Computer Graphics

- Rasterization -

Overview

• So far:
  – Clipping

• Today:
  – Drawing 1D shapes
    • speed
    • quality
    • consistency
  – Filling 2D shapes
    • Finding inside pixels
    • Ambiguities

• Next:
  – RC presentation, computer graphics arts
Computer Graphics

- Rasterization -

Marcus Magnor
Philipp Slusallek

Overview

• So far:
  – Clipping

• Today:
  – Drawing 1D shapes
    • speed
    • quality
    • consistency
  – Filling 2D shapes
    • Finding inside pixels
    • Ambiguities

• Next:
  – RC presentation, computer graphics arts
Cohen-Sutherland revisited

- Unknown case: How to decide against which plane to clip
  1. Take one endpoint outside window (outcode ≠ 0000)
  2. Set outcode bits correspond to actual clipping planes
  3. From left to right (or right to left): intersect line with set-bit plane, assign intersection point as new end point
  4. Switch corresponding bit to 0
  5. Trivial accept / reject? No: repeat from 3. for next set-bit plane

You are here …
Cohen-Sutherland revisited

- Unknown case: How to decide against which plane to clip
  1. Take one endpoint outside window (outcode ≠ 0000)
  2. Set outcode bits correspond to actual clipping planes
  3. From left to right (or right to left): intersect line with set-bit plane, assign intersection point as new end point
  4. Switch corresponding bit to 0
  5. Trivial accept / reject? No: repeat from 3. for next set-bit plane

![Diagram of clipping planes and endpoints]

You are here …
Shapes to Draw

- Shapes to draw
  - Lines
  - Circles, ellipses
  - Spline curves
  - ...

- **Rasterization** is the process of deciding which pixels to fill
  - Term comes from the regular *raster* grid pattern for pixels

- **Necessity of pixel displays**
  - Line is infinitely thin, pixel is not
  - Want to draw best approximation to ideal line
  - Want to be efficient

Drawing a Line

- **Assumption**
  - Pixels are sample points on a 2D-integer-grid
    - OpenGL: integer-coordinate bottom left; X11, Foley: in the middle
  - Simple raster operations
    - setting of binary pixels
    - antialiasing later
  - End points at pixel coordinates
    - simple generalization
  - On straight lines: gradient $|m| \leq 1$
    - separate handling of horizontal and vertical lines
    - otherwise exchange of $x$ & $y$: $|1/m| \leq 1$
  - Line width is one pixel
    - $|m| \leq 1$: 1 pixel per column (X-driving axis)
    - $|m| > 1$: 1 pixel per row (Y-driving axis)

$\Rightarrow$ Jaggies, aliasing!
Shapes to Draw

- **Shapes to draw**
  - Lines
  - Circles, ellipses
  - Spline curves
  - ...

- **Rasterization** is the process of deciding which pixels to fill
  - Term comes from the regular raster grid pattern for pixels

- **Necessity of pixel displays**
  - Line is infinitely thin, pixel is not
  - Want to draw best approximation to ideal line
  - Want to be efficient

---

Drawing a Line

- **Assumption**
  - Pixels are sample points on a 2D-integer-grid
    - OpenGL: integer-coordinate bottom left; X11, Foley: in the middle
  - Simple raster operations
    - setting of binary pixels
    - antialiasing later
  - End points at pixel coordinates
    - simple generalization
  - On straight lines: gradient $|m| \leq 1$
    - separate handling of horizontal and vertical lines
    - otherwise exchange of $x$ & $y$: $|1/m| \leq 1$
  - Line width is one pixel
    - $|m| \leq 1$: 1 pixel per column (X-driving axis)
    - $|m| > 1$: 1 pixel per row (Y-driving axis)

$\Rightarrow$ Jaggies, aliasing!
Lines: As Function

- **Specification**
  - end points: \((x_0, y_0), (x_e, y_e)\)
  - functional form: \(y = mx + B\)

- **Goal**
  - find pixels whose distance to the line is smallest

- **Brute-Force-Algorithm**
  - it is assumed that +X is the driving axis
  
  ```
  for x_i = x_0 to x_e
  y_i = m * x_i + B
  setpixel(x_i, Round(y_i))
  // Round(y_i)=Floor(y_i+0.5)
  ```

- **Comments**
  - \(m\) and \(y_i\) must be calculated with floating-point precision
  - expensive operations per pixel

---

Lines: DDA Algorithm

- **DDA: Digital Differential Analyzer**
  - Origin: solvers for simple incremental differential equations (the Euler method)
    - per step in time: \(x' = x + \frac{dx}{dt}, y' = y + \frac{dy}{dt}\)

- **Incremental algorithm**
  - Per pixel
    - \(x_{i+1} = x_i + 1\)
    - \(y_{i+1} = m (x_i + 1) + B = y_i + m\)
    - `setpixel(x_{i+1}, Round(y_{i+1}))`

- **Remark**
  - Utilization of line coherence through incremental calculation
    - avoids multiplication
    - Cumulative error
      - usually negligible for short lines
      - double precision is recommended
    - Still floating point operations necessary
### Lines: As Function

**• Specification**
- end points: \((x_0, \ y_0), \ (x_e, \ y_e)\)
- functional form: \(y = mx + B\)

**• Goal**
- find pixels whose distance to the line is smallest

**• Brute-Force-Algorithm**
- it is assumed that \(+X\) is the driving axis

\[
\begin{align*}
\text{for } x_i &= x_0 \text{ to } x_e \\
y_i &= m \times x_i + B \\
\text{setpixel}(x_i, \ \text{Round}(y_i)) \\
// \text{ Round}(y_i) = \text{Floor}(y_i + 0.5)
\end{align*}
\]

**• Comments**
- \(m\) and \(y_i\) must be calculated with floating-point precision
- expensive operations per pixel

---

### Lines: DDA Algorithm

**• DDA: Digital Differential Analyzer**
- Origin: solvers for simple incremental differential equations (the Euler method)
- per step in time: \(x' = x + \frac{dx}{dt}, \ y' = y + \frac{dy}{dt}\)

**• Incremental algorithm**
- Per pixel
  - \(x_{i+1} = x_i + 1\)
  - \(y_{i+1} = m \times (x_i + 1) + B = y_i + m\)
  - \(\text{setpixel}(x_{i+1}, \ \text{Round}(y_{i+1}))\)

**• Remark**
- Utilization of line coherence through incremental calculation
- avoids multiplication
- Cumulative error
  - usually negligible for short lines
  - double precision is recommended
- Still floating point operations necessary
Lines: Midpoint Line Algorithm

- **Bresenham (’63)**
  - Also incremental, but integer arithmetic only
  - Uses a decision variable instead of the actual line equation
  - Presented for slope between 0 and 1, others can be done by symmetry
  - Implicit definition of line function: \( F(x,y) := ax+by+c = 0 \)

\[
\begin{align*}
F(x,y) &< 0 \\
F(x,y) &= 0 \\
F(x,y) &> 0
\end{align*}
\]

Bresenham Algorithm: Overview

- **Goal:** For each \( x \), plot the pixel whose \( y \)-value is closest to the line
  - Given \((x_i, y_i)\), must choose from either \((x_i+1, y_i+1)\) or \((x_i+1, y_i)\)

- **Idea:** compute a decision variable
  - Value that will determine which pixel to draw
  - Easy to update from one pixel to the next

- **Bresenham algorithm:** midpoint algorithm for lines
  - Other midpoint algorithms for conic sections (circles, ellipses)
Lines: Midpoint Line Algorithm

- **Bresenham ('63)**
  - Also incremental, but integer arithmetic only
  - Uses a decision variable instead of the actual line equation
  - Presented for slope between 0 and 1, others can be done by symmetry
  - Implicit definition of line function: \( F(x,y) := ax + by + c = 0 \)

\[
\begin{align*}
F(x,y) &< 0 \\
F(x,y) &= 0 \\
F(x,y) &> 0
\end{align*}
\]

Bresenham Algorithm: Overview

- **Goal:** For each \( x \), plot the pixel whose \( y \)-value is closest to the line
  - Given \((x_i, y_i)\), must choose from either \((x_i + 1, y_i + 1)\) or \((x_i + 1, y_i)\)
- **Idea:** compute a **decision variable**
  - Value that will determine which pixel to draw
  - Easy to update from one pixel to the next
- **Bresenham algorithm:** midpoint algorithm for lines
  - Other midpoint algorithms for conic sections (circles, ellipses)
Midpoint Method

- Consider the midpoint between \((x_i+1, y_i+1)\) and \((x_i+1, y_i)\)
- If it’s above the line, we choose \((x_i+1, y_i)\), otherwise we choose \((x_i+1, y_i+1)\)

Midpoint Decision Variable

- Write the line in implicit form:
  \[ D = x_2 - x_1, \quad D = y_2 - y_1 \]
  \[ F(x, y) = ax + by + c = \Delta y \cdot x - \Delta x \cdot y + (\Delta x \cdot y_i - \Delta y \cdot x_i) \]
- The value of \(F(x, y)\) tells us where pixels are with respect to the line
  - \(F(x, y) = 0\): the point is on the line
  - \(F(x, y) < 0\): The point is above the line
  - \(F(x, y) > 0\): The point is below the line
- The decision variable is the value of
  \[ d_i = 2F(x_i+1, y_i+0.5) \]
  - The factor of two makes the math easier: eliminates fraction
**Midpoint Method**

- Consider the midpoint between \((x_i+1,y_i+1)\) and \((x_{i+1},y_i)\)
- If it’s above the line, we choose \((x_i+1,y_i)\), otherwise we choose \((x_{i+1},y_{i+1})\)

![Diagram of midpoint method](attachment:image.png)

**Midpoint Decision Variable**

- Write the line in *implicit form*:
  \[\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1\]
  \[F(x,y) = ax + by + c = \Delta y \cdot x - \Delta x \cdot y + (\Delta x \cdot y_1 - \Delta y \cdot x_1)\]

- The value of \(F(x,y)\) tells us where pixels are with respect to the line
  - \(F(x,y) = 0\): the point is on the line
  - \(F(x,y) < 0\): The point is above the line
  - \(F(x,y) > 0\): The point is below the line

- The decision variable is the value of \(d_i = 2F(x_i+1,y_{i+0.5})\)
  - The factor of two makes the math easier: eliminates fraction
What Can We Decide?

\[ d_i = 2\Delta y(x_i + 1) - 2\Delta xy_i + \Delta x(2c - 1) \]

- \( d_i \) negative => next point at \((x_i+1, y_i)\)
- \( d_i \) positive => next point at \((x_i+1, y_i+1)\)
- At each point, we compute \( d_i \) and decide which pixel to draw
- How do we update it? What is \( d_{i+1} \)?

Updating The Decision Variable

- \( d_{k+1} \) is the old value, \( d_k \), plus an increment:
  \[ d_{k+1} = d_k + (d_{k+1} - d_k) \]
- If we chose \( y_{i+1} = y_i + 1 \):
  \[ d_{k+1} = d_k + 2\Delta y - 2\Delta x \]
- If we chose \( y_{i+1} = y_i \):
  \[ d_{k+1} = d_k + 2\Delta y \]
- What is \( d_i \) (assuming integer endpoints)?
  \[ d_i = 2\Delta y - \Delta x \]
- Notice that we don’t need \( c \) any more
What Can We Decide?

\[ d_i = 2\Delta y(x_i + 1) - 2\Delta xy_i + \Delta x(2c - 1) \]

- \(d_i\) negative => next point at \((x_i+1,y_i)\)
- \(d_i\) positive => next point at \((x_i+1,y_i+1)\)
- At each point, we compute \(d_i\) and decide which pixel to draw
- How do we update it? What is \(d_{i+1}\)?

Updating The Decision Variable

- \(d_{k+1}\) is the old value, \(d_k\), plus an increment:
  \[ d_{k+1} = d_k + (d_{k+1} - d_k) \]
- If we chose \(y_{i+1}=y_i+1\):
  \[ d_{k+1} = d_k + 2\Delta y - 2\Delta x \]
- If we chose \(y_{i+1}=y_i\):
  \[ d_{k+1} = d_k + 2\Delta y \]
- What is \(d_i\) (assuming integer endpoints)?
  \[ d_i = 2\Delta y - \Delta x \]
- Notice that we don’t need \(c\) any more
Bresenham Algorithm

- For integers, slope between 0 and 1:
  - \(x = x_1, \ y = y_1, \ d = 2dy - dx\), draw \((x, y)\)
  - until \(x = x_2\)
    - \(x = x + 1\)
    - If \(d > 0\) then \{ \(y = y + 1\); draw \((x, y)\); \(d = d + 2\Delta y - 2\Delta x\); \}
    - If \(d < 0\) then \{ \(y = y\); draw \((x, y)\); \(d = d + 2\Delta y\); \}

- Compute the constants \((2\Delta y - 2\Delta x\) and \(2\Delta y\)) once at the start
  - Inner loop does only adds and comparisons

- Floating point has slightly more difficult initialization, but is otherwise the same

- Care must be taken to ensure that it doesn’t matter which order the endpoints are specified in (make a uniform decision if \(d == 0\))

Example: \((2,2)\) to \((7,6)\)

\[x = x_1, \ y = y_1, \ d_1 = 2dy - dx\]

<table>
<thead>
<tr>
<th>(d)</th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\(\Delta x = 5, \ \Delta y = 4\)
Bresenham Algorithm

- For integers, slope between 0 and 1:
  - \( x = x_1, \ y = y_1, \ d = 2dy - dx \), draw \((x, y)\)
  - until \( x = x_2 \)
    - \( x = x + 1 \)
    - If \( d > 0 \) then \( \{ y = y + 1; \text{draw} (x, y); \ d = d + 2\Delta y - 2\Delta x; \} \)
    - If \( d < 0 \) then \( \{ y = y; \text{draw} (x, y); \ d = d + 2\Delta y; \} \)

- Compute the constants \( (2\Delta y - 2\Delta x \text{ and } 2\Delta y) \) once at the start
  - Inner loop does only adds and comparisons

- Floating point has slightly more difficult initialization, but is otherwise the same

- Care must be taken to ensure that it doesn’t matter which order the endpoints are specified in (make a uniform decision if \( d == 0 \))

---

Example: \((2,2)\) to \((7,6)\)

\[ x = x_1, \ y = y_1, \ d_1 = 2dy - dx \]

\[ \text{If } d > 0 \text{ then } \{ y = y + 1; \text{draw} (x, y); d = d + 2\Delta y - 2\Delta x; \} \]
\[ \text{If } d < 0 \text{ then } \{ y = y; \text{draw} (x, y); d = d + 2\Delta y; \} \]
Example: (2,2) to (7,6)

\[ \Delta x = 5, \Delta y = 4 \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

**Lines: Arbitrary Directions**

- 8 different cases
  - driving (active) axis: ±X or ±Y
  - Increment/decrement of y or x, respectively

\[ +Y,x-- \quad +Y,x++ \]
\[ -X,y++ \quad +X,y++ \]
\[ -X,y-- \quad +X,y-- \]
\[ -Y,x-- \quad -Y,x++ \]
Example: (2,2) to (7,6)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\Delta x = 5, \Delta y = 4$

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Lines: Arbitrary Directions

- **8 different cases**
  - driving (active) axis: $\pm X$ or $\pm Y$
  - Increment/decrement of $y$ or $x$, respectively

\[\begin{align*}
+Y,x- & \quad +Y,x+ \\
-X,y++ & \quad +X,y++ \\
-X,y- & \quad +X,y-- \\
-Y,x- & \quad -Y,x+
\end{align*}\]
**Lines: Some Remarks**

- **Reversed end point order – consistency of pixel choices**
  - $m > 0$: $(d \leq 0)$?
  - $m < 0$: $(d \geq 0)$?

- **Dashed lines**
  - `glLineStipple(Factor, 16-BitSample)`
  - if (BitSample[(n++)/Factor]%16)) then `setpixel(...)`
  - consistent continuation of dashing for line strips and loops

- **Weaker intensity of diagonal lines**
  - Same number of pixel on a larger distance (up to 41%)

- **Subpixel-precision**
  - Clipping, subpixel-coordinates
  - Correct initialization of the decision variable

---

**Thick Lines**

- **Pixel replication**
  - problems with even-numbered widths,
  - varying the intensity of a line as a function of slope

- **The moving pen**
  - for some pen footprints the thickness of a line might change as a function of its slope

- **Filling areas between boundaries**
Lines: Some Remarks

- **Reversed end point order – consistency of pixel choices**
  - $m > 0$: $(d \leq 0)$?
  - $m < 0$: $(d \geq 0)$?

- **Dashed lines**
  - `glLineStipple(Factor, 16-BitSample)`
  - `if (BitSample[(n++/Factor)%16]) then setpixel(...)`
  - Consistent continuation of dashing for line strips and loops

- **Weaker intensity of diagonal lines**
  - Same number of pixel on a larger distance (up to 41%)

- **Subpixel-precision**
  - Clipping, subpixel-coordinates
  - Correct initialization of the decision variable

---

Thick Lines

- **Pixel replication**
  - Problems with even-numbered widths,
  - Varying the intensity of a line as a function of slope

- **The moving pen**
  - For some pen footprints the thickness of a line might change as a function of its slope

- **Filling areas between boundaries**
Line Joints

- End point handling

- Avoid multiple drawings
  - Local bitmap with already set pixels

Drawing Circles

- Square roots and multiplication and trigonometry. Yuck.
- Symmetry. Yay.
- Similar to line scan conversion. Fine.
Line Joints

• End point handling

• Avoid multiple drawings
  – Local bitmap with already set pixels

Drawing Circles

• Square roots and multiplication and trigonometry. Yuck.
• Symmetry. Yay.
• Similar to line scan conversion. Fine.
Midpoint Circle Algorithm

• Look at top right eighth of circle
• \( d = F(x,y) = x^2 + y^2 - R^2 \)
• \( d = 0 \) on circle, < 0 under circle, > 0 over circle

• When have value at \((x,y)\), choose next pixel by calculating \( d = F(x+1, y-.5) \)
• Initial \( d \) derivation, assuming start point is \((0,R)\):
  \[ F(1, R-.5) = 1 + (R^2 - R + .25) - R^2 \]
  \[ = 1.25 - R \]
• Eliminate float:
  Define \( h = d - .25 \) and substitute \( h + .25 \) for \( d \)
  Initialize \( h = 1 - R \) and check for \( h < -.25 \) instead of \( d < 0 \)
  Since \( h \) is always an integer, can just check for \( h < 0 \)

Midpoint Circle Algorithm

• How to get next value of \( d \) incrementally:
  – If didn’t go down one line (same \( y \), next \( x \))
    \[
    d = F(x+2, y-.5) = (x+2)^2 + (y-.5)^2 - R^2
    = x^2 + 4x + 4 + (y-.5)^2 - R^2
    = x^2 + 2x + 1 + (y-.5)^2 - R^2 + 2x + 3
    = (x+1)^2 + (y-.5)^2 - R^2 + (2x + 3)
    = F(x+1, y-.5) + (2x + 3)
    
    So new \( d \) is previous \( d \) plus \((2x + 3)\)
  
  – If did go down one line, similar derivation shows
    new \( d \) is previous \( d \) plus \((2x - 2y + 5)\)
Midpoint Circle Algorithm

- Look at top right eighth of circle
- \( d = F(x,y) = x^2 + y^2 - R^2 \)
- \( d = 0 \) on circle, \( < 0 \) under circle, \( > 0 \) over circle

- When have value at \((x,y)\), choose next pixel by calculating \( d = F(x+1, y-.5) \)
- Initial \( d \) derivation, assuming start point is \((0,R)\):
  \[
  F(1, R-.5) = 1 + (R^2 - R + .25) - R^2 \\
  = 1.25 - R
  \]
- Eliminate float:
  Define \( h = d - .25 \) and substitute \( h + .25 \) for \( d \)
  Initialize \( h = 1 - R \) and check for \( h < -.25 \) instead of \( d < 0 \)
  Since \( h \) is always an integer, can just check for \( h < 0 \)

Midpoint Circle Algorithm

- How to get next value of \( d \) incrementally:
  - If didn’t go down one line (same \( y \), next \( x \))
    \[
    d = F(x+2, y-.5) = (x+2)^2 + (y-.5)^2 - R^2 \\
    = x^2 + 4x + 4 + (y-.5)^2 - R^2 \\
    = x^2 + 2x + 1 + (y-.5)^2 - R^2 + 2x + 3 \\
    = (x+1)^2 + (y-.5)^2 - R^2 + (2x + 3) \\
    = F(x+1, y-.5) + (2x + 3)
    \]
    So new \( d \) is previous \( d \) plus \((2x + 3)\)
  - If did go down one line, similar derivation shows
    new \( d \) is previous \( d \) plus \((2x - 2y + 5)\)
Bresenham: Circle

- Eight different cases here +X, y—

Initialization: \( x=0, y=R \)
\[ F(x,y)=x^2+y^2-R^2 \]
\[ d=F(x+1, y-1/2) \]
- \( d < 0: \)
  \[ d=F(x+2,y-1/2) \]
- \( d > 0: \)
  \[ d=F(x+2,y-3/2) \]
\[ y=y-1 \]
\[ x=x+1 \]

- Eight-way symmetry: only one 45° segment is needed to determine all pixels in a full circle

Second Order Differences

- Not only look at difference between previous \( d \) and current \( d \), also look at difference of the differences
  - Take into account what happened at the last 2 previous pixels
- Use change in \( d \) based on previous choice at each iteration
- Accumulate changes in \( d \) with second order differences
- Simple change to algorithm, produces slightly better results
Second Order Differences

• If aren’t going down one line:
  – If didn’t last time, original change in d was 2x+3
  – New change is 2(x+1) + 3 = original + 2
  – Second order difference = 2
  – If did, original change in d was 2x-2y+5
  – New change is 2(x+1)-2y+5
  – Second order difference is again 2

• If are going down one line:
  – If didn’t last time, original change in d was 2x+3
  – New change is 2(x+1) + 3 = original + 2
  – Second order difference = 2 (independent of y)
  – If did, original change in d was 2x-2y+5
  – New change is 2(x+1)-2(y-1)+5
  – Second order difference is now 4

Bresenham: More General

• Midpoint method works well for ellipses and other implicitly definable curves
  • Parabolas, hyperbolas, ...
**Anti-Aliasing**

- **Supersampling**
  - Calculates solution in virtual screen space
    - higher resolution
  - Downsampling to real screen space
    - Grey values for partially covered pixels
  - Leaves rendering methodology unaltered

**Polygons**

- **Types**
  - triangles
  - trapezoids
  - rectangles
  - convex polygons
  - concave polygons
  - arbitrary polygons
    - holes
    - non-coherent

- **Two approaches**
  - polygon tessellation into triangles
    - edge-flags for internal edges
  - direct scan-conversion
Anti-Aliasing

• **Supersampling**
  - Calculates solution in virtual screen space
    • higher resolution
  - Downsampling to real screen space
    • Grey values for partially covered pixels
  - Leaves rendering methodology unaltered

Polygons

• **Types**
  - triangles
  - trapezoids
  - rectangles
  - convex polygons
  - concave polygons
  - arbitrary polygons
    • holes
    • non-coherent

• **Two approaches**
  - polygon tessellation into triangles
    • edge-flags for internal edges
  - direct scan-conversion
Triangle Filling

- **Possible approaches**
  - first bounding-box, then triangle
  - First triangle, then bounding-box
- **Brute-Force algorithm**

```c
Raster3_box(vertex v[3])
{
    int x, y;
    bbox b;
    bound3(v, &b);
    for (y = b.ymin; y < b.ymax; y++)
        for (x = b.xmin; x < b.xmax; x++)
            if (inside(v, x, y))
                fragment(x, y);
}
```

Filling Polygons

- **Sampling polygons:**
  - When is a pixel inside a polygon?
  - Given a pixel, which polygon does it lie in? *Point location*
- **Polygon representation:**
  - Polygon defined by a list of edges
    - each edge is a pair of vertices
  - All vertices are inside the view volume and map to valid pixels (clipping is behind us now)
  - Let’s assume integer window coordinates
    - to simplify things for now
Triangle Filling

- **Possible approaches**
  - first bounding-box, then triangle
  - First triangle, then bounding-box
- **Brute-Force algorithm**

```c
Raster3_box(vertex v[3])
{
    int x, y;
    bbox b;
    bound3(v, &b);
    for (y = b.ymin; y < b.ymax; y++)
        for (x = b.xmin; x < b.xmax; x++)
            if (inside(v, x, y))
                fragment(x, y);
}
```

Filling Polygons

- **Sampling polygons:**
  - When is a pixel inside a polygon?  
  - Given a pixel, which polygon does it lie in? *Point location*
- **Polygon representation:**
  - Polygon defined by a list of edges
    - each edge is a pair of vertices
  - All vertices are inside the view volume and map to valid pixels
    (clipping is behind us now)
  - Let’s assume integer window coordinates
    - to simplify things for now
Inside-Outside Tests

- **What is the interior of a polygon?**
  - Jordan curve
    - A planar curve homeomorphic to a circle is called Jordan curve. A Jordan curve separates a plane in two connected components, one of which is bounded.
  - Odd-even rule (odd parity rule)
    - counting the number of edge crossings with a ray starting at the queried point P
    - inside, if the number of crossings is odd
  - Non-zero winding number rule
    - signed intersections with a ray
    - inside, if the number is not equal to zero

What is Inside?

- Assume sampling with an array of spikes
- If spike is inside, pixel is inside
What is Inside?

- **Easy for simple polygons** – no self intersections or holes
  - Required by OpenGL; other cases undefined
  - Additionally, OpenGL also requires convex polygons

- **For general polygons, three rules are possible:**
  - **Non-exterior rule**: A point is inside if every ray to infinity intersects the polygon
  - **Non-zero winding number rule**: Draw a ray to infinity that does not hit a vertex. If the number of edges crossing in one direction is not equal to the number crossing the other way, the point is inside
  - **Parity rule**: Draw a ray to infinity and count the number or edges that cross it. If even, the point is outside, if odd, it’s inside

---

Inside/Outside Rules

- **Polygon**
- **Non-exterior**
- **Non-zero Winding No.**
- **Parity**
Inside-Outside Tests

- **What is the interior of a polygon?**
  - Jordan curve
    - A planar curve homeomorphic to a circle is called Jordan curve. A Jordan curve separates a plane in two connected components, one of which is bounded.
  - Odd-even rule (odd parity rule)
    - counting the number of edge crossings with a ray starting at the queried point $P$
    - inside, if the number of crossings is odd
  - Non-zero winding number rule
    - signed intersections with a ray
    - inside, if the number is not equal to zero

---

What is Inside?

- Assume sampling with an array of spikes
- If spike is inside, pixel is inside
What is Inside?

- Assume sampling with an array of spikes
- If spike is inside, pixel is inside

```
  o o o o o o o o
  o ● ● ● ● ● ● o
  o ● ● ● ● ● ● o
  o ● ● ● ● ● ● o
  o ● ● ● ● ● ● o
  o ● ● ● ● ● ● o
  o ● ● ● ● ● ● o
  o o o o o o o o
```

Ambiguous Cases

- Ambiguous case: What if a pixel lies on an edge?
  - Problem because if two polygons share a common edge, we don't want pixels on the edge to belong to both
  - Ambiguity would lead to different results if the drawing order was different
- Rule: if \((x+\epsilon, y+\epsilon)\) is in, \((x, y)\) is in
- What if a pixel is on a vertex? Does our rule still work?
What is Inside?

- Assume sampling with an array of spikes
- If spike is inside, pixel is inside

Ambiguous Cases

- Ambiguous case: What if a pixel lies on an edge?
  - Problem because if two polygons share a common edge, we don't want pixels on the edge to belong to both
  - Ambiguity would lead to different results if the drawing order was different
- Rule: if \((x+\epsilon, y+\epsilon)\) is in, \((x, y)\) is in
- What if a pixel is on a vertex? Does our rule still work?
Ambiguous Case I

- **Rule:**
  - On edge?
    - If \((x+\varepsilon, y+\varepsilon)\) is in, pixel is in
  - Which pixels are colored?
    - OpenGL origin convention!

![Diagram of Ambiguous Case I](image)

- **Rule:**
  - Keep left and bottom edges
  - Assuming \(y\) increases in the up direction
  - If rectangles meet at an edge, how often is the edge pixel drawn?

![Diagram of Ambiguous Case I](image)
Ambiguous Case I

• Rule:
  – On edge?
    If \((x+\varepsilon, y+\varepsilon)\) is in, pixel is in
  – Which pixels are colored?
    • OpenGL origin convention!

    \[
    \begin{array}{cccccccc}
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \end{array}
    \]

Ambiguous Case I

• Rule:
  – Keep left and bottom edges
  – Assuming \(y\) increases in the up direction
  – If rectangles meet at an edge, how often is the edge pixel drawn?

    \[
    \begin{array}{cccccccc}
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
    \end{array}
    \]
Ambiguous Case II

- **Rule:**
  - On edge?
    - If \((x+\varepsilon, y+\varepsilon)\) is in, pixel is in
  - What happens for diagonal edges?
Ambiguous Case II

- **Rule:**
  - On edge?
    - If \((x+\varepsilon, y+\varepsilon)\) is in, pixel is in
  - What happens for diagonal edges?
Really Ambiguous

- We will accept ambiguity in such cases
  - The center pixel may end up colored by one of two polygons in this case
  - Which two?
- Might be solvable using \((x+\epsilon, y+2\epsilon^2)\)?
  - Arbitrarily small, irrational slope
  - Rule stays the same

Scanline Conversion

- Fill pixel area inside polygon edges
- Exploiting Coherence when filling a polygon
  - Several contiguous pixels along a row tend to be in the polygon - a span of pixels
    - Scanline coherence
  - Consider whole spans, not individual pixels
  - Pixel number and position don’t vary much from one span to the next
    - Edge coherence
  - Incrementally update span endpoints
Really Ambiguous

• We will accept ambiguity in such cases
  – The center pixel may end up colored by one of two polygons in this case
  – Which two?

• Might be solvable using \((x+\epsilon, y+\epsilon^2)\) (?)
  – Arbitrarily small, irrational slope
  – Rule stays the same

Scanline Conversion

• Fill pixel area inside polygon edges
• Exploiting Coherence when filling a polygon
  – Several contiguous pixels along a row tend to be in the polygon - a span of pixels
    • Scanline coherence
  – Consider whole spans, not individual pixels
    – Pixel number and position don’t vary much from one span to the next
    • Edge coherence
  – Incrementally update span endpoints
Sweep Fill Algorithms

- **Algorithmic issues:**
  - Reduce to filling many spans
  - Which edges define the span of pixels to fill?
  - How do you update these edges when moving from span to span?
  - What happens when you cross a vertex?

---

Spans

- Process - fill the bottom horizontal span of pixels; move up and keep filling
- Have $x_{min}$, $x_{max}$ for each span
- Define:
  - floor($x$): largest integer < $x$
  - ceiling($x$): smallest integer >= $x$
- Fill from ceiling($x_{min}$) up to floor($x_{max}$)
- Consistent with convention
Algorithm

- For each row in the polygon:
  - Throw away irrelevant edges
  - Obtain newly relevant edges
  - Fill span
  - Update current edges

- Issues:
  - How do we update existing edges?
  - When is an edge relevant/irrelevant?

- All can be resolved by referring to our convention about what polygon the pixel belongs to

Updating Edges

- Each edge is a line of the form:
  \[ y = mx + c \quad \text{or} \quad x = ym' - c' \]

- Next row is:
  \[ x_{i+1} = (y + 1)m' - c' = x_i + m' \]

- So, each current edge can have its x position updated by adding a constant stored with the edge

- Other values may also be updated, such as depth or color information