Rendering

A simple X program to illustrate rendering

The programs in this directory provide a simple x based application for us to develop some graphics routines.

Please notice the following:

- All points are integers.
- The line drawing algorithm is very simple and has some definite flaws.

Syntax

Point
- point x y where x and y are integers

Line
- line x1 y1 x2 y2
- where x1, x2, y1 and y2 are integers

Other routines

- The main routine
  - main.c

- Include file with the definitions of the data structures
  - mygraph.h

- The routines to Access X functions directly.
  - Xroutines.c
  - Xroutines.h

Point and line drawing routines

- These are the primitives in the simple package!!

Point drawing routines
- point.c
- point.h

Line drawing routines
- line.c
- line.h
Line Drawing and Scan Conversion

- A preliminary step to drawing lines is choosing a suitable representation for them.
- There are three possible choices which are potentially useful.
- Explicit: $y = f(x)$
  - $y = m(x - x_0) + y_0$ where $m = \frac{dy}{dx}$
- Parametric: $x = f(t)$, $y = f(t)$
  - $x = x_0 + t(x_1 - x_0)$, $t$ in $[0,1]$
  - $y = y_0 + t(y_1 - y_0)$
- Implicit: $f(x,y) = 0$
  - $F(x,y) = (x-x_0)dy - (y-y_0)dx$
  - If $F(x,y) = 0$ then $(x,y)$ is on line
  - $F(x,y) > 0$ then $(x,y)$ is below line
  - $F(x,y) < 0$ then $(x,y)$ is above line

Point and Line Drawing Routines (ctd)

- Implicit: $f(x,y) = 0$
  - $F(x,y) = (x-x_0)dy - (y-y_0)dx$
  - If $F(x,y) = 0$ then $(x,y)$ is on line
  - $F(x,y) > 0$ then $(x,y)$ is below line
  - $F(x,y) < 0$ then $(x,y)$ is above line

Rasterization or Scan Conversion

- Drawing lines on a raster grid implicitly involves approximation.
- The general process: rasterization or scan-conversion.
- What is the best way to draw a line from the pixel $(x_1,y_1)$ to $(x_2,y_2)$?
- Such a line should ideally have the following properties.
  - Straight
  - Pass through endpoints
  - Smooth
  - Independent of endpoint order
  - Uniform brightness
  - Brightness independent of slope
  - Efficient

Line Drawing - Algorithm 1

A Straightforward Implementation

```c
int x1,y1,x2,y2;    // endpoints
float y;
for (x=x1; x<=x2; x++) {
    y = y1 + (x-x1)*(y2-y1)/(x2-x1);
    SetPixel(x, y);   // y must be rounded to nearest integer
}
```

Line Drawing - Algorithm 2

A Better Implementation

```c
int x1,y1,x2,y2;    // endpoints
float dy, dx, m;    // slope and difference
for (y=y1; y<=y2; y++) {
    float x = x1 + (y-y1)*dx/dy;  // use slope and difference
    SetPixel(x, y);     // x must be rounded to nearest integer
}
```

Line Drawing Algorithm Comparison

- Advantages over Algorithm 1
  - Eliminates multiplication
  - Improves speed
- Disadvantages
  - Round-off error builds up
  - Get pixel drift
  - Rounding and floating point arithmetic still time consuming
  - Works well only for $|m| < 1$
  - Need to loop in $y$ for $|m| > 1$
  - Need to handle special cases
Point and line drawing routines

Implicit: \( f(x, y) = 0 \)
- \( f(x, y) = (x-x_0)dy - (y-y_0)dx \)
- if \( f(x, y) = 0 \) then \((x, y)\) is on line
- if \( f(x, y) > 0 \) then \((x, y)\) is below line
- if \( f(x, y) < 0 \) then \((x, y)\) is above line

Line Drawing - Midpoint Algorithm

The Midpoint or Bresenham's Algorithm
- The midpoint algorithm is even better than the above algorithm in that it uses only integer calculations. It treats line drawing as a sequence of decisions. For each pixel that is drawn the next pixel will be either E or NE, as shown below.

Midpoint Algorithm
- The midpoint algorithm makes use of the implicit definition of the line, \( f(x, y) = 0 \). The N/NE decisions are made as follows.
- \( d = F(x_p + 1, y_p + 0.5) \)
- if \( d < 0 \) line below midpoint choose E
- if \( d > 0 \) line above midpoint choose NE
- if E is chosen
  - \( d_{new} = F(x_p + 2, y_p + 0.5) \)
  - \( d_{new} - d_{old} = F(x_p + 2, y_p + 0.5) - F(x_p + 1, y_p + 0.5) \)
  - \( \Delta d = d_{new} - d_{old} = dy \)
- If NE is chosen
  - \( d_{new} = F(x_p + 2, y_p + 1.5) \)
  - \( \Delta d = dy - dx \)

Initialization
- \( d_{start} = F(x_0 + 1, y_0 + 0.5) = (x_0 + 1)dy - (y_0 + 0.5)dx \)
- \( \Delta d' = 2dy - dx \)

Integer only algorithm
- \( F'(x, y) = 2F(x, y) \); \( d' = 2d \)
- \( d'_{start} = 2dy - dx \)
- \( \Delta d' = 2 \Delta d \)

Midpoint Algorithm for \( x_1 < x_2 \) and slope \( \leq 1 \)

```c
drawline(x1, y1, x2, y2, colour)
int x1, y1, x2, y2, colour;
{ int dx, dy, incE, incNE, x, y;
dx = x2 - x1;
dy = y2 - y1;
d = 2*dy - dx;
incE = 2*dy;
incNE = 2*(dy - dx);
y = y1;
for (x=x1; x<=x2; x++) {
    setpixel(x, y, colour);
    if (d>0) {
        d = d + incNE;
y = y + 1;
    } else {
        d = d + incE;
    }
}
```

General Bresenham's Algorithm
- To generalize to lines with arbitrary slope
  - consider symmetry between various octants and quadrants
  - for \( m > 1 \), interchange roles of \( x \) and \( y \), that is step in \( y \) direction, and decide whether \( x \) value is above or below line
  - if \( m > 1 \), and right endpoint is the first point, both \( x \) and \( y \) decrease. To ensure uniqueness, independent of direction, always choose upper (or lower) point if the line goes through the mid-point
  - handle special cases without invoking algorithm: horizontal, vertical and diagonal lines
**Additional Issues**

- End-point order
  - cannot just interchange end-points
  - does not work when we use line styles since we need the pattern to go the same way on all segments of a polygon
- later we must deal with issue of clipping
- varying the intensity of a line with the slope
  - consider horizontal line and diagonal line
  - both have same number of pixels
  - diagonal $\sqrt{2}$ times horizontal line in length
  - intensity per unit length less for diagonal