Scan Conversion: Rasterization

• Deals with converting objects defined by real coordinates into the frame buffer
• In frame buffer, objects exist in terms of discrete pixels
• Basic operations that are involved:
  – Scan conversion:
    * Converting from description in terms of reals into pixels
  – Filling:
    * Filling interior of polygon
  – Clipping:
    * Eliminating parts that lie outside of a region defined by a polygon
  – Antialiasing:
    * Smoothing effects that arise due to discrete representation
Scan Conversion: Straight Lines - Intro

• Process identifies those pixels that best approximate straight line

• If $-1 \leq \text{slope} \leq 1$, want 1 pixel per column
  
  – Otherwise, want 1 pixel per row

• Lines should have same brightness regardless of slope

• Algorithm should be able to deal with
  
  – Lines thicker than 1 pixel
  – Lines with shaped ends
  – Etc.

• Device characteristics may affect display:
  
  – Pixels may overlap (a)
  – Pixels may have space between them (b)
  – Inter-pixel space may differ vertically and horizontally (c)
  – Pixels may be centered at grid intersection, or centered within grid box (d)

• Following discussions assume:
  
  – Pixels centered at grid intersection
  – Lines 1 pixel thick
  – Monochrome display device

  $|\text{slope}| < 1$
Scan Conversion: Straight Lines - Basic Approaches

1. Brute force
   - Use slope-intercept representation:
     \[ y_i = mx_i + b \]
   - For each \( x_0 \leq x_i \leq x_n \), calculate \( y_i \)
   - Want pixel closest to absolute line, so point is \( P_i(x_i, \text{round}(y_i)) \)
   - Problems:
     (a) Uses 2 floating point operations
     (b) Uses rounding

2. Incremental algorithm
   - Given
     (a) \( y_i = mx_i + b \)
     (b) \( y_{i+1} = mx_{i+1} + b \)
     (c) \( x_{i+1} = x_i + \Delta x \)
   - Then
     \[
     y_{i+1} = m(x_i + \Delta x) + b \\
     = mx_i + b + m\Delta x \\
     = y_i + m\Delta x
     \]
   - Since \( \Delta x = 1 \),
     \[ y_{i+1} = y_i + m \]
   - Requires only floating point addition and floor
   - This type of algorithm is called incremental because new value is based on increment of previous value in series
   - Called a digital differential analyzer
   - Problems:
     (a) Limited precision results in accumulated error in long spans
     (b) Uses rounding
   - Horizontal, vertical, and ±45° lines are special cases
   - If \( |\text{slope}| > 1 \), use
     \[
     \Delta y = 1, \\
     \Delta x = 1/m
     \]
Scan Conversion: Straight Lines - Midpoint (Bresenham’s) Algorithm

- Algorithm uses integer arithmetic exclusively
- Applicable to lines, circles, arbitrary conics
- In following, $0 \leq m \leq 1$
  - Lines with other slopes symmetric

- Let $(x_p, y_p)$ be current pixel
- Next will be at either $(x_p + 1, y_p)$, or $(x_p + 1, y_p + 1)$
- Call these 2 points $E$ and $NE$, respectively
- Let $M$ be midpoint of $NE - E$
- Let $Q$ be intersection of line with $x = x_p + 1$
- Whichever side of $M$ that $Q$ falls determines whether the next pixel should be $NE$ or $E$
- Implicit equation of a line:

$$y = mx + b = x \frac{dy}{dx} + b$$  \hspace{1cm} (1)

Multiplying through by $dx$ and moving all terms to the same side,

$$ydx = xdy + bdx$$
$$0 = xdy - ydx + bdx$$  \hspace{1cm} (2)

This has form

$$f(x, y) = ax + by + c = 0$$  \hspace{1cm} (3)

, where

$$a = dy$$
$$b = -dx$$
$$c = bdx$$

Equation 3 (or 2) is called the *implicit form* of a line
Scan Conversion: Straight Lines - Midpoint (Bresenham’s) Algorithm (2)

\( f(x, y) = 0 \) for points on the line
\( f(x, y) > 0 \) for points below the line
\( f(x, y) < 0 \) for points above the line

- To determine \( M \) in relation to line, calculate

\[
\begin{align*}
  f(m) &= f(x_p + 1, y_p + 1/2) \\
  &= a(x_p + 1) + b(y_p + 1/2) + c \\
  &= d
\end{align*}
\]  

(4)

1. If \( d > 0 \) (\( M \) below line), choose \( NE \)
2. If \( d < 0 \) (\( M \) above line), choose \( E \)
3. If \( d = 0 \) (\( M \) on line), choose either (will arbitrarily choose \( E \))

- Next value of \( d \) depends on point last selected
  1. If \( E \),

\[
\begin{align*}
  d_{new} &= f(x_p + 2, y_p + 1/2) \\
  &= a(x_p + 2) + b(y_p + 1/2) + c
\end{align*}
\]  

(5)

Using equations 4 and 7

\[
\begin{align*}
  d_{new} - d_{old} &= a = dy = \Delta_E \\
  \text{and thus} \\
  d_{new} &= d_{old} + dy
\end{align*}
\]  

(6)

(7)

2. If \( NE \), (using a similar formulation),

\[
\begin{align*}
  d_{new} &= f(x_p + 2, y_p + 3/2) \\
  &= a(x_p + 2) + b(y_p + 3/2) + c
\end{align*}
\]  

(8)

\[
\begin{align*}
  d_{new} - d_{old} &= a + b = dy - dx = \Delta_{NE} \\
  \text{and} \\
  d_{new} &= d_{old} + dy - dx
\end{align*}
\]  

(9)

(10)
Scan Conversion: Straight Lines - Midpoint (Bresenham’s) Algorithm (3)

- To determine an initial value for $d$
  - Start of line at $(x_0, y_0)$
  - First $M$ at $(x_0 + 1, y_0 + 1/2)$

Substituting $(x_0 + 1, y_0 + 1/2)$ in equation 3

$$f(x_0 + 1, y_0 + 1/2) = f(x_0, y_0) + a + b/2 \quad (11)$$

And since $f(x_0, y_0) = 0$

$$d_{start} = f(x_0 + 1, y_0 + 1/2)$$
$$= a + b/2$$
$$= dy - dx/2 \quad (12)$$

- In performing computations, eliminate fraction $1/2$ by multiplying by 2
  - Since only concerned with sign of $d_{new}$, this has no effect on results
Scan Conversion: Straight Lines - Issues

1. End point order
   - Line appearance
     - Should be independent of initial order of end points
     - The only point in the algorithm where there is ambiguity is when \( M = Q \)
     - To be consistent, choose SW when moving in opposite direction and \( M = Q \)
   - Stylized lines
     - Write mask usually anchored at start point, independent of direction
     - Pattern continued as pass through other points (polylines)

2. Clipping
   - When line clipped, start point may not be true endpoint (i.e., is pixel approximation to point on true line)
   - 2 situations to consider:

(a) Clip against vertical (e.g., left) edge
   - Intersection has integer \( x \) but float \( y \)
   - Pixel at \( (x_{\text{min}}, \text{round}(mx_{\text{min}} + b)) \) is same as for unclipped line
   - \( d \) initialized to \( M \), not using \( d_{\text{start}} \)

(b) Clip against horizontal (e.g., bottom) edge
   - When intersect at \( y = y_{\text{min}} \), nearest \( x \) may not be correct value
   - True \( x \) is where line intersects \( y = y_{\text{min}} - 1/2 \)
   - Find this point of intersection, and round \( x \) up
Scan Conversion: Straight Lines - Issues (2)

• Intensity and slope
  – A diagonal line from \((x_0, y_i)\) to \((x_1, y_j)\) and horizontal line from \((x_0, y_k)\) to \((x_1, y_l)\) have the same number of pixels
  – Diagonal will appear less intense because is longer and pixel density lower
  – As increase the number of bit planes, can remedy the situation

• Multiline primitives
  – Polylines
    * Draw one segment at a time
  – Polygons
    * Problematic if do one segment at a time
  – Shared vertices
    * Must be written 1X
    * If using XOR mode, writing 2X changes color
    * Produces 2X intensity, depending on medium (film, plotter)
Scan Conversion: Circles - Basic Approaches

1. Basic circle formula
   - $r^2 = x^2 + y^2$
   - $y = \sqrt{r^2 - x^2}$
   - Problems:
     (a) Uses squares and square root
     (b) Non-uniform density
       - Points thinner when $x \approx r$

2. Use trig functions
   - $x = r\cos \theta, y = r\sin \theta, 0 \leq \theta \leq 90$
   - Provides uniform density
   - Problems:
     (a) Uses trig functions and floating point operations

3. Can use 8-way symmetry (e.g., $0 \leq \theta \leq 45$) to improve efficiency
Scan Conversion: Circles - Midpoint (Bresenham’s) Algorithm

- Uses same strategy as midpoint line algorithm

- Consider 45° arc from $x = 0$ to $x = y = \frac{r}{\sqrt{2}}$

- Let $(x_p, y_p)$ be current pixel

- Next pixel will be at either $(x_p + 1, y_p)$, or $(x_p + 1, y_p - 1)$
  - Call these 2 points $E$ and $SE$, respectively

- Let $M$ be midpoint of $E - SE$

- Let $Q$ be intersection of line with $x = x_p + 1$

- The side of $M$ that $Q$ falls on determines whether the next pixel should be $E$ or $SE$

\[
\begin{align*}
 r^2 &= x^2 + y^2 \\
 0 &= x^2 + y^2 - r^2 \\
 &= f(x, y)
\end{align*}
\] (13)

1. $f(x, y) = 0$ for points on the circle
2. $f(x, y) > 0$ for points outside the circle
3. $f(x, y) < 0$ for points inside the circle
Scan Conversion: Circles - Midpoint (Bresenham’s) Algorithm (2)

- To determine $M$ in relation to line, calculate (using equation 13)

  \[ f(m) = d_{\text{old}} = f(x_p + 1, y_p - 1/2) = (x_p + 1)^2 + (y_p - 1/2)^2 - r^2 \]  

(14)

1. If $d_{\text{old}} < 0$ ($M$ inside circle), choose $E$

   \[ d_{\text{new}} = f(x_p + 2, y_p - 1/2) = (x_p + 2)^2 + (y_p - 1/2)^2 - r^2 = ((x_p + 1) + 1)^2 + (y_p - 1/2)^2 - r^2 = (x_p + 1)^2 + 2(x_p + 1) + 1 + (y_p - 1/2)^2 - r^2 = (x_p + 1)^2 + (y_p - 1/2)^2 - r^2 + 2(x_p + 1) + 1 = d_{\text{old}} + (2x_p + 3) \]  

(15)

So

\[ \Delta_E = 2x_p + 3 \]  

(16)

2. If $d_{\text{old}} \geq 0$ ($M$ on line), choose $SE$

   \[ d_{\text{new}} = f(x_p + 2, y_p - 3/2) = (x_p + 2)^2 + (y_p - 3/2)^2 - r^2 = ((x_p + 1) + 1)^2 + ((y_p - 1/2) - 1)^2 - r^2 = (x_p + 1)^2 + (y_p - 1/2)^2 - r^2 + (2x_p + 2 + 1) + (-2y_p + 1 + 1) = d_{\text{old}} + (2x_p - 2y_p + 5) \]  

(17)

So

\[ \Delta_{SE} = 2x_p - 2y_p + 5 \]  

(18)

3. $\Delta_E, \Delta_{SE}$ not constant

   - Linear wrt previous $P(x,y)$ value
   - $P$ called **point of evaluation**
Scan Conversion: Circles - Midpoint (Bresenham’s) Algorithm (3)

- Need to determine an initial value for \( d \)
  - Initial point at \((0, r)\)
  - Initial \( M \) at \((1, r - 1/2)\)

\[
d_{\text{init}} = f(1, r - 1/2) \\
= 1 + (r^2 - r + 1/4) - r^2 \\
= 5/4 - r \quad (19)
\]

- Do not want fraction \( 5/4 \)
  * Let \( h = d - 1/4 \)
  * Initialization becomes \( h = 1 - r \) \( (20) \)

  * If \( d < 0 \Rightarrow h < -1/4 \)
  * But since \( h \) is initialized to an int, and incremented by int value, can simply check if \( h < 0 \)

- Incremental calculations:

1. Case \( E \)
   \((x_p, y_p) \rightarrow (x_p + 1, y_p)\)

   Using equation 16

   \[
   \Delta E_{\text{old}} = 2x_p + 3 \\
   \Delta E_{\text{new}} = 2(x_p + 1) + 3 \\
   \Delta E_{\text{new}} - \Delta E_{\text{old}} = 2 \quad (21)
   \]

   Using equation 18

   \[
   \Delta S E_{\text{old}} = 2x_p - 2y_p + 5 \\
   \Delta S E_{\text{new}} = 2(x_p + 1) - 2y_p + 5 \\
   \Delta S E_{\text{new}} - \Delta S E_{\text{old}} = 2 \quad (22)
   \]
Scan Conversion: Circles - Midpoint (Bresenham’s) Algorithm (4)

2. Case $SE$

$(x_p, y_p) \rightarrow (x_p + 1, y_p - 1)$

\[
\begin{align*}
\Delta E_{old} &= 2x_p + 3 \\
\Delta E_{new} &= 2(x_p + 1) + 3 \\
\Delta E_{new} - \Delta E_{old} &= 2
\end{align*}
\]  

\[
\begin{align*}
\Delta SE_{old} &= 2x_p - 2y_p + 5 \\
\Delta SE_{new} &= 2(x_p + 1) - 2(y_p - 1) + 5 \\
\Delta SE_{new} - \Delta SE_{old} &= 4
\end{align*}
\]  

- Algorithm:

1. Choose pixel based on sign of $d$
2. Update $d$ using $\Delta E$ or $\Delta SE$ using equations 16 and 18
3. Update $\Delta E$, $\Delta SE$ using equations 21, 22, 23, 24,
4. Repeat at new pixel
Scan Conversion: Thick Lines

• Several factors that affect quality of thick \((w > 1 \text{ pixel})\) lines

  1. What do ends of lines look like?
  2. What happens where lines meet?
  3. How do line style and pen style interact?

• General approaches to implementing thick lines

  1. Replicating pixels
     - If \(-1 < \text{slope} < 1\)
       * Use vertical columns of pixels equal to width of line
     - For other slopes
       * Use horizontal rows of pixels equal to width of line
Scan Conversion: Thick Lines (2)

− Pros
  (a) Easy to implement
− Cons
  (a) Line ends always horizontal or vertical
  (b) Gaps appear where lines meet at an angle
  (c) Misses pixels when slope shifts across $\pm 1$ (e.g., for curves)
  (d) Horizontal and vertical lines thicker than others
    * For $m = \pm 1$, line width = $w/\sqrt{2}$
  (e) For even $w$, cannot center on true line
− Best for not-too-thick lines

2. Moving pen
− Simulate drawing with a real pen/brush
− Pen has a profile
  * Generally assume rectangular
  * Sometimes called a footprint
− Line drawn by moving pen along true line
  * Point of contact is center or corner of profile
− Two additional characteristics are applicable to this approach:
  (a) Orientation of brush (for non-circular shapes)
  (b) Shape of profile
Scan Conversion: Thick Lines (3)

- If brush has fixed orientation

* Cons
  (a) Ends vertical or horizontal
  (b) Thickness dependent on slope
    - Thickest when \( m = \pm 1 \)
- If brush oriented wrt line tangent

* Pros
  (a) Line width uniform
  (b) Ends square with line
Scan Conversion: Thick Lines (4)

- Additional negatives
  (a) If trace out line, pixels written multiple times
  (b) For non-rectangular profiles, need to mask bits outside of shape
    * (Assuming rectangular bitmapped profile)
    * Both of these remedied by representing profile as spans of pixels

3. Filling

- Lines:
  * Draw 2 lines of same length and fill
  * Lines drawn at \( \pm w/2 \)

- Circles:
  * Draw 2 circles with same center
  * Drawn at \( r \pm w/2 \)

- Ellipses:
  * Two ellipses with same curvature do not share foci
  * Fudge using
    (a) Inner foci of \( a - w/2, b - w/2 \)
    (b) Outer foci of \( a + w/2, b + w/2 \)

- Pros
  (a) Uniform width
  (b) Ends square with line

4. Think polylines

- Approximate line with straight line segments
- Pros
  (a) Have existing efficient algorithms for scan conversion and clipping
- Cons
  (a) Need many segments for sections of high curvature
• Area-defining primitives
  – To handle borders of 2D primitives
    1. Draw border centered on true border using above techniques
      * Cons
        (a) Increases area of primitive
        (b) Problem as above with even widths
    2. Use true border as outer perimeter
      * Inner border is \( w \) pixels inside
      * Pros
        (a) No problem with even widths
        (b) Area unaffected by border width
      * Cons
        (a) Primitive ”shrinks”
          · Center of primitive perimeter moved inward by \( w/2 \)

• Line style and pen style
  – If use typical bit mask for line stippling, have problem:
    * Length of dash dependent on slope
  – Must represent dash as line segment of fixed length
    * Scan convert each segment
  – For thick primitives, use filled rectangles
Scan Conversion: Antialiasing - Increased Resolution

- The greater the resolution, the less pronounced the aliasing
- Problems with this approach:
  1. Display device has max resolution
  2. Greater resolution increases
     (a) Memory requirements
     (b) Bandwidth
     (c) Scan conversion time
  3. Does not eliminate aliasing
     - Simply scales it down
Scan Conversion: Antialiasing - Unweighted Area Sampling

- Treats pixels as though they were rectangles
  - Treat display as a non-overlapping grid of pixels
- Non-vertical and non-horizontal lines will intersect multiple pixels per row
- Strategy is to have all intersected pixels contribute some amount towards line’s intensity
- To determine per cent of overlap, subdivide pixels into subpixels
- Unweighted sampling has these characteristics:
  1. Intensity of intersected pixels decreases as distance between center of pixel and line edge increases
  2. Pixels that do not intersect line are unaffected
  3. Pixels with equal areas of intersection contribute equal amounts of intensity
     - The amount is independent of the distance between the pixel center and line edge
Scan Conversion: Antialiasing - Weighted Area Sampling

- This approach based on modifications of characteristics of unweighted sampling:
  1. Intensity of intersected pixels decreases as distance between center of pixel and line edge increases
     - Still holds
  2. Pixels that do not intersect line are unaffected
     - Still holds in general, but
       (a) Intersected pixel influences neighboring pixels
       (b) Radius of influence \( r > 1/2w \) (or \( 1/2h \)) of pixel
  3. Pixels with equal areas of intersection contribute equal amounts of intensity
     - The amount is now dependent on the distance between the pixel center and line edge

- Pixel’s contribution to intensity based on weight function \( W(x, y) \)
  - For area \( dA \), height of \( W(x, y) \) at \( x, y \) is contribution of point \( (x, y) \) to intensity
  - Total contribution is the volume \( (W_s) \) over the area of overlap
    * I.e., \( W_s = \int W(x, y), \) where \( 0 \leq W_s \leq 1 \)
    * Intensity \( I = I_{max} \times W_s \)
    - \( W(x, y) \) called a filter function

- For unweighted sampling, \( W(x, y) \) is a box:
  - Called a box filter
  - Height of box is normalized to 1
  - Volume of box is 1
Scan Conversion: Antialiasing - Weighted Area Sampling (2)

• For weighted sampling
  – Want function that decreases as move away from center
  – Simplest case is linear function
    * Volume is a cone

* Base of cone called support of filter
* Research has shown that a good choice for radius of base = 1
  · This insures that no parts of the grid are uncovered
* Height of cone chosen so that volume is 1

– Differences from unweighted case:
  * More pixels affected
  * Pixel entirely covered will not have 100% intensity
  * Horizontal and vertical lines affect more than one pixel per row/column
Scan Conversion: Antialiasing - Gupta-Sproull Algorithm

- Applied during scan conversion
- References a stored table of filter function values based on distance from pixel center

![Image showing the calculation of D]

- Lookup function is $\text{filter}(D, t)$, where
  - $D$ is distance from line to pixel
  - $t$ is constant for line of given thickness
- Works in conjunction with midpoint line scan conversion algorithm
  - In addition to id’ing next pixel, determine
    - Intensity of next pixel
    - Intensity of vertical neighbors of next pixel
- Details
  - From the diagram,
    $$D = v \cos \phi = \frac{v dx}{\sqrt{dx^2 + dy^2}} \tag{25}$$
  - Since $v$ is signed, call filter with $|D|$}
  - Want to calculate $v$ incrementally
  - Do so indirectly based on $d = F(M) = F(x_p + 1, y_p + 1/2)$
From the derivation of the midpoint algorithm earlier (equation 3),

$$F(x, y) = 2(ax + by + c) = 0$$

(26)

where $a = dy$, $b = -dx$, and $c = Bdx$.

(Note: The multiplicative factor 2 does not appear in the earlier equation.)

Solving for $y$ in equation 26 above:

$$y = \frac{ax + c}{-b}$$

(27)

– For pixel $E$ (see above diagram)

$$x = x_p + 1$$

$$y = y_p$$

$$v = y - y_p$$

(28)

Substituting 27 for $y$ in equation 28 gives:

$$v = \frac{a(x_p + 1) + c}{-b} - y_p$$

(29)

* Multiplying through by $b$ and substituting $-dx$ for $b$ in equation 29:

$$-bv = a(x_p + 1) + by_p + c \ (c.f \ equation \ 26 \ above)$$

$$= F(x_p + 1, y_p)/2$$

$$vdx = F(x_p + 1, y_p)/2$$

(30)

* To represent $vdx$ in terms of $d$

$$2vdx = F(x_p + 1, y_p)$$

$$= 2a(x_p + 1) + 2by_p + 2c$$

$$= 2a(x_p + 1) + 2b(y_p + 1/2) - 2b/2 + 2c$$

$$= d + dx$$

(31)

(In the above, $2a(x_p + 1) + 2b(y_p + 1/2) + 2c = d$, and $-2b/2 = dx$)

* Therefore, using equations 25 and 31,

$$D = \frac{d + dx}{2\sqrt{x^2 + y^2}}$$

(32)

which represents the distance of pixel $E$ from the line.
Scan Conversion: Antialiasing - Gupta-Sproull Algorithm (3)

For the pixels below and above this pixel, substituting $1 - v$ and $1 + v$ (see diagram), respectively for $v$ in equation 31, the numerator becomes

- For $y_p + 1$, numerator $= 2(1 - v)dx = 2dx - 2vdx$
- For $y_p - 1$, numerator $= 2(1 + v)dx = 2dx + 2vdx$

For pixel $NE$

\[ 2vdx = F(x_p + 1, y_p + 1) \]
\[ = 2a(x_p + 1) + 2b(y_p + 1/2) + 2b/2 + 2c \]
\[ = d - dx \] (33)

- For $y_p + 2$, numerator $= 2(1 - v)dx = 2dx - 2vdx$
- For $y_p$, numerator $= 2(1 + v)dx = 2dx + 2vdx$
void antiAliasedMidpoint (int x0, y0, x1, y1)
{
    //Variables for midpoint
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx;
    int incE = 2 * dy;
    int incNE = 2 * (dy - dx);

    //Variables for Gupta-Sproull
    int twoVDx = 0;
    double invDenom = 1.0 / (2.0 * sqrt(dx * dx + dy * dy));
    double twoDxTimesInvDenom = 2.0 * dx * invDenom; //Initial distance to (x0, y0 + 1) and (x0, y0 - 1)

    int x = x0;
    int y = y0;

    //Third intensity of initial pixels
    //Third argument is distance to pixel from line
    intensifyPixel(x, y, 0);  //
    intensifyPixel(x, y + 1, twoDxTimesInvDenom);
    intensifyPixel(x, y - 1, twoDxTimesInvDenom);

    //Main loop with updates of variables
    while (x < x1) {
        if (d < 0) {
            twoVDx = d + dx;
            d += incrE;
            x++;
        }
        else {
            twoVDx = d - dx;
            d += incrNE;
            x++;
            y++;
        }
        intensifyPixel(x, y, twoVDx * invDenom);
        intensifyPixel(x, y + 1, twoDxTimesInvDenom - twoVDx * invDenom);
        intensifyPixel(x, y - 1, twoDxTimesInvDenom - twoVDx * invDenom);
    }
}