ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

DEPARTMENT OF SOFTWARE ENGINEERING

Computer Graphics Assignment

Title: Graphics premitives

Name: Michael Getu

Id: ETS1061/14

SUBMITTED TO: INST. Lelise Daniel

SUBMISSION DATE: 12/05/2024

Geometry Line Attributes in Computer Graphics

- The **line** is one of the major inbuilt functions in computer graphics. It helps to create more interactive and visually interesting images.
- As the line is a function, it has attributes or arguments that assist in drawing a line at the desired position.
- There are primarily **four coordinates**:
    - Two for the **starting point**.
    - Two for the **ending point**.

Syntax:

```
line(int X1, int Y1, int X2, int Y2);
```

Types of Lines in Computer Graphics

Users can derive three main types of lines by adjusting the attribute values:

1. **Horizontal Line:**
   Example: line(100, 300, 400, 300);
   (Y-coordinates remain the same.)
2. **Vertical Line:**
   Example: line(600, 200, 600, 400);
   (X-coordinates remain the same.)
3. **Tangent Line:**
   Example: line(250, 400, 450, 500);
   (Both X and Y coordinates vary.)

Steps to Draw a Line in Computer Graphics

1. **Include the graphics library:**
   The graphics.h library is required to facilitate graphical operations.
    - It supports drawing shapes, displaying text, animations, and games.
2. **Set the color of the line:**
   Use the setcolor() function to define the line color.
3. **Define line attributes based on the type of line:**
    - **Horizontal Line:** Keep Y-coordinates constant, while the distance depends on the difference between X1 and X2.
    - **Vertical Line:** Keep X-coordinates constant, while the distance depends on the difference between Y1 and Y2.

- **Tangent Line:** Both X and Y coordinates must differ, and the slope defines the distance.

4. **Draw the line using the line() function.**
5. **Close the graph:** Use the closegraph() function to terminate the graphics program.

Examples of Line Drawing

1. **Example 1:**

Input: x1 = 150, y1 = 150, x2 = 450, y2 = 150

Result: Draws a horizontal line.

2. **Example 2:**

Input: x1 = 150, y1 = 200, x2 = 450, y2 = 200

Result: Draws another horizontal line.

3. **Example 3:**

Input: x1 = 150, y1 = 250, x2 = 450, y2 = 250

Result: Draws a horizontal line at a different Y-coordinate.

Why Use Lines in Computer Graphics?

1. **Separating Borders:** Helps define boundaries in a design.
2. **Emphasizing Edges:** Assists in distinguishing edges or shapes in an image.
3. **Conveying Tone:** Lines can suggest a mood or tone in the work.
4. **Drawing Attention:** Directs focus to specific parts of an image.
5. **Decorative Elements:** Enhances the design with varying line weights and colors.
6. **Dividing Space:** Segments the graphical space into distinct sections.

Bresenham's Line Drawing Algorithm

- Introduced by **Jack Elton Bresenham**.
- It is an efficient line-drawing algorithm because it uses **integer operations** (addition, subtraction, multiplication).
- It determines the next pixel based on the **minimum distance** from the true line.

Steps of Bresenham's Algorithm

1. **Start the algorithm.**
2. Declare the variables: x1, x2, y1, y2, d, dx, dy, i1, i2.
3. Input the starting point (x1, y1) and the ending point (x2, y2).
4. Calculate:

- dx = x2 - x1
- dy = y2 - y1
- i1 = 2 * dy
- i2 = 2 * (dy - dx)
- d = i1 - dx

5. Determine the starting and ending points based on the sign of dx:
   - If dx < 0: Set (x, y) as (x2, y2) and xend = x1.
   - If dx > 0: Set (x, y) as (x1, y1) and xend = x2.
6. Generate the first point at (x, y).
7. Check if the whole line is drawn:
   - If x >= xend, stop.
8. Calculate the coordinates of the next pixel:
   - If d < 0: d = d + i1
   - If d ≥ 0: d = d + i2, increment y = y + 1.
9. Increment x = x + 1.
10. Draw the next point using (x, y).
11. Repeat steps 7-10 until the line is completed.
12. End the algorithm.

Example of Bresenham's Algorithm

Input: Starting point = (1, 1), Ending point = (8, 5)
Solution:

1. Calculate:
   - dx = x2 - x1 = 8 - 1 = 7
   - dy = y2 - y1 = 5 - 1 = 4
   - i1 = 2 * dy = 2 * 4 = 8
   - i2 = 2 * (dy - dx) = 2 * (4 - 7) = -6
   - d = i1 - dx = 8 - 7 = 1
2. Iteratively generate points:
   - Initial point: (1, 1)

- Intermediate points: (2, 2), (3, 2), (4, 3), (5, 3), (6, 4), (7, 4), (8, 5)

3. The final line connects these points.

Filled Area Primitives

Filled area primitives are methods used in computer graphics to fill shapes with colors or patterns. They are essential for rendering polygons, circles, and other shapes. Two primary approaches are used:

1. **Scan-Line Method**:
    - Calculates intersection points of scan lines with the edges of the shape.
    - Fills the shape between these intersection points.
    - Works well for simple shapes like polygons and circles.
2. **Flood-Fill and Boundary-Fill Methods**:
    - Begin at a point inside the area and spread outward to fill it until reaching a boundary or a condition.
    - Suitable for complex or irregular shapes.

Polygon Fill Areas

Polygons are a fundamental shape in computer graphics, and their interiors can be filled using specific algorithms. Key points include:

- Most graphics libraries (e.g., OpenGL) support only **convex polygons** for filling.
- Non-polygon objects (e.g., curves) can be approximated using **polygon meshes** or **surface tessellation**.

Types of Polygons

1. **Simple Polygons**:
    - All vertices lie on the same plane, and no edges intersect.
    - Examples: Triangles, rectangles.
2. **Convex and Concave Polygons**:
    - **Convex Polygon**: All interior angles are less than 180°, and any line connecting two points within the polygon lies entirely inside.
    - **Concave Polygon**: Contains at least one interior angle greater than 180° and can be divided into multiple convex polygons for processing.

Inside-Outside Tests

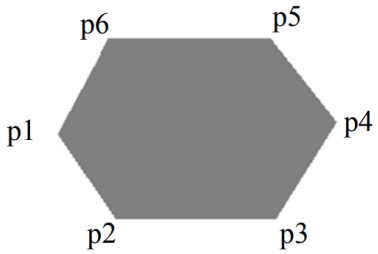
To determine if a point is inside a polygon:

- **Odd-Even Rule**: Draw a line from the point to a distant position. Count how many edges this line crosses. If the count is odd, the point is inside; if even, it's outside.

Examples of OpenGL Primitives for Filling

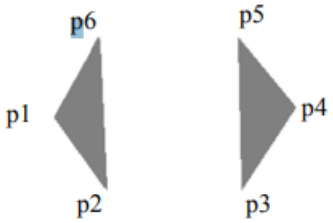
1. **GL_POLYGON**: Used to fill convex polygons with a single color.

```
glBegin(GL_POLYGON);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
glEnd();
```



2. **GL_TRIANGLES**: Represents a series of triangles for filling.

```
glBegin(GL_TRIANGLES);
        glVertex2iv(p1);
        glVertex2iv(p2);
        glVertex2iv(p3);
        glVertex2iv(p4);
        glVertex2iv(p5);
glEnd();
```

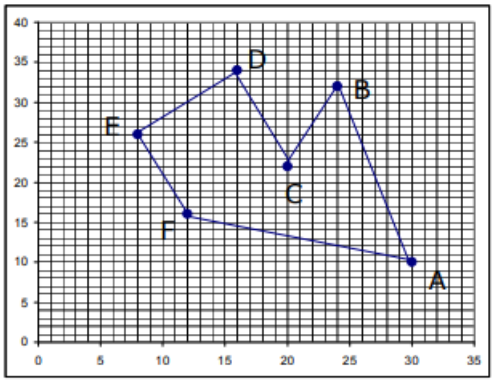


3. **Scan-Line Algorithm for Polygons**:
   - Locate the intersection of scan lines with polygon edges.
   - Fill the areas pairwise between intersections.

   Example:

   A: (30,10), B: (24,32), C: (20,22), D: (16,34)

   E: (8,26), F: (12,16)



Character Generation

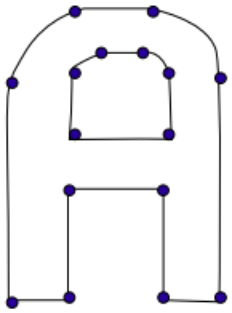
Character generation in graphics refers to the methods used to render text. Two primary methods are:

1. **Bitmap Fonts**:
   - Represent characters as a grid of pixels (on/off values).

- Simple to implement but not scalable.

```
0 0 0 0 0 0 0 0
0 0 0 1 1 1 0 0
0 0 1 1 0 1 1 0
0 1 1 0 0 0 1 1
0 1 1 0 0 0 1 1
0 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1
0 1 1 0 0 0 1 1
0 1 1 0 0 0 1 1
```

2. **Outline Fonts**:
    - Use mathematical curves and lines to define the outline of characters.
    - Scalable and flexible but computationally more complex.



Curve Generation

Curve generation is vital in computer graphics to represent smooth transitions and shapes. Common methods include:

- **Polygon Approximation**: Complex curves are represented as a series of small line segments or polygons.
- **Parametric Equations**: Represent curves mathematically using parameters to calculate points. Examples include Bézier curves and B-splines.

Example

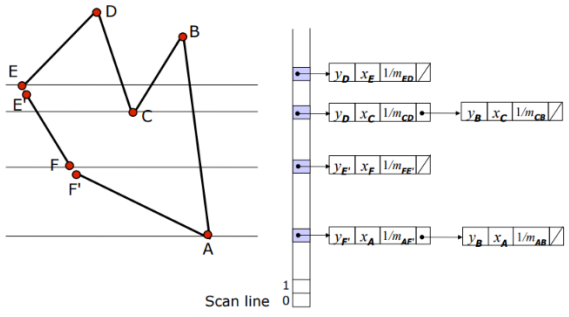
1. **Polygon Filling**: Make a (counter) clockwise traversal and shorten the single intersection edges by one pixel (so that we do not need to re-consider single/double edges).

    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



2. **Flood-Fill Algorithm**: used for cases when the boundary is not single-color. 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);
        }
}
```

3. **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
    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);
        }
}
```