for next iteration
    - I.e., add inverse slope
  - Resort edges by x values
ACTIVE EDGE TABLE: EXAMPLE

Hit $y_A$

Add list from $\text{SET}[y_A]$

Active Edge List

Scan line $y_C$

- $y_E$  $x_A$  $\Delta x_{AE}$  $\Delta y_{AE}$
- $y_B$  $x_A$  $\Delta x_{AB}$  $\Delta y_{AB}$

Scan line $y_D$

- $y_C$  $x_D$  $\Delta x_{DC}$  $\Delta y_{DC}$
- $y_E$  $x_D$  $\Delta x_{DE}$  $\Delta y_{DE}$

Scan line $y_A$

- $y_E$  $x_A$  $\Delta x_{AE}$  $\Delta y_{AE}$
- $y_B$  $x_A$  $\Delta x_{AB}$  $\Delta y_{AB}$

$\text{windowHeight} - 1$

1

0
ACTIVE EDGE TABLE: EXAMPLE

Hit $y_D$

Add list from $\text{SET}[y_D]$

Active Edge List

Scan line $y_C$

$y_C \quad x_D \quad \Delta x_{DC} \quad \Delta y_{DC}$

Scan line $y_D$

$y_C \quad x_D \quad \Delta x_{DC} \quad \Delta y_{DC}$

$y_D \quad x_D \quad \Delta x_{DC} \quad \Delta y_{DC}$

$y_A \quad x_A \quad \Delta x_{AE} \quad \Delta y_{AE}$

$y_B \quad x_A \quad \Delta x_{AB} \quad \Delta y_{AB}$

(windowHeight - 1)

$y_D \quad y_C \quad y_E \quad y_A \quad y_B$

$\Delta x_{DE} \quad \Delta y_{DE}$

$\Delta x_{AE} \quad \Delta y_{AE}$

$\Delta x_{AB} \quad \Delta y_{AB}$
ACTIVE EDGE TABLE: EXAMPLE

Hit \( y_E \nabla \)

Remove edges AE and DE

Active Edge List

\[
\begin{array}{cccc}
  y_C & x_D & \Delta x_{DC} & \Delta y_{DC} \\
  y_E & x_D & \Delta x_{DE} & \Delta y_{DE} \\
  y_E & x_A & \Delta x_{AE} & \Delta y_{AE} \\
  y_B & x_A & \Delta x_{AB} & \Delta y_{AB} \\
\end{array}
\]

\[
\begin{array}{cccc}
  y_C & x_C & \Delta x_{CB} & \Delta y_{CB} \\
  y_D & x_D & \Delta x_{DC} & \Delta y_{DC} \\
  y_E & x_D & \Delta x_{DE} & \Delta y_{DE} \\
  y_E & x_A & \Delta x_{AE} & \Delta y_{AE} \\
  y_B & x_A & \Delta x_{AB} & \Delta y_{AB} \\
\end{array}
\]
ACTIVE EDGE TABLE: EXAMPLE

Hit $y_C$
- Remove DC
- Add list from $SET[y_C]$

Active Edge List

Scan line $y_C$

Scan line $y_D$

Scan line $y_A$
ACTIVE EDGE TABLE: EXAMPLE

Hit $y_B$

Remove AB and CB

Active Edge List

Scan line $y_D$

Scan line $y_A$
FILLING CONVEX POLYGONS

• We can simplify the scan-line fill algorithm if our polygon is:
  • **Convex** → single interior area → only two edge intersections at a time
    • Fixed-size active edge list (only 2 or zero edges active at a time)
    • Fixed-sized lists inside sorted edge list
  • **Triangle** → just 3 edges total
    • Full algorithm is overkill
FILLING REGIONS WITH CURVED BOUNDARIES

• We can use the scan-line filling approach for curved boundaries (e.g., circles), but:
  • 1) Intersection done with curve equations (rather than linear equations) → more difficult
  • 2) Slope changing, so cannot use straightforward x and y increments

• *Circles and ellipses* → only two intersection points per scanline (convex shape) AND can take advantage of symmetry

• *Curve sections (e.g., elliptical arc)* → use combination of curve and linear equations

• *More complex curves* → numerical techniques OR cheat and use line segment approximations
BOUNDARY-FILL AND FLOOD-FILL ALGORITHM
INTRODUCTION

• An alternative approach to the scan-line algorithm is to:
  • 1) start at some interior pixel (i.e., your “seed” pixel)
  • 2) “paint” outwards to fill in pixels
  • 3) stop when you hit the boundary OR can’t find interior color

• Advantages: allows filling of irregularly-shaped areas
• Disadvantages: more computationally expensive

• We’re going to talk about two variants of this idea:
  • Boundary-fill algorithm
    • Stop when hit BOUNDARY color
  • Flood-fill algorithm
    • Fill if you see the INTERIOR color
PIXEL NEIGHBORS

• Given a pixel we just filled in, we need to fill in its neighbors as well
• Which pixels do we consider neighbors?
  • “Strong” neighbors → north, south, east, and west
  • “Weak” neighbors → NE, SE, NW, SW
• 4-neighbors = “Strong” neighbors
• 8-neighbors = “Strong” + “Weak” neighbors

• Regions filled with:
  • 4-neighbors → 4-connected region
  • 8-neighbors → 8-connected region
CHOOSING NEIGHBORS

- Depending on what kind of neighbors you choose, you will get different filling results:
BOUNDARY-FILL ALGORITHM

• Assume the boundary of the region has a single color B
• Keep filling region until we encounter a pixel with color B

Stop when you see boundary color
FLOOD-FILL ALGORITHM

- Assume interior region has same color C
- Fill pixel only if it has color C

- *Advantages*: boundaries can be different colors
EFFICIENT IMPLEMENTATION OF FLOOD/BOUNDARY FILLING

• Problem with these approaches → lots of pixels to check, many iterations/recursive calls → lots of pixels pushed on stack

• *Alternative:*
  • Have a stack of starting points only for each span → put seed point as first
    • In this case, a span = horizontal run of interior pixels bounded by boundary pixels
  • For each starting point:
    • Fill in span of pixels
    • Find left-most starting points on lines above and below
    • Repeat until stack of starting points empty
Seed point

Stacked Positions

2
1

Stacked Positions

3
1

Stacked Positions

6
5
4
1

Stacked Positions

5
4
1