Rasterization: Geometric Primitives

Outline

• Rasterizing lines
• Rasterizing polygons
Rasterization: What is it?

• How to go from real numbers of geometric primitives’ vertices to integer coordinates of pixels on screen
• Geometric primitives
  - Points
  - Lines
  - Polygons

Rasterizing Lines: Goals

• Basics:
  - Draw pixels as close to the ideal line as possible
  - Use the minimum number of pixels without leaving any gaps
  - Do it efficiently
• Extras
  - Minimize aliasing ("jaggies")
  - Handle different line styles
    • Width
    • Stippling (dotted and dashed lines)
Line Drawing

- Slope-intercept form of a line: \( y = mx + b \)
  - \( m = \frac{dy}{dx} \)
  - \( b \) is where the line intersects the \( Y \) axis

DDA Line Drawing

- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions
- DDA's basic idea: If we increment the \( x \) coordinate by 1 pixel at each step, the slope of the line tells us how to much increment \( y \) per step
  - This only works if \( m \leq 1 \)—otherwise there are gaps
  - Solution: Reverse axes and step in \( Y \) direction.
DDA Line Drawing

- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions.
- DDA's basic idea: If we increment the $x$ coordinate by 1 pixel at each step, the slope of the line tells us how much increment $y$ per step.
  - This only works if $m \leq 1$—otherwise there are gaps.
  - Solution: Reverse axes and step in $Y$ direction.

DDA: Algorithm

1. Given endpoints $(x_0, y_0), (x_1, y_1)$
   - Integer coordinates: Round if endpoints were originally real-valued.
   - Assume $(x_0, y_0)$ is to the left of $(x_1, y_1)$: Swap if they aren’t.
2. Then we can compute non-negative slope:
   \[ m = dy/dx = (y_1 - y_0) / (x_1 - x_0) \]
   - Iterate
     - If $m \leq 1$: Iterate integer $x$ from $x_0$ to $x_1$, incrementing by 1 each step.
       - Initialize real $y = y_0$.
       - Add $m$ to $y$ at each step and plot point $(x, \text{round}(y))$.
     - If $m > 1$: Iterate integer $y$ from $y_0$ to $y_1$, incrementing by 1.
       - Initialize real $x = x_0$.
       - Add $1/m$ to $x$ at each step and plot $(\text{round}(x), y)$. 

DDA: Notes

- Can deal with negative slopes by consideration of symmetry
  1. E.g., rotate 90 degrees clockwise to iterate: \((x, y) \rightarrow (y, -x)\)
  2. Treat as normal case
  3. Unrotate 90 degrees counterclockwise to draw

- DDA is somewhat slow
  - Floating-point calculations, rounding are relatively expensive
Midpoint (Bresenham’s) line drawing

- Big idea: Avoid rounding, do everything with integer arithmetic for speed
- Assume slope between 0 and 1
  - Again, handle lines with other slopes by using symmetry

Midpoint line drawing: Line equation

- Recall that the slope-intercept form of the line is
  \[ y = \left(\frac{dy}{dx}\right)x + b \]
- Multiplying through by \(dx\), we can rearrange this in **implicit form** for all points on the line:
  \[ F(x, y) = dyx - dxy + dx\ b = 0 \]
- \(F\) is:
  - Zero for points on the line
  - Positive for points below the line
  - Negative for points above the line
Midpoint line drawing: The Decision

- Given our assumptions about the slope, after drawing \((x, y)\) the only choice for the next pixel is between the upper pixel \(U = (x + 1, y + 1)\) and the lower one \(L = (x + 1, y)\).

Midpoint line drawing: Midpoint decision

- After drawing \((x, y)\), in order to choose the next pixel to draw we consider the midpoint \(M = (x + 1, y + 0.5)\):
  - If \(M\) is on the line, then \(U\) and \(L\) are equidistant from the line
  - If \(M\) is below the line, pixel \(U\) is closer to the line than pixel \(L\)
  - If \(M\) is above the line, then \(L\) is closer than \(U\).
Midpoint line drawing: Midpoint decision

- So $F$ is a **decision function** about which pixel to draw:
  - If $F(M) = F(x + 1, y + 0.5) > 0$ ($M$ below the line), we pick $U$
  - If $F(M) = F(x + 1, y + 0.5) \leq 0$ ($M$ above or on line), we pick $L$

Midpoint line drawing: Implementation

- Key efficiency insight: $F$ does not have to be fully evaluated every step
- Suppose we do the full evaluation once and get $F(x + 1, y + 0.5)$
  - If we choose $L$, next midpoint to evaluate $M'$ is at $F(x + 2, y + 0.5)$
  - If we choose $U$, next midpoint $M''$ would be at $F(x + 2, y + 1.5)$
Midpoint line drawing: Implementation

- Key efficiency insight: $F$ does not have to be fully evaluated every step
- Suppose we do the full evaluation once and get $F(x + 1, y + 0.5)$
  - If we choose $L$, next midpoint to evaluate $M'$ is at $F(x + 2, y + 0.5)$
  - If we choose $U$, next midpoint $M''$ would be at $F(x + 2, y + 1.5)$
Midpoint line drawing: Implementation

Expanding these out:

\[ F_{3r} = F(x + 1, y + 0.5) = dy(x + 1) - dx(y + 0.5) + dx b \]
\[ F_{3r} = F(x + 2, y + 0.5) = dy(x + 2) - dx(y + 0.5) + dx b \]
\[ F_{3u} = F(x + 2, y + 1.5) = dy(x + 2) - dx(y + 1.5) + dx b \]

But \( F_{3f} - F_{3u} = dy \) and \( F_{3f} - F_{3l} = dy - dx \)

So depending on whether we choose \( L \) or \( U \), we just have to add \( dy \) or \( dy - dx \), respectively, to the old value of \( F \) in order to get the new value.

Midpoint line drawing: Algorithm

- To initialize, we do a full calculation of \( F \) at the first midpoint next to the left endpoint:

\[
F(x_0 + 1, y_0 + 0.5) = dy(x_0 + 1) - dx(y_0 + 0.5) + dx b
\]
\[
= dy x_0 - dx y_0 + dx b + dy - 0.5 dx
\]
\[
= F(x_0, y_0) + dy - 0.5 dx
\]

- But \( F(x_0, y_0) = 0 \) since it's on the line, so our first \( F = dy - 0.5 dx \)

- Only the sign matters for the decision, so to make it an integer value we multiply by 2 to get \( 2F = 2 dy - dx \)

- To update, keep current values for \( x \) and \( y \) and a running total for \( F \):
  - When \( L \) is chosen: \( F += 2dy \) and \( x++ \)
  - When \( U \) is chosen: \( F += 2(dy-dx) \) and \( x++, y++ \)
Polygon Rasterization

- Given a set of vertices, want to fill the interior
- Basic procedure:
  - Iterate over scan lines between top and bottom vertex
  - For each scan line, find all intersections with polygon edges
  - Sort intersections by $x$-value and fill pixel runs between pairs of intersections