CMSC 427: Chapter 10
Rasterization and Filling

Reading: Sect 8.1 in Shirley.
Overview:
- Curve Representations
- Line Segment Scan Conversion (DDA, Bresenham)
- Filling (Flood filling, Implicit function test, Scan-line algorithm)

Overview

- Curve Representations
- Line Segment Scan Conversion
  - Naive algorithm
  - DDA
  - Bresenham's algorithm
- Filling
  - Flood filling
  - Implicit function test
  - Scan-line algorithm
Rasterization (a.k.a. Scan Conversion)

Rasterization:
- Maps primitive geometric objects (lines, curves, triangles) to a set of pixels, typically represented as a sequence of scan-line intervals.

Must be Efficient:
- Required for every primitive and every frame.

Must be Simple:
- Implemented in assembly language or microcode.
- Implemented in GPU using only integer arithmetic.

Curve Representations

Geometric curves in 2-d (and surfaces in 3-d) are typically represented in one of two ways:

Implicit representation: The zero-set of a function \( f(x, y) \).

Parametric representation: The \((x, y)\) coordinates are a function of one or more scalar parameters.
Implicit Representation (Curve in 2D)

Implicit Representation: The points of the curve are represented by a function \( f(x, y) \). The curve is the set of points

\[ \{(x, y) \mid f(x, y) = 0 \} \]

Examples:
- **Line of slope 3/2 with y-intercept -1**: Satisfies:
  \[ y = \left(\frac{3}{2}\right) x - 1 \]
  Thus, points \((x, y)\) on the line satisfy:
  \[ 3x - 2y - 2 = 0 \]
  Implicit representation is given by \( f(x, y) = 3x - 2y - 2 \).
- **Circle of radius 4 centered at (2,3)**: Satisfies:
  \[ (x - 2)^2 + (y - 3)^2 = 4^2 \]
  Thus, point \((x, y)\) lies on the circle if and only if:
  \[ (x - 2)^2 + (y - 3)^2 - 16 = 0 \]
  Implicit representation given by \( f(x, y) = (x - 2)^2 + (y - 3)^2 - 16 \).

Parametric Representation (Curve in 2D)

Parametric Representation: Represents the curve as a set of points \((x(t), y(t))\), as two functions of a parameter \( t \).

Examples:
- **Line of slope 3/2 with y-intercept -1**: Is the set of points:
  \[ \{(x, y) \mid y = \left(\frac{3}{2}\right) x - 1 \} \]
  We can parameterize the line as a function of \( t = x \), giving
  \[ x(t) = t \quad \text{and} \quad y(t) = \left(\frac{3}{2}\right) t - 1, \quad \text{where} \ -\infty < t < \infty \]
  This is only one of many possibilities. Another is:
  \[ x(t) = 2t \quad \text{and} \quad y(t) = 3t - 1, \quad \text{where} \ -\infty < t < \infty \]
- **Circle of radius 4 centered at (2,3)**: We can parameterize the circle as a function of angle,
  \[ x(t) = 2 + 4 \cdot \cos t \quad \text{and} \quad y(t) = 3 + 4 \cdot \sin t, \quad \text{where} \ 0 \leq t \leq 2\pi \]
Which Representation is Better?

**Implicit Representation:**

**Good:** Can be used to represent not only a curve, but the interior and exterior as well.
- Unit circle: \( x^2 + y^2 = 1 \).
- Interior: \( x^2 + y^2 < 1 \).
- Exterior: \( x^2 + y^2 > 1 \).

**Good:** Easy to intersect, with other objects or with other implicit objects by forming a system of multiple equations.

**Bad:** Less intuitive.

**Bad:** Harder to generate points along the curve.

**Parametric Representation:**

**Good:** Easy to generate points on the curve or decompose into pieces.

**Good:** Can easily generate objects of any dimension. E.g., using two parameters you can generate a surface.

**Bad:** Harder to intersect with other objects.

---

Line Segment Representation: Implicit

Consider line segment between \( q = (q_x, q_y) \) and \( r = (r_x, r_y) \). Assume integer coordinates.

Let \( d = (d_x, d_y) \) be the vector \( r - q \).

**Implicit Representation:** By similar triangles we have

\[
\frac{y - q_y}{x - q_x} = \frac{d_y}{d_x}.
\]

Expressing this as a linear equation in \( x \) and \( y \) we have the implicit representation:

\[
(x - q_x)d_y - (y - q_y)d_x = 0.
\]

or equivalently:

\[
ax + by + c = 0,
\]

where

- \( a = d_y \),
- \( b = -d_x \),
- \( c = q_y d_x - q_x d_y \).

Remember this form: We will use it later in Bresenham's algorithm.
Line Segment Representation: Explicit

Consider the line segment between points \( q = (q_x, q_y) \) to \( r = (r_x, r_y) \).

Let \( d = (d_x, d_y) \) be the vector \( r - q \).

\[
(x - q_x)d_x - (y - q_y)d_y = 0.
\]

**Explicit Representation:** We can express \( y \) as an explicit function of \( x \) in slope-intercept form:

\[
y = \frac{d_y}{d_x}(x - q_x) + q_y
\]

\[
= \frac{d_y}{d_x}x + \left(q_y - \frac{d_y}{d_x}q_x\right)
\]

\[
= m \cdot x + b.
\]

Line Segment Representation: Parametric

**Parametric Representation:** This can also be represented parametrically as

\[
p(t) = q + td, \quad \text{where } 0 \leq t \leq 1.
\]

Thus, \( p(0) = q \) and \( p(1) = r \).

In terms of coordinates we have

\[
p(t) = (x(t), y(t)), \quad \text{where } 0 \leq t \leq 1,
\]

where \( x(t) = q_x + t \cdot d_x \) and \( y(t) = q_y + t \cdot d_y \).
Overview

- Curve Representations
- Line Segment Scan Conversion
  - Naive algorithm
  - DDA
  - Bresenham's algorithm
- Filling
  - Flood filling
  - Implicit function test
  - Scan-line algorithm

Pixel Coordinate System

We usually think of pixels as small squares. For scan conversion, it is more natural to think of a pixel centers as lying on a coordinate grid coinciding with the pixel centers.
Naïve Algorithm for Line Scan-Conversion

Let the line segment be from \( q = (q_x, q_y) \) to \( r = (r_x, r_y) \). As seen before, we can convert this to slope-intercept form as \( y = mx + b \). For each integer \( x \)-coordinate, round to the nearest integer \( y \)-coordinate.

**Naive Scan Conversion:**

\[
\text{for ( int } x \leftarrow q_x; x \leq r_x; x++ ) { \\
\quad \text{int } y \leftarrow \text{round} \ (m \cdot x + b) \\
\quad \text{writePixel} \ (x, y) 
}\]

**Assumptions:**
- \( q_x \leq r_x \): If not, swap \( q \) and \( r \).
- **Slope is small**: If \(|m| > 1\), then pixels will be too widely scatter. In this case, have loop step along \( y \), rather than \( x \).

Digital Differential Analyzer (DDA)

One disadvantage of the naïve algorithm is that it involves a floating-point multiplication and floating-point addition with each step.

**Idea:** Compute each \( y \)-coordinate incrementally from the last. We have \( y_i = mx_i + b \) and \( y_{i+1} = mx_{i+1} + b \). Therefore:

\[
y_{i+1} = y_i + m(x_{i+1} - x_i) \\
= y_i + m\Delta x \\
= y_i + m, \text{ if (as normal) } \Delta x = 1.
\]

**DDA Algorithm:** (Same assumptions: \( q_x \leq r_x \) and \(|m| \leq 1\).)

\[
\text{float } m \leftarrow d_y / d_x \\
\text{float } b \leftarrow q_y - (d_y/d_x)q_x \\
\text{float } y \leftarrow mq_x + b \\
\text{writePixel} \ (q_x, \text{round}(y)); \\
\text{for ( int } x \leftarrow q_x+1; x \leq r_x; x++ ) { \\
\quad y \leftarrow y + m \\
\quad \text{writePixel} \ (x, \text{round}(y)); 
}\]

// one floating point addition per step
Bresenham's Algorithm

Also called the **midpoint algorithm**. A very simple scan conversion algorithm for line segments which only requires:
- integer addition and subtraction.
- integer multiplication by 2 (equivalently a left shift by one bit).

**Inputs:** \( q = (q_x, q_y) \) and \( r = (r_x, r_y) \) with integer coordinates.

**Same Assumptions:** \( q_x \leq r_x, 0 \leq \text{slope} \leq 1 \).

---

Bresenham's Algorithm: Basics

**Initial setup:**
- Given \( q \) and \( r \), recall that we can express the line implicitly as \( a \cdot x + b \cdot y + c = 0 \).
- Multiplying by 2 does not change this, so we can instead use \( f(x, y) = 2a \cdot x + 2b \cdot y + 2c = 0 \). (We’ll see why later).

**Incremental Step:**
- Let \( p = (p_x, p_y) \) denote the previous pixel drawn. (Initially \( p = q \).)
- Since \( 0 \leq \text{slope} \leq 1 \), the next pixel is either
  - \( E = (p_x+1, p_y) \) or
  - \( NE = (p_x+1, p_y+1) \).
Bresenham's Algorithm: Basics

Incremental Step:
- Next pixel is either \( E = (p_x+1, p_y) \) or \( NE = (p_x+1, p_y+1) \). Which?
- Consider the midpoint, \( p' = (p_x+1, p_y+\frac{1}{2}) \). If the line passes below \( p' \), go to \( E \). Otherwise go to \( NE \).

Decision Value: Let \( D = f(p') \).
- If \( D \leq 0 \), then line passes on or below \( p' \) and so we go to \( E \).
- Otherwise line passes above \( p' \) and so we go to \( NE \).

Key idea: Use value of \( D \) to determine next step. Then update \( D \).

Bresenham's Algorithm: Update for E

Decision Value: \( D = f(p') \). Let \( D_{old} \) be current value.
\[
D_{old} = 2a(p_x+1) + 2b(p_y+\frac{1}{2}) + 2c = 2ap_x + 2bp_y + 2a + b + 2c.
\]

How to Update \( D \) for \( E \)? Let \( D_{new} \) be the next value of \( D \).

If we go to \( E \):
\[
D_{new} = f(p_x + 2, p_y + (1/2))
= 2a(p_x + 2) + 2b(p_y + (1/2)) + 2c
= 2ap_x + 2bp_y + 4a + b + 2c
= D + 2a = D + 2d_y.
\]
Bresenham’s Algorithm: Update for NE

Decision Value: \( D = f(p') \). Let \( D_{\text{old}} \) be current value.
\[
D = 2a(p_x + 1) + 2b(p_y + \frac{1}{2}) + 2c = 2a p_x + 2b p_y + (2a + b + 2c).
\]

How to Update \( D \) for NE? Let \( D_{\text{new}} \) be the next value of \( D \).

If we go to NE: Then \( D_{\text{new}} = f(p_x + 2, p_y + (3/2)) \)
\[
= 2a(p_x + 2) + 2b(p_y + (3/2)) + 2c
= 2a p_x + 2b p_y + 4a + 3b + 2c
= D + 2(a + b) = D + 2(d_x - d_y).
\]

Bresenham’s Algorithm: Setup

Decision Value: \( D = f(p') \).
\[
D = 2a(p_x + 1) + 2b(p_y + \frac{1}{2}) + 2c = 2a p_x + 2b p_y + (2a + b + 2c).
\]

What is \( D \)'s initial value?
\[
D_{\text{init}} = f(q_x + 1, q_y + (1/2))
= 2a(q_x + 1) + 2b(q_y + (1/2)) + 2c
= (2a q_x + 2b q_y + 2c) + 2a + b
= 2a + b = 2d_y - d_x.
\]

This is 0 because \( q \) lies on the line.
Bresenham's Algorithms: Full Algorithm

```c
bresenham (Point q, Point r) {
    // assume qx ≤ rx and 0 ≤ slope ≤ 1
    int dx, dy, D, px, py;
    dx = rx - qx; // line width and height
    dy = ry - qy;
    D = 2·dy - dx; // initial decision value
    py = qy; // start at (q.x,q.y)
    for (px = qx; px <= rx; px++) {
        writePixel(px, py); // write the current pixel
        if (D ≤ 0) D += 2·dy // below midpoint - go to E
        else { // above midpoint - go to NE
            D += 2·(dy - dx);
            py++
        }
    }
}
```

Note: Only operations needed are integer addition and subtraction and multiplication times 2.

Bresenham's Algorithm: Final Details

What if assumptions (qx ≤ rx, 0 ≤ slope ≤ 1) are not satisfied?

a) 0 ≤ slope ≤ 1: The standard version.
b) -1 ≤ slope < 0: Similar to standard, but cases are SE and E rather than E and NE. Update rules are similar, but py is decremented in SE case.
c) 1 ≤ slope ≤ ∞: Same as a), but with x and y reversed.
d) -∞ ≤ slope ≤ -1: Same as b), but with x and y reversed.

If q and r are not in proper order, then swap them.
Overview

- Curve Representations
- Line Segment Scan Conversion
  - Naive algorithm
  - DDA
  - Bresenham's algorithm
- Filling
  - Flood filling
  - Implicit function test
  - Scan-line algorithm

Polygon Scan Conversion

Polygon Scan Conversion: Given a 2-dimensional polygon, determine which pixels lie within this polygon and color (or shade) them.

Coordinate Grid: Think of pixels as lying on a coordinate grid coinciding with the pixel centers.
Polygon Scan Conversion

When does a pixel lie within a polygon?

**Implicit Function:** Test whether the center of each pixel lies on the appropriate side of each of the polygon's sides.

**Even-odd Rule:** Count the times a line through the pixel to infinity crosses the polygon's edge. **Odd → Inside; Even → Outside.**

**Winding Number:** Count how many times the boundary "winds" around the pixel. **Non-Zero → Inside; Zero → Outside.**

These rules produce the same result if the polygon is simple but may differ if boundary self-intersects.

Flood Filling

**Legend:** White = Uncolored  
Orange = Colored

**Flood Filling:** Given a region bounded by a set of colored pixels, and a starting pixel within this region, color everything that is reachable within the region from the starting pixel.
Flood Filling

\[
\text{Legend: White = Uncolored} \quad \text{Orange = Colored}
\]

\[
\text{floodFill}(x, y) \{
\text{if (readPixel}(x, y) \neq \text{Colored}) \{
\text{writePixel}(x, y) \leftarrow \text{Colored};
\text{floodFill}(x - 1, y);
\text{floodFill}(x + 1, y);
\text{floodFill}(x, y - 1);
\text{floodFill}(x, y + 1);
\}
\}
\]

Note: This is basically depth-first search, where
“Colored” means “Visited”.

Chapter 10, Slide 27
Copyright © D. M. Mount and A. Varshney
Flood Filling

Legend: White = Uncolored
Orange = Colored

\[
floodFill(x, y) \{
    \text{if (readPixel}(x, y) \neq \text{Colored}) \{
        \text{writePixel}(x, y) \leftarrow \text{Colored};
        \text{floodFill}(x - 1, y);
        \text{floodFill}(x + 1, y);
        \text{floodFill}(x, y - 1);
        \text{floodFill}(x, y + 1);
    \}
\}
\]
Flood Filling

Legend: White = Uncolored
Orange = Colored

```plaintext
floodFill (x, y) {
    if (readPixel (x, y) ≠ Colored) {
        writePixel (x, y) ← Colored;
        floodFill (x - 1, y);
        floodFill (x + 1, y);
        floodFill (x, y - 1);
        floodFill (x, y + 1);
    }
}
```

Chapter 10, Slide 31
Copyright © D. M. Mount and A. Varshney

Connectedness

What pixels are "connected"?

- → 4-connected
- → 8-connected

Easier to "leak out" of a region when using 8-connectedness than in 4-connectedness.
**Connectedness**

If we try the same algorithm but trying to fill all 8 neighbors in the previous example, the algorithm will leak out of the region.

Eventually the entire window will be colored.

**Limitations of Flood Filling**

- Unlike the even-odd rule and winding number method, flood filling cannot fill objects with non-simple (self intersecting) boundaries.

- May visit the same pixel several times (but will color it only once) - wasted effort.

- Requires an ability to read-back the buffer (which is on the GPU). This is slow, since data transfer is optimized to move from the CPU to the GPU.
Implicit Function Test

For convex polygons: Each pixel must lie within halfplanes defined by the edges of the polygon.

\[ e_1 = a_1 x + b_1 y + c_1 \]
\[ e_2 = a_2 x + b_2 y + c_2 \]
\[ e_3 = a_3 x + b_3 y + c_3 \]

Select all pixels (whose centers) satisfy
\[ a_1 x + b_1 y + c_1 < 0. \]
Implicit Function Test

Select further the subset of selected pixels for which
\[ a_2x + b_2y + c_2 < 0. \]

[Diagram showing a shaded triangle]

Chapter 10, Slide 37
Copyright © D. M. Mount and A. Vezhney

Implicit Function Test

Select further subset of selected pixels for which
\[ a_3x + b_3y + c_3 < 0. \]

[Diagram showing a shaded triangle]

Chapter 10, Slide 38
Copyright © D. M. Mount and A. Vezhney
**Implicit Function Test**

**Advantages:**
- Great for parallel implementation.
  (One processor per pixel: Pixel Planes architecture)
- Easy to implement.

**Disadvantages:**
- Works only on convex polygons.
  - Not significant since everyone uses triangles.
- Requires $k$ tests per pixel, where $k =$ number of edges.

---

**Scan-line Algorithm**

**Edge Table:** Use bucket sort to order edges.

![Diagram showing scan-line algorithm with equations and coordinates for points a, b, c, and x₀ = 12.](image)
**Scan-line Algorithm**

**Active Edge Table (AET):**
- Changes incrementally from one scan-line to the next.
- Stores the sorted edge sequence intersected by the scan-line.

For every scan-line $i$:
- Add new entries from `Edge_Table[i]` (for all starting edges) and set $x$ coordinate to $x_0$.
- Color the interval of pixels between consecutive $x$-values.
- For each edge in AET do:
  - $\Delta y = \Delta y - 1$ // decrement row count
  - $x = x + \Delta x$ // adjust $x$-coordinate
- if ($\Delta y = 0$) remove entry from AET; // bottom of line

---

**AET:**

```
  ab: (\Delta y = 9, \Delta x = -1, x = 12),
  ac: (\Delta y = 12, \Delta x = 0.5, x = 12)
```

$y = 0$
Scan-line Algorithm

AET: [ ab: (Δy = 9-1 = 8, Δx = -1, x = 12+Δx = 11),
   ac: (Δy = 12-1 = 11, Δx = 0.5, x = 12+Δx = 12.5) ]

y = 1

Scan-line Algorithm

AET: [ ab: (Δy = 8-1 = 7, Δx = -1, x = 11+Δx = 10),
   ac: (Δy = 11-1 = 10, Δx = 0.5, x = 12.5+Δx = 13) ]

y = 2
Scan-line Algorithm

\[ \text{AET: [ } \begin{align*}
&ab: (\Delta y = 0, \Delta x = -1, x = 3) , \\
&ac: (\Delta y = 3, \Delta x = 0.5, x = 16.5) \end{align*} \] \leftarrow \text{Remove this and insert bc} \]

\[ y = 9 \]

Scan-line Algorithm

\[ \text{AET: [ } \begin{align*}
&bc: (\Delta y = 3 - 1 = 2, \Delta x = 5, x = 3 + \Delta x = 8), \\
&ac: (\Delta y = 3 - 1 = 2, \Delta x = 0.5, x = 16.5 + \Delta x = 17) \end{align*} \]

\[ y = 10 \]
Scan-line Algorithm

**Final Result**

![Algorithm visualization](image)

**Advantages:**
- Incremental, efficient
- Can handle concave, convex polygons

**Disadvantages:**
- Cannot handle non-simple (self-intersecting) polygons
  (But these are rare in most real-life graphics datasets.)
Summary:

- Curve Representations
- Line Segment Scan Conversion
  - Naïve algorithm
  - DDA
  - Bresenham's algorithm
- Filling
  - Flood filling
  - Implicit function test
  - Scan-line algorithm