Lecture 02: Implementation Algorithms for Primitives (Chapter 6)

I. We will discuss a few algorithms from Chapter 6 in depth for drawing lines and polygons and discuss high level implementation of attributes.

II. Line-Drawing

A. Setup

i. Assume integer coordinates for start/end (can acquire by projecting to screen coordinates)

ii. As input we get 2 endpoints of a line \((x_0, y_0) - (x_{\text{end}}, y_{\text{end}})\)

iii. Recall line equation:

\[
\begin{align*}
y &= mx + b \\
\text{where } m &= \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0}, \quad b = y_0 - mx_0
\end{align*}
\]

we can compute changes (intervals) from this

\[
\delta y = m \delta x \quad \text{or} \quad \delta x = \frac{\delta y}{m}
\]

iv. To draw line we exploit these properties/equations to determine cells of grid pixels

To set in frame buffer to the line color. 

B. DDA - Digital Differential Analyzer - Based on computing \(\delta x\) and \(\delta y\) for a line.

i. For simplicity, assume \(x < x_0 < x_{\text{end}}\)

ii. Let \(\delta x = 1\), then \(y_{k+1} = y_k + m\)

where \(k\) starts at 0 until endpoint is reached

then we round \(y_k\) to an integer.

iii. For \(m > 1\) we swap roles of \(x\) and \(y\). That is: \(\delta y = 1\) and \(x_{k+1} = x_k + \frac{1}{m}\)

iv. For \(m < 1\) \(m = -1\) similar equations are derived.

v. Code:

\[
\text{Int x0, int y0, int xend, int yend)}
\]

1. \(\text{Int dx} = x_{\text{end}} - x_0\) \(\text{Int dy} = y_{\text{end}} - y_0\), steps, k;

2. \(\text{SInt xinc, yinc, x=x0, y=y0)}

3. \(\text{if} (\text{dx} > \text{dy})\)

4. \(\text{steps} = \lceil \text{dx} \rceil\)

5. else

6. \(\text{steps} = \lceil \text{dy} \rceil\)

7. \(\text{xinc} = \lceil \text{dx} \rceil / \text{steps}\), \(\text{yinc} = \lceil \text{dy} \rceil / \text{steps}\)

8. \(\text{y} = \text{round(y)}\), set\((\text{round}(x), \text{round}(y))\)

9. for \(\text{Int k = 0; k < steps; k++}\)

10. \(x + xinc, y + yinc)\)

11. \(\text{set} (\text{round}(x), \text{round}(y))\)

vi. Issues? *Ask Class*

a. Good: Better than naive to eliminate multiplication.

b. Bad: Error in floats accumulates, float arithmetic is slow!
C. Bresenham's Line Algorithm - accurate efficient (integer arithmetic only).

i. Observations:
- Only 2 possible y pixels could be chosen, (1,10) or (1,11) in example.
- Algorithm exploits this with single test.
- Anym for simplicity let m ≤ 1.

ii. Idea: Start at $(x_0, y_0)$ go to next x and pick y which is closest to mathematical line, i.e.:

$$x_{k+1} = x_k + 1$$

and we choose $y_{k+1} = y_k$ or $y_{k+1} = y_k + 1$

iii. Derivation:

$$s_0 = y = m(x_k) + b$$

$$d_{lower} = y - y_k = m(x_k + 1) + b - y_k$$

$$d_{upper} = y_{k+1} - y = y_{k+1} - m(x_k + 1) - b$$

Proximity is based on sign of $d_{lower} - d_{upper} =

$$= (m(x_k + 1) + b - y_k) - (y_{k+1} - m(x_k + 1) - b)$$

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

We can substitute $m = \frac{Δy}{Δx}$ and multiply by $Δx$ (to remove division) to derive the decision $p_k$. Its sign will be the same as $d_{lower} - d_{upper}$ to define our proximity.

$$p_k = Δx(d_{lower} - d_{upper}) = Δx((Δx(x_k + 1) - Δy_k + 2b + 1)$$

$$= 2Δxy_k - 2Δx y_k + 2Δy + Δx(b + 1)$$

If $y_k$ is closer $d_{lower} < d_{upper}$ than $p_k$ will be negative otherwise $p_k$ is positive.

So, we can determine $p_{k+1}$ using incremental integer computations.

$$p_{k+1} = 2Δy x_k + 2Δx y_k + C$$

Start $p_0 = 0$ from $p_k$ formula depending on sign of $p_k$.

$$p_c = 2Δy x_0 - 2Δx y_0 + 2Δy + 2Δx y_0 + 2Δx y_0 + 2Δx y_0 - 3Δy - Δx$$

$$= 2Δy - Δx$$
iv. Algorithm for \( |m| < 1.0 \)

- **Input:** \( x_0, y_0, x_{\text{end}}, y_{\text{end}} \) as integers
- **Output:**

1. set \( (x_0, y_0) \)
2. Integers \( y = y_{\text{end}} - y_0, \Delta y = x_{\text{end}} - x_0, \Delta x_0, \Delta y - \Delta x \)
3. \( P_0 = 2 \Delta y - \Delta x \)
4. for each \( x_k \) along line
5. if \( p < 0 \)
6. set \( (x_{k+1}, y_k) \)
7. \( p_{k+1} = p_k + \Delta y \)
8. else
9. set \( (x_k + 1, y_{k+1}) \)
10. \( p_{k+1} = p_k + \Delta y + \Delta x \)

For all cases of \( m \) through symmetry of octants/quadrannts.

b. Advantages: integer arithmetic

c. Code

```c
1. int dx = |x_{\text{end}} - x_0|, dy = |y_{\text{end}} - y_0|;
2. int p = 2 \cdot dy - dx;
3. int t = 2 \cdot \Delta y - \Delta x;
4. int x = x_0;
5. if (x < x_{\text{end}}) j = 1;
6. c[z \times x = x_0, y = y_0] j
7. set (x, y)
8. while \((x < x_{\text{end}}) \) &
9. \( x += \Delta x; \)
10. if \( p < 0 \)
11. \( p += \Delta y; \)
12. else
13. \( x += \Delta x; \)
14. \( p += \Delta y; \)
15. set (x, y)
```

V. Example: \( (x_0, y_0) = (30, 10) \), \( (x_{\text{end}}, y_{\text{end}}) = (50, 8) \)
\( \Delta x = 20, \Delta y = 8 \), \( p_0 = 2 \Delta y - \Delta x = 6 \), \( 2 \Delta y = 16, \Delta y = \Delta x = 4 \)

<table>
<thead>
<tr>
<th>( k )</th>
<th>( P_k )</th>
<th>( (x_k, y_k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>(21, 11)</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>(23, 10)</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
<td>(23, 10)</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>(25, 14)</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>(27, 18)</td>
</tr>
</tbody>
</table>

Discussion:

- Polylines:
  - Parallelism: split line into \( n \) segments and assign each processor a group of pixels
  - Bresenham's Alg. extends to circles/ellipses and other constructs (line)
  - Line width: draw \( n \) pixels per segment and split \( n \) sides of line (1, 9)
  - Line style: \( x \) pixels on \( x \) pixels off
  - Pen/brush styles are similar to width
III. Polygon Fill

A. Preliminaries

1. Polygon is defined by an ordered set of vertices (typically clockwise order).

\[ P = \{v_1, v_2, \ldots, v_n\} \]

This defines edges \[ \{v_i, v_{i+1}\} \] with a final edge \[ \{v_n, v_1\} \]

we will only consider simple polygons; those without holes.

2. Problem: Which pixels do we color?


4. Odd-even rule: Any point \( P \) is interior to a polygon if extending an infinite ray in any direction crosses an odd number of polygon segments. An even number of crossings yields an exterior point.

B. General Scan-line Polygon Fill Algorithm.

1. Idea: on each scan line, determine and sort all edge crossings. Pixels in-between crossings will be set to the fill color.

However, some complexities exist; If a scan line hits a vertex it intersects two edges and can mess up the odd-even rule.

To solve, look at topological differences. Ask class how to solve.

a. If both intersections are same side, scan line: count twice

b. If they are on opposite sides: count once

c. Identify vertices by searching polygon in order. If two points are monotonically increasing then count once, otherwise a local minimum maximum is found and count twice.

I can also solve by "shortening" appropriate polygon edges. Always choose lower edge.

II. We can also look for coherence properties to reduce properties. Successive scan line intersections are nearby and can be updated incrementally using integer arithmetic similar to Bresenham's line algorithm. (see 6.10 for details.)

\[ x_{k+1} = x_k + \frac{1}{m} \]
\[ y_{k+1} = y_k + 1 \]

iii. Can do in parallel. Each processor gets different sector of scan lines.
V. For efficient polygon fill we can store polygon in sorted-edge table, sorted by bucket sort of smallest y-value on each edge. List creates events for a plane-sweep algorithm.

a. Plane-sweep is algorithm technique in geometry whereby a plane/line is swept through a volume/plane and critical events such as start or end of line are tracked and analyzed.

b. Example

C. Algorithm.

Process scan lines from bottom to top storing an active-edge list. At plane-sweep events, update active-edge list.

V. Algorithm can be considerably simplified for convex polygons, e.g., triangles. The reason is that all pixels will be contiguous on scan line. Also can be extended to curved boundaries.

C. Boundary-Fill Algorithm — similar to "paint program's fill tool.

i. Input is starting pixel, boundary color, interior color.

ii. Idea: Start at initial pixel and set pixel colors afterward until boundary color is detected. Can consider neighboring pixels as central direction of all 8 neighboring cells.

iii. Code: boundaryFill(int x, int y, int sill, int border) { 

1. int interior color;
2. getPixel(x, y, interior);
3. if interior != border && interior != fill) {
4. set (x, y, fill);
5. boundaryFill(x+1, y, sill, border);
6. boundaryFill(x-1, y, sill, border);
7. boundaryFill(x, y+1, sill, border);
8. boundaryFill(x, y-1, sill, border);
9. } 

Downside: *Ask class*

- recursive depth *Ask thoughts on filling.*
iv. Better stack usage: store beginning of scan line and fill entire scan at a time.
   a. Example

   Call Stack
   BF(2)  BF(1)

   BF(7)  BF(1)

   BF(6)  BF(5)  BF(4)  BF(1)

   D. Flood-Fill - For recalloly. Not based on boundary color. Same algorithm with
   simpler condition as Boundary-Fill. Same issues and how to solve it.

   E. Fill styles - need reference point of pattern
      - Blended regions also possible (6-14)

   IV. Anti-aliasing
   A. Aliasing - Distortion of information due to low-frequency sampling (undersampling)
      see lines of stair-step appearance
      e.g. Perimeter function

      X means sample
      is approximated line.

   B. Anti-aliasing methods compensate for this process.

   C. Option E - higher resolution! Ok, but effect is still present

   D. Anti-aliasing method 1 - Increase sampling and display at lower resolution, called supersampling
      or post-sieving. Essentially subdivide pixels into subpixels and determine final pixel accurately,
      e.g. Line segments.

      4. Greenbaum's Algorithm applied here
      a. Each pixel can have 0, 1, 2, or 3 subpixels filled. Choose
         intensity value of pixel in relation to line color.
      b. Can also increase line width to 3 and have 9 intensity values.
      c. Can give different pixels as well.
E. Antialiasing method 2 - area sampling (postfiltering) - Determine overlap of object displayed

in e.g. Straight line - Determine proportion of pixel overlapped by line. Use percentage to change color intensity.

F. Can easily be extended to polygons, nicely.