Overview

- Drawing Lines and Circles
- Filling Algorithms
  - Boundary, Floodfill
  - Scanline Algorithm
- Clipping Algorithms
  - Lines - Cohen-Sutherland, Liang-Barsky Algorithm
  - Polygon - Sutherland-Hodgman
- Anti-aliasing
“The process of converting geometric primitives into their discrete approximations”

**Scan Conversion:**

⇒ Approximate geometric primitives (analytically defined) by a set of pixels, stored in frame-buffer or memory.

**Clipping:**

⇒ Only sections of primitives determined to be within a clipping region is drawn (scan-converted).

**Speed:**

⇒ Algorithms must be very efficient. Why?
Drawing Lines

Given the line end points \((x_0, y_0)\) and \((x_1, y_1)\)

Line Equation

\[ y = mx + b \]

\(m = \text{Slope}, \ b = \text{y intercept}\)

To determine

the sequence of points between \((x_0, y_0)\) and \((x_1, y_1)\) on the raster grid
(end points are in screen coordinates).
Brute Force Algorithm

\[
\text{for } i = x_0 \text{ to } i = x_1 \\
\{ \\
y_i = mx_i + b \\
PLOT (x_i, \text{ROUND} ( y_i )) \\
\}
\]

**Inefficient:** Involves multiply and rounding.
DDA Algorithm

Features

- Exploits the fact that the line equation is a linear function that needs to be evaluated over a regular lattice.
- **Constant** increments in both dimensions to obtain successive points along the line eliminates multiplies in the inner loop.

\[
\begin{align*}
y_{i+1} &= mx_{i+1} + b \\
&= m(x_i + \Delta x) + b \\
&= (mx_i + b) + m\Delta x \\
&= y_i + m(1)
\end{align*}
\]
DDA Algorithm

Algorithm:

\[ m = \frac{(y_1 - y_0)}{(x_1 - x_0)} \]

\textbf{for} \; x_i = x_0 \; \textbf{to} \; x_i = x_1 

\{ 
\begin{align*}
\text{plot} \; (x_i, \; \text{Round}(y_i)) \\
y_i &= y_i + m \\
x_i &= x_i + 1 
\end{align*} \\
\}

\textbf{Inefficient}, involves division to compute slope, and rounding.
MidPoint Line Algorithm

For lines and circles, same as Bresenham’s algorithm.

Features:

• Uses only integer operations.
• Uses incremental calculations to determine successive pixels.

Idea

• Determine location of mid point, $M_i$, with respect to the line.
• Mid points $M_i$ are computed using incremental calculations.
MidPoint Line Algorithm (contd)

\[ y = \left( \frac{dy}{dx} \right)'x + B \]

\[ dx.y = dy.x + B.dx, \text{ or} \]

\[ F(x, y) = dy.x - dx.y + B.dx = 0 \]

\[ = a.x + b.y + c = 0, \quad (a = dy, b = -dx, c = B .dx) \]

\[ d = F(x, y) = 0, \quad x, y \text{ on the line} \]

\[ > 0, \quad x, y \text{ below the line} \]

\[ < 0, \quad x, y \text{ above the line} \]
MidPoint Line Algorithm (contd)

Strategy:

- The sign of $d = F(x, y)$, the decision variable, will determine whether $A$ or $B$ is chosen as the next pixel on the line.
- “Insert $M_i$ into $F(x, y)$ and check the sign of $F$.”
Efficient calculation of $d$

Case 1: A is chosen

\[
M_i = F(x_p + 1, y_p + 1/2) \\
M_{i+1} = F(x_p + 2, y_p + 3/2) \\
M_{i+1} - M_i = F(x_p + 2, y_p + 3/2) - F(x_p + 1, y_p + 1/2) \\
= \{a(x_p + 2) + b(y_p + 3/2) + c\} - \{a(x_p + 1) + b(y_p + 1/2) + c\} \\
= (a + b) \\
M_{i+1} = M_i + (a + b) = M_i + dy - dx
\]
Efficient calculation of $d$ (contd)

Case 2: B is chosen

\[
\begin{align*}
M_{i+1} &= F(x_p + 2, y_p + 1/2) \\
M_{i+1} - M_i &= F(x_p + 2, y_p + 1/2) - F(x_p + 1, y_p + 1/2) \\
&= a \\
M_{i+1} &= M_i + (a) \\
&= M_i + dy
\end{align*}
\]
Initialization

\[ d = F(x_0 + 1, y_0 + 1/2) \]
\[ = a(x_0 + 1) + b(y_0 + 1/2) + c \]
\[ = (ax_0 + by_0 + c) + a + b/2 \]
\[ = a + b/2 \]

or

\[ d = 2a + b \] (only sign of d is important)

Hence

\[ d_1 = 2(a + b) = 2(dy - dx) \]
\[ d_2 = 2a = 2(dy) \]
Final Algorithm (Quadrant 1 only)

\[ d = 2 \, dy - \, dx \]
\[ d1 = 2 \, dy - 2 \, dx \]
\[ d2 = 2 \, dy \]

\[
\text{for } x = x_0 \text{ to } x_1 \text{ by 1 }
\{
\text{if } (d \leq 0 )
\quad d = d + d_1
\text{else}
\quad \{
\quad d = d + d_2
\quad y = y + 1
\quad \}
\text{plot } (x, y)
\}
\]
Rotational symmetry makes it sufficient to scanconvert only 1 octant of a circle.
MidPoint Circle Algorithm

\[ F(x, y) = x^2 + y^2 - R^2 = 0 \]  
(circle centered at the origin)

\[ d = F(x, y) = 0, \quad x, y \text{ on the circle} \]
\[ > 0, \quad x, y \text{ outside the circle} \]
\[ < 0, \quad x, y \text{ inside the circle} \]
MidPoint Circle Algorithm (contd)

\[ F(M_i) = F(x_p + 1, y_p - 1/2) \]
\[ = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2 \]
\[ F(M_{i+1}) = F(x_p + 2, y_p - 1/2) \text{ (East Neighbor)} \]
\[ = (x_p + 2)^2 + (y_p - 1/2)^2 - R^2 \]
\[ F(M'_{i+1}) = F(x_p + 2, y_p - 3/2) \text{ (S. East Neighbor)} \]
\[ = (x_p + 2)^2 + (y_p - 3/2)^2 - R^2 \]
MidPoint Circle Algorithm (contd)

\[ d_{old} = F(M_i) \]
\[ d_{new} = F(M_{i+1}), \text{ A chosen} \]
\[ = F(M'_{i+1}), \text{ B chosen} \]
\[ d_{incr} = d_{new} - d_{old} \]
\[ = 2x_p + 3, \text{ A chosen.} \]
\[ = 2(x_p - y_p) + 5, \text{ B chosen.} \]
Midpoint Circle Alg.: Initialization

\[(x_0, y_0) = (0, R)\]
\[F(1, R - 1/2) = 1 + R^2 + 1/4 - R - R^2\]
\[= 5/4 - R\]

Efficiency

The algorithm can be made more efficient (Refer Foley/van Dam).
Fill Algorithms

4-Connected
8-Connected

4 - Connected
8 Connected
FloodFill Algorithm

Interior pixels are defined by a unique color.

FloodFill \((x, y, \text{InteriorColor}, \text{NewColor})\)

\[
\begin{align*}
\text{if } & \text{pixel\_color} (x, y) \text{ EQUALS InteriorColor} \\
& \{ \\
& \quad \text{set\_pixel} (x, y, \text{NewColor}) \\
& \quad \text{FloodFill} (x-1, y, \text{InteriorColor}, \text{NewColor}) \\
& \quad \text{FloodFill} (x+1, y, \text{InteriorColor}, \text{NewColor}) \\
& \quad \text{FloodFill} (x, y+1, \text{InteriorColor}, \text{NewColor}) \\
& \quad \text{FloodFill} (x, y-1, \text{InteriorColor}, \text{NewColor}) \\
& \} \\
\} 
\]
Boundary Fill Algorithm

Boundary pixels are defined by a unique color.

BoundaryFill (x, y, BoundaryColor, NewColor)
{
    if (pixel_color (x, y) NOT_EQUAL_TO BoundaryColor) AND
        (pixel_color (x, y) NOT_EQUAL_TO NewColor)
    {
        set_pixel (x, y, NewColor)
        BoundaryFill (x-1, y, BoundaryColor, NewColor)
        BoundaryFill (x+1, y, BoundaryColor, NewColor)
        BoundaryFill (x, y+1, BoundaryColor, NewColor)
        BoundaryFill (x, y-1, BoundaryColor, NewColor)
    }
}
The Scanline Polygon Fill Algorithm

Approach

- Exploit the geometry of the polygon.
- Use scanline and edge coherence for efficient filling.

Algorithm

for each scanline

- Calculate Intersections between scanline and all polygon edges
- Sort Intersections on X.
- Fill pixels between pairs of intersection points.
The Scanline Polygon Fill Algorithm: Special Cases

1. **Horizontal Edges**: Discard all horizontal edges

2. **Scanline passing through a vertex**:
   - scanline $\alpha$: no problem, scanline $\beta$: wrong results
   - **Solution**: check for local extrema (in Y)
   - if local maxima/minima count two intersections
   - else count one intersection
   - **Implementation**: if vertex does not conform to a local extrema, **shorten edge by 1 pixel in Y.**
Improving Performance

- Use edge and scanline coherence.
- Implemented using an Edge Table (ET) and an Active Edge Table (AET)

Edge Record

```c
{ 
    Integer y_upper /* y coordinate at upper end point */
    Integer x_int /* x coordinate of intersection with edge */
    Float recip_slope /* 1/m */
    Pointer next; /* to next edge record */
}
```
Edge Table
Example

Edge Table

Raster Algorithms
Algorithm

Initialize /* take care of special cases */
AET = NULL
for y = min_row to max_row
{
    add all edge records in ET[y] to AET
    if ( AET NOT_EQUAL_TO NULL )
    {
        Sort AET on x /* x coordinates of intersection points */
        Fill pixel runs.
        Delete edge records for whom y = y_upper
        Update x_int values by addiing $1/m$
    }
}
Line Clipping

Problem Definition

Given a clip region and a line segment, determine the sections of the line interior to the region.

Simplification

Clip region (window) is an upright rectangle.
Line Clipping Alg: Strategy

if end points are inside clip region
    entire line segment is visible
else
    calculate intersections between line segment and clip region
    output segments interior to clip region
end

Note:

• we deal with line segments, not infinite lines.
• Slope-Intercept form \((y = mx + c)\) of line equation is not convenient
• Parametric representation preferred.
Cohen-Sutherland Clipping Algorithm

Idea:

Consider clip region to be the intersection of half-spaces.

- Define In and Out half-spaces for each boundary segment of the clip window.
- clip region = Intersection of all “In” spaces.
- Each In/Out region is represented by a 1 bit code.
Cohen-Sutherland Clipping Algorithm (contd.)

Given line end points: \((x_0, y_0), (x_1, y_1)\), and clip Boundaries: \(x_{\min}, x_{\max}, y_{\min}, y_{\max}\)

<table>
<thead>
<tr>
<th>1011</th>
<th>0001</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>0010</td>
<td>0110</td>
</tr>
</tbody>
</table>

Encoding a point:

\[
\begin{align*}
\text{code0}[0] &= x_0 < x_{\min} \\
\text{code0}[1] &= x_0 > x_{\max} \\
\text{code0}[2] &= y_0 < y_{\min} \\
\text{code0}[3] &= y_0 > y_{\max}
\end{align*}
\]

and similarly for \((x_1, y_1)\)
Algorithm

Calculate codes of both end points.

\[
\text{done} = \text{FALSE}
\]

\begin{algorithm}
\textbf{while} not done \\
\hspace{1em} \begin{algorithmic}
    \Statex \textbf{if} (all 4 codes of both endpoints equal zero)
    \hspace{1em} \text{"Accept line segment", done = TRUE}
    \Statex \textbf{else if} (bit-wise logical AND of code0 and code1 is non-zero)
    \hspace{1em} \text{"Reject line segment", done = TRUE}
    \Statex \textbf{else}
    \hspace{1em} \begin{algorithmic}
        \Statex \hspace{1em} \text{choose end point } P \text{ outside clip window}
        \Statex \hspace{1em} \text{Intersect line segment with clip window}
        \Statex \hspace{1em} \text{Discard line segment from } P \text{ to intersection point}
        \Statex \hspace{1em} \text{compute codes of the new (clipped) line segment}
    \end{algorithmic}
\end{algorithmic}
\end{algorithm}
Example

Trivial Accept: No!
Trivial Reject: \((1000 \& 0001) = 0000\). NO!

Pick end point with non-zero code - A (can also pick B)
Intersect with left boundary (corresponding to the first non-zero bit)

\[
\begin{align*}
\frac{t}{1.0} &= \frac{x_{\text{min}} - x_0}{x_1 - x_0} \\
y &= y_0 + t(y_1 - y_0) = y_0 + \frac{x_{\text{min}} - x_0}{x_1 - x_0}(y_1 - y_0) \\
\text{Intersection } I_1 &= (x_{\text{min}}, y), \quad x_0 = x_{\text{min}}, y_0 = y
\end{align*}
\]
Example (contd.)

Code \((I_1) = 0000\), Code \((B) = 0001\)

Cannot trivially accept or reject, choose end point B
Intersect with top boundary, \(y = y_{max}\)

\[
\frac{t}{1.0} = \frac{y_{max} - y_1}{y_0 - y_1}
\]
\[
x = x_1 + t(x_0 - x_1) = x_1 + \frac{y_{max} - y_1}{y_0 - y_1}(x_0 - x_1)
\]

Intersection \(I_2 = (x, y_{max})\)

Calculate codes of \(I_1\) and \(I_2\)
Trivially accept the segment \((I_1, I_2)\)
Polygon Clipping

Using a Line Clipper

- Apply Cohen Sutherland algorithm to each segment of polygon
- Insufficient, as polygons can become fragmented after clipping
- Polygons must remain polygons after clipping
Liang-Barsky Clipping Algorithm

- A special case of the Cyrus-Beck clipping algorithm.

Features

- In general, more efficient than the Cohen-Sutherland algorithm.
- Most of the operations are performed in parametric space.
Liang-Barsky Clipping: Idea

- Intersections between a line and the clipping boundaries are distinguished as entering or exiting the boundary ($t_{\text{enter}}$ and $t_{\text{exit}}$).
- If $\max(t_{\text{enter}k}) < \min(t_{\text{exit}k})$ and the interval is a subset of $(0, 1.0)$, then the line segment intersects the interior of the clipping region.
Liang-Barsky Clipping: Details

A point \((x, y)\) is in the interior if

\[
\begin{align*}
wx_{min} & \leq x_0 + t\Delta x \leq wx_{max} \\
yw_{min} & \leq y_0 + t\Delta y \leq yw_{max}
\end{align*}
\]

which can be rewritten as

\[
tp_k \leq q_k, \quad k = 1, 2, 3, 4
\]

\[
\begin{align*}
p_1 &= -\Delta x, & q_1 &= x_0 - wx_{min} \\
p_2 &= \Delta x, & q_2 &= wx_{max} - x_0 \\
p_3 &= -\Delta y, & q_3 &= y_0 - yw_{min} \\
p_4 &= \Delta y, & q_4 &= yw_{max} - y_0
\end{align*}
\]
Liang-Barsky Clipping: Details

Intersection

\[ t = \frac{q_k}{p_k} \]

- \( p_k = 0 \): line parallel to the \( k \)th clipping boundary; if \( q_k < 0.0 \), line is trivially rejected.
- \( p_k < 0 \): line proceeds from outside to inside across clipping boundary.
- \( p_k > 0 \): line proceeds from inside to outside across clipping boundary.
Algorithm

\( t_{\text{min}} = 0.0, t_{\text{max}} = 1.0 \)

\[
\text{if } \text{ClipBoundary} \left( p_1, q_1, &t_{\text{min}}, &t_{\text{max}} \right) \\
\quad \text{if } \text{ClipBoundary} \left( p_2, q_2, &t_{\text{min}}, &t_{\text{max}} \right) \\
\quad \quad \text{if } \text{ClipBoundary} \left( p_3, q_3, &t_{\text{min}}, &t_{\text{max}} \right) \\
\quad \quad \quad \text{if } \text{ClipBoundary} \left( p_4, q_4, &t_{\text{min}}, &t_{\text{max}} \right) \\
\quad \quad \quad \quad \{ \\
\quad \quad \quad \quad \quad \text{if } (t_{\text{min}} > 0.0) \{ \\
\quad \quad \quad \quad \quad \text{pt1.x} = \text{pt1.x} + t_{\text{min}} \ast d_{x} \\
\quad \quad \quad \quad \quad \text{pt1.y} = \text{pt1.y} + t_{\text{min}} \ast d_{y} \\
\quad \quad \quad \quad \quad \} \\
\quad \quad \quad \quad \quad \text{if } (t_{\text{min}} < 1.0) \{ \\
\quad \quad \quad \quad \quad \text{pt2.x} = \text{pt2.x} + t_{\text{max}} \ast d_{x} \\
\quad \quad \quad \quad \quad \text{pt2.y} = \text{pt2.y} + t_{\text{max}} \ast d_{y} \\
\quad \quad \quad \quad \quad \} \\
\quad \quad \quad \quad \} \\
\quad \quad \quad \text{"draw line from pt1 to p2"} \\
\quad \quad \} \\
\} \\
\]