Interactive Graphic System

Application Model
  - Represents data and objects to be displayed on the output device

Application Program
  - Creates, stores into, and retrieves from the application model
  - Handles user-inputs
  - Sends output commands to the graphics system:
    - *Which* geometric object to view (point, line, circle, polygon)
    - *How to view it* (color, line-style, thickness, texture)

Graphics System

  - Intermediates between the application program and the interface hardware:
    - Output flow
    - Input flow
  - Causes the application program to be *device-independent.*

Display Hardware

CRT - Cathode Ray Tube

- Westinghouse Type 2553
- Westinghouse Type 2554
- Raytheon Type 4120

- *Shadow mask* and phosphor coated screen
- Electron guns
- Phosphors on glass screen
**Terminology**

**Pixel:** Picture element.
- Smallest accessible element in picture
- Assume rectangular or circular shape

**Aspect Ratio:** Ratio between physical dimensions of a pixel (not necessarily 1)

**Dynamic Range:** The ratio between the minimal (not zero!) and the maximal light intensity a display pixel can emit

**Resolution:** The number of distinguishable rows and columns in the device. Measured in:
- Absolute values (1K x 1K) or,
- Density values (300 dpi [=dots per inch])

**Screen Space:** A discrete Cartesian coordinate system of the screen pixels

**Object Space:** The Cartesian coordinate system of the universe, in which the objects (to be displayed) are embedded

**Scan Conversion**

The conversion from a geometrical representation of an object to pixels in a raster display

---

<table>
<thead>
<tr>
<th>I</th>
<th>R</th>
<th>G</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Display Hardware**

**FED - Field Emission Display**

- Anode
- Phosphor
- Microtips (Emitters)
- C adsode electrodes

---

**Scan Conversion**

The conversion from a geometrical representation of an object to pixels in a raster display

---

**Raster Display**

- Display commands
- Interaction data
- Keyboard
- Mouse

---

**Terminology**

**Pixel:** Picture element.
- Smallest accessible element in picture
- Assume rectangular or circular shape

**Aspect Ratio:** Ratio between physical dimensions of a pixel (not necessarily 1)

**Dynamic Range:** The ratio between the minimal (not zero!) and the maximal light intensity a display pixel can emit

**Resolution:** The number of distinguishable rows and columns in the device. Measured in:
- Absolute values (1K x 1K) or,
- Density values (300 dpi [=dots per inch])

**Screen Space:** A discrete Cartesian coordinate system of the screen pixels

**Object Space:** The Cartesian coordinate system of the universe, in which the objects (to be displayed) are embedded
Representations

Implicit formula:
Constraint(s) expressed as
\( f(x_1, x_2, ..., x_n) = 0 \)
A k-dimensional surface embedded in n-dimensions
\( f_i(x_1, x_2, ..., x_n) = 0 \); \( i = 1..n-k \)

Explicit formula:
For each \( x \) define \( y = f(x) \)
Good only for "functions".

Parametric formula:
Depending on free parameter(s)
\( x = f_x(t) \)
\( y = f_y(t) \)
For k-dimensional surface there are k free parameters

Line in 2 dimensions

- Implicit representation:
  \( \alpha x + \beta y + \gamma = 0 \)
- Explicit representation:
  \( y = mx + B \)
  \( m = \frac{y_1 - y_0}{x_1 - x_0} \)
- Parametric representation:
  \[ P = \begin{pmatrix} x \\ y \end{pmatrix} \]
  \[ P = P_0 + (P_1 - P_0) t \]
  \( t \in [0..1] \)

Scan Conversion - Lines

Incremental Algorithm

\[ y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m \Delta x \]
if \( \Delta x = 1 \) then
\[ y_{i+1} = y_i + m \]

Algorithm
\( y = y_0 \)
For \( x = x_0 \) to \( x_1 \)
  PlotPixel(\( x \), round(\( y \)))
  \( y = y + m \)
end;

Scan Conversion - Lines

Pseudo Code for Basic Line Drawing

Assume \( x_0 > x_0 \) and line slope absolute value is \( \leq 1 \)

\( \text{Line}(x_0, y_0, x_1, y_1) \)
begin
  float \( dx, dy, x, y, \text{slope} \);
  \( dx := x_1 - x_0 \);
  \( dy := y_1 - y_0 \);
  \text{slope} := dy/dx;
  \text{if} \( x = x_0 \) to \( x_1 \), do begin
    PlotPixel(\( x \), round(\( y \)));
    \( y := y + \text{slope} \);
  end;
end;
Basic Line Drawing

Symmetric Cases:

\[ |m| \geq 1 \]

\[ x = x_0 \]

For \( y = y_0 \) to \( y_1 \)

\[ PlotPixel(round(x),y) \]

\[ x = x + 1/m \]

end;

Special Cases:

\[ m = \pm 1 \] (diagonals)

\[ m = 0, \infty \] (horizontal, vertical)

Symmetric Cases:

if \( x_0 > x_1 \) for \( |m| \leq 1 \) or \( y_0 > y_1 \) for \( |m| \geq 1 \)

swap((x_0,y_0),(x_1,y_1))

Midpoint (Bresenham) Line Drawing

Assumptions:

- \( x_0 < x_1 \), \( y_0 < y_1 \)
- \( 0 < \text{slope} < 1 \)

For each iteration:

- 1 addition, 1 rounding

Drawbacks:

- Accumulated error
- Floating point arithmetic
- Round operations

Midpoint (Bresenham) Line Drawing

Implicit form of a line:

\[ f(x,y) = ax + by + c = 0 \]

Decision Variable:

\[ d = f(x,y) = a(x + 1/2) + b(y + 1/2) + c \]

- choose NE if \( d > 0 \)
- choose E if \( d \leq 0 \)

Midpoint (Bresenham) Line Drawing

Initialization:

First point = \((x_0,y_0)\), first MidPoint = \((x_0+1,y_0+1/2)\)

\[ d_{\text{start}} = d(f(x_0+1,y_0+1/2)) = a(x_0 + 1) + b(y_0 + 1/2) + c \]

\[ = ax_0 + by_0 + c + a + b/2 \]

\[ = f(x_0,y_0) + a + b/2 = a + b/2 \]

Enhancement:

To eliminate fractions, define:

\[ f(x,y) = 2(ax + by + c) = 0 \]

\[ d_{\text{start}} = 2dy - dx/2 \]

Midpoint (Bresenham) Line Drawing

Comments:

- Integer arithmetic (dx and dy are integers)
- One addition for each iteration
- No accumulated errors
- By symmetry, we deal with \( 0 < \text{slope} < 1 \)
Pseudo Code for Midpoint Line Drawing

```pseudocode
Line(x0,y0,x1,y1)
begin
    int dx, dy, x, y, d, E1, E2;
    x := x0; y := y0;
    dx := x1 - x0; dy := y1 - y0;
    d := 2*dy - dx;
    E1 := 2*dy;
    PlotPixel(x,y);
    while(x < x1) do
        if (d < 0) then
            d := d + E1;
            x := x + 1;
        else
            d := d + E2;
            x := x + 1;
            y := y + 1;
        end;
        PlotPixel(x,y);
    end;
end;
```

Assume \( x > x_0 \) and \( 0 < \text{slope} \leq 1 \)

---

Scan Conversion - Circles

### Implicit representation (centered at the origin, radius R):
\[
x^2 + y^2 - R^2 = 0
\]

### Explicit representation:
\[
y = \pm \sqrt{R^2 - x^2}
\]

### Parametric representation:
\[
\begin{pmatrix}
    x \\
    y
\end{pmatrix} = \begin{pmatrix}
    R \\
    R
\end{pmatrix} \begin{pmatrix}
    \cos (t) \\
    \sin (t)
\end{pmatrix}, \quad t \in [0, 2\pi]
\]

---

Scan Conversion - Circles

**Basic Algorithm**

For \( x = -R \) to \( R \)
- \( y = \sqrt{R^2-x^2} \)
- PlotPixel(x,round(y))
- PlotPixel(x,-round(y))

**Comments:**
- Square-root operations are expensive
- Floating point arithmetic
- Large gap for \( x \) values close to \( R \)

---

Scan Conversion - Circles

**Exploiting Eight-Way Symmetry**

For a circle centered at the origin:
If \((x,y)\) is on the circle then
\(\text{(-x,-y) (-y,-x) (-y,x) (-x,y)}\)
\(\text{(x,-y) (y,-x) (y,x) (x,y)}\)
are on the circle as well.

Therefore we need to compute only one octant (45°) segment.

---

Circle Midpoint (for one octant)
(The circle is located at (0,0) with radius R)

- We start from \((x_0,y_0) = (0, R)\)
- One can move either **East** or **South-East**
- Again, \( d(x,y) \) will be a threshold criteria at the midpoint

---

Scan Conversion - Circles

**CirclePoints(x, y)**

```pseudocode
CirclePoints(x, y)
begin
    PlotPixel(x,y);
    PlotPixel(y,x);
    PlotPixel(y,-x);
    PlotPixel(x,-y);
    PlotPixel(-x,-y);
    PlotPixel(-y,-x);
    PlotPixel(-y,x);
    PlotPixel(-x,y);
end;
```
Circle Midpoint (for one octant)

Threshold Criteria:
\[ d(x,y) = f(x,y) = x^2 + y^2 - R^2 = 0 \]

- \( f(x,y) = 0 \)
- \( f(x,y) < 0 \)
- \( f(x,y) > 0 \)

Threshold Criteria:
- At the beginning
  \[ d_{\text{start}} = d(x_0+1,y_0-1/2) = (1,R-1/2) = 5/4 - R \]
- If \( d < 0 \) we move East:
  \[ \Delta_E = d(x_0+2,y_0-1/2) - d(x_0+1,y_0-1/2) = 2x_0 + 3 \]
- If \( d > 0 \) we move South-East:
  \[ \Delta_{SE} = d(x_0+2,y_0-3/2) - d(x_0+1,y_0-1/2) = 2(x_0-y_0) + 5 \]
- \( \Delta_E \) and \( \Delta_{SE} \) are not constant anymore
- Since \( d \) is incremented by integer values, we can use
  \[ d_{\text{start}} = 1-R \]
  yielding an integer algorithm. This has no affect on the threshold criteria.

Pseudo Code for Circle Midpoint

```plaintext
MidpointCircle(R)
begin
int x, y, d;
x := 0;
y := R;
d := 5.0/4.0 - R;
CirclePoints(x,y);
while ( y > x ) do
    if ( d < 0 ) then            /* East */
        d := d+2*x+3; /* Multiplication! */
        x := x+1;
    else /* South East */
        d := d+2*(x-y)+5; /* Multiplication! */
        x := x+1;
y := y-1;
end;
CirclePoints( x, y ); /* Mirror to create the other seven octants */
end;
end;
```

Pseudo Code for Circle Midpoint

```plaintext
MidpointCircle(R)
begin
int x, y, d;
x := 0;
y := R;
d := 1-R; /* originally d := 5.0/4.0 - R */
CirclePoints(x,y);
while ( y > x ) do
    if ( d < 0 ) then            /* East */
        d := d+2*x+3; /* See Foley & van Dam pg. 87 */
        x := x+1;
    else /* South East */
        d := d+2*(x-y)+5; /* See Foley & van Dam pg. 87 */
        x := x+1;
y := y-1;
end;
CirclePoints( x, y ); /* Mirror to create the other seven octants */
end;
```

Pseudo Code for Circle Midpoint

```plaintext
MidpointCircle(R)
begin
int x, y, d;
x := 0;
y := R;
\[ d := 3; \]
\[ \Delta_E := 0; \]
\[ \Delta_{SE} := 5; \]
CirclePoints(x,y);
while ( y > x ) do
    if ( d < 0 ) then            /* East */
        \[ d := d + \Delta_E; \]
        \[ \Delta_E := \Delta_E + 2; \]
        \[ x := x+1; \]
    else /* South East */
        \[ d := d + \Delta_{SE}; \]
        \[ \Delta_E := \Delta_E + 2; \]
        \[ x := x+1; \]
y := y-1;
end;
CirclePoints( x, y ); /* Mirror to create the other seven octants */
end;
```
**Polygon Fill**

**Representation:**
- Polygon = $V_0, V_1, V_2, .. V_n$
- $V_i = (x_i, y_i)$ - vertex
- $E_i = (V_i, V_{i+1})$ - edge
- $E_n = (V_n, V_0)$

**Scan Conversion - Polygon Fill**

**Problem:** Given a closed 2D polygon, fill its interior with a specified color, on a graphics display

**Assumption:** Polygon is simple, i.e. no self intersections, and simply connected, i.e. without holes

**Solutions:**
- Flood fill
- Scan Conversion

**Flood Fill Algorithm**

- Let $P$ be a polygon with $n$ vertices, $v_0 .. v_{n-1}$
- Denote $v_n = v_0$
- Let $c$ be a color to paint $P$
- Let $p = (x, y)$ be a point in $P$

```
FloodFill(P, x, y, c)
if (OnBoundary(x, y, P) or Colored(x, y, c))
then return;
else begin
PlotPixel(x, y, c);
FloodFill(P, x+1, y, c);
FloodFill(P, x, y+1, c);
FloodFill(P, x, y-1, c);
FloodFill(P, x-1, y, c);
end;
```

**Scan Conversion – Basic Algorithm**

```plaintext
ScanConvert(P, c)
For j := 0 to ScreenYMax do
  I := points of intersection of edges from P with line y = j;
  Sort I in increasing x order and fill with color c alternating segments;
end;
```

**Question:** How do we know if a given point is inside or outside a polygon?

**Fill Polygon**

**Question:** How do we find the intersecting edges?
Scan Conversion – Fill Polygon

What happens in with these cases?

Intersections at pixel coordinates

Rule: In the odd/even count, we count $y_{\text{min}}$ vertices of an edge, but not $y_{\text{max}}$ vertices

Vertex B is counted once because $y_{\text{min}}$ of (A,B)
Vertex E is not counted because $y_{\text{max}}$ of both (D, E) and (E, F)
Vertex H is counted twice because $y_{\text{min}}$ of both (G, H) and (H, I)

Fill Polygon – Optimized Algorithm

Uses a list of “active” edges $A$ (edges currently intersecting the scan line)

$$\text{ScanConvert}(P, c)$$

Sort all edges $E = \{E_i\}$ in increasing $\text{MinY}(E_i)$ order.
$A := \emptyset$;
For $k := 0$ to ScreenYMax do
  For each $E_i \in E$,
    if $\text{MinY}(E_i) \leq k$ then
      $A := A \cup E_i$;
      $E := E - E_i$
    For each $E_i \in A$,
      if $\text{MaxY}(E_i) \leq k$ then
        $A := A - E_i$
  $I := \text{Points of intersection of members from } A \text{ with line } y = k$;
  Sort $I$ in increasing $x$ order and draw with color $c$ alternating segments;
end;

Implementation with linked lists

Flood Fill vs. Scan Conversion

<table>
<thead>
<tr>
<th>Flood Fill</th>
<th>Scan Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Very simple.</td>
<td></td>
</tr>
<tr>
<td>• Requires a seed point</td>
<td></td>
</tr>
<tr>
<td>• Requires large stack size</td>
<td></td>
</tr>
<tr>
<td>• Common in paint packages</td>
<td></td>
</tr>
<tr>
<td>• More complex</td>
<td></td>
</tr>
<tr>
<td>• No seed point is required</td>
<td></td>
</tr>
<tr>
<td>• Requires small stack size</td>
<td></td>
</tr>
<tr>
<td>• Used in image rendering</td>
<td></td>
</tr>
</tbody>
</table>