Output Primitives

- Graphic SW and HW provide subroutines to describe a scene in terms of basic geometric structures called output primitives.
- Output primitives are combined to form complex structures.
- Simplest primitives
  - Point (pixel)
  - Line segment
Scan Conversion

• Converting output primitives into frame buffer updates. Choose which pixels contain which intensity value.

• Constraints
  – Straight lines should appear as a straight line
  – Primitives should start and end accurately
  – Primitives should have a consistent brightness along their length
  – They should be drawn rapidly
Line Drawing

- Simple approach:
  sample a line at discrete positions at one coordinate from start point to end point, calculate the other coordinate value from line equation.

  \[ y = mx + b \quad x = \frac{1}{m} y + \frac{b}{m} \]

  \[ m = \frac{y_{\text{start}} - y_{\text{end}}}{x_{\text{start}} - x_{\text{end}}} \]

  If \( m > 1 \), increment \( y \) and find \( x \)
  If \( m \leq 1 \), increment \( x \) and find \( y \)
Digital Differential Analyzer

• Simple approach: too many floating point operations and repeated calculations

• Calculate $y_{k+1}$ from $y_k$ for a $\Delta x$ value

\[
\Delta y = m \Delta x \quad \quad y_{k+1} = y_k + m \quad \text{for} \quad \Delta x = 1, \quad 0 < m < 1
\]

\[
\Delta x = \frac{\Delta y}{m} \quad \quad x_{k+1} = x_k + \frac{1}{m} \quad \text{for} \quad \Delta y = 1, \quad m \geq 1
\]
Bresenham's Line Algorithm

- DDA: Still floating point operations

Assume $|m| \leq 1$

If already at $(x_k, y_k)$, choices:

$(x_k + 1, y_k)$ if $d_1 \leq d_2$

$(x_k + 1, y_k + 1)$ if $d_1 > d_2$

$y = m(x_k + 1) + b \Rightarrow$

$d_1 = y - y_k = m(x_k + 1) + b - y_k$

$d_2 = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b + 1$

$m = \frac{\Delta y}{\Delta x} = \frac{y_{\text{end}} - y_{\text{start}}}{x_{\text{end}} - x_{\text{start}}}$

Define $p_k = \Delta x(d_1 - d_2) = 2\Delta y x_k - 2\Delta x y_k + c$

$c = 2\Delta y + \Delta x (2b - 1)$ Independent from pixel position
If $d_1 < d_2 \Rightarrow p_k < 0 \Rightarrow \text{Choose } y_k$

Else

Choose $y_k + 1$

at step $k+1$:

\[ p_{k+1} = 2 \Delta y x_{k+1} - 2 \Delta x y_{k+1} + c \]

\[ p_{k+1} - p_k = 2 \Delta y (x_{k+1} - x_k) - 2 \Delta x (y_{k+1} - y_k) \]

\[ x_{k+1} = x_k + 1 \Rightarrow p_{k+1} = p_k + 2 \Delta y - 2 \Delta x (y_{k+1} - y_k) \]

\[ y_{k+1} - y_k = 0 \quad \text{If } y_k \text{ is choosen,} \]

\[ y_{k+1} - y_k = 1 \quad \text{If } y_{k+1} \text{ is choosen} \]

for $x_0$:

\[ c = 2 \Delta y + \Delta x (2b - 1) = 2 \Delta y + \Delta x (2y_0 - 2x_0 \frac{\Delta y}{\Delta x}) - \Delta x \]

\[ = 2 \Delta y + 2 \Delta x y_0 - 2 \Delta y x_0 - \Delta x \]
\( p_0 = 2 \Delta y x_0 - 2 \Delta x y_0 + c = 2 \Delta y x_0 - 2 \Delta x y_0 + 2 \Delta y + 2 \Delta x y_0 - 2 \Delta y x_0 - \Delta x \)
\( p_0 = 2 \Delta y - \Delta x \)

\((x_0, y_0) \) known

\( p_{k+1} = p_k + 2 \Delta y - 2 \Delta x(y_{k+1} - y_k) \)

- **Algorithm:**
  
  \[
  \text{draw}(x_0, y_0) \\
  p_k \leftarrow 2 \Delta y - \Delta x; \quad x_k \leftarrow x_0 \\
  \text{while } x_k < x_{\text{end}} \\
  \quad x_{k+1} \leftarrow x_k + 1 \\
  \quad \text{if } p_k \leq 0 \quad \text{choose } y_k \\
  \quad \quad y_{k+1} \leftarrow y_k; \quad p_{k+1} \leftarrow p_k + 2 \Delta y \\
  \quad \text{else } \quad \text{choose } y_{k+1} \\
  \quad \quad y_{k+1} \leftarrow y_k + 1; \quad p_{k+1} \leftarrow p_k + 2 \Delta y - 2 \Delta x \\
  \text{draw}(x_{k+1}, y_{k+1}) \\
  x_k \leftarrow x_k + 1 \\
  p_k \leftarrow p_{k+1} 
  \]
Circle Generation

\[(x - x_0)^2 + (y - y_0)^2 = r^2\]

unit steps in \(x\) \(\Rightarrow\) \(y = y_0 \mp \sqrt{r^2 - (x - x_0)^2}\)

- Computationally complex
- Non uniform spacing
- Polar coordinates:
  \[x = r \cos(\theta) + x_c\]
  \[y = r \sin(\theta) + y_c\]
• Fixed angular step size to have equally spaced points

\[
x_k = r \cos \theta \quad x_{k+1} = r \cos(\theta + d \theta)
\]
\[
y_k = r \sin \theta \quad y_{k+1} = r \sin(\theta + d \theta)
\]

\[
x_{k+1} = r \cos \theta \cos d \theta - r \sin \theta \sin d \theta
\]
\[
= x_k \cos d \theta - y_k \sin d \theta
\]
\[
y_{k+1} = r \sin \theta \cos d \theta + r \cos \theta \sin d \theta
\]
\[
= y_k \cos d \theta - y_k \sin d \theta
\]

fixed \( d \theta \) so compute \( \cos d \theta \) and \( \sin d \theta \) initially
• Computation can be reduced by considering symmetry of circles:

• Still too complex, multiplications, trigonometric calculations

• Bresenham's circle generation algorithm involves simple integer operations

• Midpoint Circle Algorithm generates the same pixels
Midpoint Circle Algorithm

- Consider the second octant. Increment $x$, decide on $y$.

- Assume that select which of 2 pixels are closer to the circle by evaluating the function at E and SE.
\[ f(x, y) = x^2 + y^2 - r^2 = \begin{cases} 
0 & \text{if on the circle} \\
>0 & \text{if outside the circle} \quad \text{choose SE} \\
<0 & \text{if inside the circle} \quad \text{choose E} 
\end{cases} \]

\[ p_k = f(x_k+1, y_k - \frac{1}{2}) = (x_k+1)^2 + (y_k - \frac{1}{2})^2 - r^2 \]

\[ p_{k+1} = f(x_{k+1}+1, y_{k+1} - \frac{1}{2}) = (x_k+1)^2 + (y_{k+1} - \frac{1}{2})^2 - r^2 \]

\[ p_{k+1} - p_k = (x_k+1+1)^2 + (y_{k+1} - \frac{1}{2})^2 - r^2 - (x_k+1)^2 - (y_k - \frac{1}{2})^2 + r^2 \]

\[ p_{k+1} = p_k + x_k^2 + 4x_k + 4 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - x_k^2 - 2x_k - 1 - y_k^2 + y_k - \frac{1}{4} \]
\[ p_{k+1} = p_k + x_k^2 + 4x_k + 4 + y_{k+1}^2 - y_{k+1} + \frac{1}{4} - x_k^2 - 2x_k - 1 - y_k^2 + y_k - \frac{1}{4} \]

\[ p_{k+1} = p_k + 2x_k + 3 + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) \]

If \( E \) is chosen \( p_k < 0 \): 
\[ y_{k+1} = y_k \Rightarrow p_{k+1} = p_k + 2x_k + 3 \]

If \( SE \) is chosen \( p_k \geq 0 \):
\[ y_{k+1} = y_k - 1 \Rightarrow p_{k+1} = p_k + 2x_k + 3 - 2y_k + 1 + 1 = p_k + 2x_k - 2y_k + 5 \]

\[ x_0 = 0; \ y_0 = r; \ p_0 = f(0, r - \frac{1}{2}) = 1 + (r - \frac{1}{2})^2 - r^2 = \frac{5}{4} - r \]

All increments are integer, rounding \( \frac{5}{4} \) will give 1 so,

\[ p_0 = 1 - r \]
• Algorithm:

\[
\text{drawoctants}(x_0, r) \\
p_k \leftarrow 1 - r; \ x_k \leftarrow 0; \ y_k \leftarrow r \\
\text{while } x_k \leq y_k \\
\quad \text{if } p_k < 0 \quad \text{choose E} \\
\quad \quad y_{k+1} \leftarrow y_k; \quad p_{k+1} \leftarrow p_k + 2x_k + 3 \\
\quad \text{else choose SE} \\
\quad \quad y_{k+1} \leftarrow y_k - 1; \quad p_{k+1} \leftarrow p_k + 2x_k - 2y_k + 5 \\
\quad x_{k+1} \leftarrow x_k + 1 \\
\text{drawoctants}(x_{k+1}, y_{k+1}) \\
\]

\[
x_k \leftarrow x_{k+1} \\
y_k \leftarrow y_{k+1} \\
p_k \leftarrow p_{k+1} \\
\]
\[ p_k < 0 \quad \text{choose E} \]
\[ y_{k+1} \leftarrow y_k; \quad p_{k+1} \leftarrow p_k + 2 x_k + 3 \]

else \quad \text{choose SE}
\[ y_{k+1} \leftarrow y_k - 1; \quad p_{k+1} \leftarrow p_k + 2 x_k - 2 y_k + 5 \]

\[ x = 0; \quad y = 10; \quad r = 10 \]

\[ p_k = 1 - 10 = -9 \quad \text{choose E} \quad \text{plot} \ (0,10) \]

\[ p_k = -9 + 2 + 3 = -4 \quad \text{choose E} \quad \text{plot} \ (1,10) \]

\[ p_k = -4 + 4 + 3 = 3 \quad \text{choose SE} \quad \text{plot} \ (2,10) \]

\[ p_k = 3 + 6 - 18 + 5 = -4 \quad \text{choose E} \quad \text{plot} \ (3,9) \]

\[ p_k = -4 + 8 + 3 = 7 \quad \text{choose SE} \quad \text{plot} \ (4,9) \]

\[ p_k = 7 + 10 - 16 + 5 = 6 \quad \text{choose SE} \quad \text{plot} \ (5,8) \]

\[ p_k = 6 + 12 - 14 + 5 = 9 \quad \text{choose SE} \quad \text{plot} \ (6,7) \]
Ellipse Generation

- Similar to circle generation with mid-point. Inside test.
- Different formula for points up to the tangent $y = -x$, slope $< 1$.
  - $(0, b)$ to tangent: increment $x$ find $y$
  - Tangent to $(a, 0)$: decrement $y$ find $x$
- Mid-point algorithm is applicable to other polynomial equations:
  - Parabola, Hyperbola
Filled Area Primitives

- Two basic approaches to area filling on raster systems:
  - Determine the overlap intervals for scan lines that cross the area (scan-line)
  - Start from an interior position and point outward from this point until the boundary condition reached (fill method)

- Scan-line: simple objects, polygons, circles,.. 
- Fill-method: complex objects, interactive fill.
Scan-line Polygon Fill

- For each scan-line:
  - Locate the intersection of the scan-line with the edges \( y = y_s \)
  - Sort the intersection points from left to right.
  - Draw the interiors intersection points pairwise. (a-b), (c-d)
- Problem with corners. Same point counted twice or not?
- $a, b, c$ and $d$ are intersected by 2 line segments each.

- Count $b, d$ twice but $a$ and $d$ once. Why?

- Solution: Make a clockwise or counterclockwise traversal on edges. Check if $y$ is monotonically increasing or decreasing. If direction changes, double intersection, otherwise single intersection.
Scan-line Polygon Filling (coherence)

- **Coherence**: Properties of one part of a scene are related with the other in a way that can it be used to reduce processing of the other.

- Scan-lines adjacent to each other. Intersection of scan-lines are close to each other (like scan conversion of a line)

- Intersection points with scan lines:

\[ x_{k+1} = \text{round}(x_k + \frac{1}{m}) \]
• Instead of floating point operations, use integer operations:

\[ m = \frac{\Delta y}{\Delta x} \quad x_{k+1} = x_k + \frac{\Delta x}{\Delta y} \]

\[ \text{counter} \leftarrow 0 \]

for each scan-line

\[ \text{counter} \leftarrow \text{counter} + \Delta x \]

while \( \text{counter} \geq \Delta y \)

\[ x \leftarrow x + 1 \]

\[ \text{counter} \leftarrow \text{counter} - \Delta y \]

• Example:

\[ m = \frac{8}{5} \]

<table>
<thead>
<tr>
<th>scanline</th>
<th>counter</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>
Efficient Polygon Fill

- Make a (counter)clockwise traversal and shorten the single intersection vertices.
- Generate a sorted edge table on the scan-line axis. Each edge has an entry in smallery valued corner point.
- Each entry keeps a linked list of all connected edges:
  - $x$ value of the point
  - $y$ value of the end-point
  - Slope of the edge
• Start with the smallest scan-line
• Keep an active edge list:
  – Update the current $x$ value of the edge based on $m$ value
  – Add the lists in the current table entry based on their $x$ value
  – Remove the completed edges
  – Draw the intermediate points of pairwise elements of the list.
- Example:
  A,B,C,D,E,F,A Polygon:
  (30,10),(24,32),(20,22),(16,34)
  (8,26),(12,16),(30,10)
  E'=(20,25), F'=(12,15)

Edge Table:

<table>
<thead>
<tr>
<th>Y</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[15,30,-3]</td>
<td>[32,30,-3/11]</td>
</tr>
<tr>
<td>16</td>
<td>[25,12,-2/5]</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>[34,20,-1/3]</td>
<td>[32,20,2/5]</td>
</tr>
<tr>
<td>26</td>
<td>[34,8,1]</td>
<td></td>
</tr>
</tbody>
</table>
Boundary Fill Algorithm

• Start at a point inside a continuous arbitrary shaped region and paint the interior outward toward the boundary. Assumption: boundary color is a single color

• 

\((x,y)\): start point; \(b\): boundary color, \(fill\): fill color

```c
void boundaryFill4(x,y,fill,b) {
    cur = getpixel(x,y)
    if (cur != b) AND (cur != fill) {
        setpixel(x,y,fill);
        boundaryFill4(x+1,y,fill,b);
        boundaryFill4(x-1,y,fill,b);
        boundaryFill4(x,y+1,fill,b);
        boundaryFill4(x,y-1,fill,b);
    }
}
```
• 4 neighbors vs 8 neighbors: depends on definition of continuity.
  8 neighbor: diagonal boundaries will not stop
• Recursive, so slow. For large regions with millions of pixels, millions of function calls.
• Stack based improvement: keep neighbors in stack
• Number of elements in the stack can be reduced by filling the area as pixel spans and pushing only the pixels with pixel transitions.
• Check the neighbor pixels as filling the area line by line

• If pixel changes from null to boundary or null when scan-line finishes, push the pixel information on stack.

• After a scan-line finishes, pop a value from stack and continue processing.
Flood-Fill

- Similar to boundary fill. Algorithm continues while the neighbor pixels have the same color.

- void FloodFill4(x,y,fill,oldcolor) {
  cur = getpixel(x,y)
  if (cur == oldcolor) {
    setpixel(x,y,fill);
    FloodFill4(x+1,y,fill,oldcolor);
    FloodFill4(x-1,y,fill,oldcolor);
    FloodFill4(x,y+1,fill,oldcolor);
    FloodFill4(x,y-1,fill,oldcolor);
  }
}

Character Generation

- Typesetting fonts:
  - Bitmap fonts: simple, not scalable.
  - Outline fonts: scalable, flexible, more complex process

Pixelwise on/off information

Points and tangents of the boundary
Attributes of Output Primitives

- Line attributes:
  - Line type
dotted, dashed, ...
  - Line width
  - Line caps and join
  miter, round, bevel
  - Line color
  - Line brush
• Area Fill Attributes:
  – Solid
  – Pattern
  – Gradient
• Character Attributes
  - Font, **bold**, *italic*, underlined, outline, shadow
  - Spacing abcdef abcdef abcdef abcdef
  - Direction normal
down 
rotated 
slanted
  - Text on arbitrary baseline path
Antialiasing

- CG generated shapes: limited resolution sampling of original shapes with infinite resolution
- Loose information, Aliasing: Jagging, stairway effects.
- Use higher resolutions. Expensive, no limit.
- Another solution: Antialiasing: use intensity levels.
• Supersampling: sample at higher resolution.
• Count the superpixels at each pixel. Give an intensity based on that value.
• Aliased vs. Antialiased.
Filtering techniques:
Giving a larger weight to center pixel and lower weight to neighbor pixels. Weight mask or volume based filters.

Pixel Weighting Mask

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Box filter

Cone filter

Gaussian filter