Scan Conversion 1: Lines, Circles and Fractals
Greg Humphreys
CS445: Intro Graphics
University of Virginia, Fall 2003

A DDA Line Drawing Function

```c
Line(int x1, int y1, int x2, int y2)
{
    int dx = x1 - x2, dy = y2 - y1;
    int n = max(abs(dx),abs(dy));
    float dt = n, dxdt = dx/dt, dydt = dy/dt;
    float x = x1, y = y1;
    while (n--)
    {
        DrawPoint( round(x), round(y) );
        x += dxdt;
        y += dydt;
    }
}
```

What's bad about this?

We Can Do Better Than That...

• Get rid of all those nasty floating point operations
• The idea: find the next pixel from the current one

The Key

• We’re only ever going to go right one pixel, or up and right one pixel (if the slope of the line is between 0 and 1). Call these two choices “E” and “NE”
• Let’s think of pixels as “lattice points” on a grid
• Given an X coordinate, we only need to choose between y and y+1 (y is the Y coordinate of the last pixel)

The Midpoint Test

• Look at the vertical grid line that our line intersects
• On which side of the midpoint (y+1/2) does the intersection lie?
• If it’s above the midpoint, go NE, otherwise go E
Implicit Functions

- Normally, a line is defined as \( y = mx + b \)
- Instead, define \( F(x, y) = ax + by + c \), and let the line be everywhere where \( F(x, y) = 0 \)
- Now, if \( F(x, y) > 0 \), we’re “above” the line, and if \( F(x, y) < 0 \), we’re “below” the line

Who Cares?

- We can evaluate the implicit line function at the midpoint to figure out where to draw next!
- In fact, we can use the last function evaluation to find the next one cheaply!
- For ANY \( x, y \):
  \[
  F(x+1, y) - F(x, y) = a \\
  F(x+1, y+1) - F(x, y) = a + b
  \]

Midpoint Algorithm

```java
void Line(int x1, int y1, int x2, int y2)
{
  int dx = x2 - x1, dy = y2 - y1;
  int e = 2*dy - dx;
  int incrE = 2*dy, incrNE = 2*(dy-dx);
  int x = x1, y = y1;
  DrawPoint(x, y);
  while (x < x2)
  {
    x++;
    if (e <= 0) { e += incrE; }
    else { y++; e += incrNE; }
    DrawPoint(x, y);
  }
}
```

- “\( e \)” holds the implicit function evaluation at each \( x \) value (actually, it’s multiplied by 2, but all we care about is the sign).
- Easy extension for lines with arbitrary slopes.

Midpoint Algorithm for Circles

- Only consider the second octant (the others are symmetric anyhow)
- Midpoint test still works: do we go right, or right and down?

More Midpoint Circle Drawing

- The circle’s implicit function (\( r \) is the radius):
  \[
  F(x, y) = x^2 + y^2 - r^2
  \]
- Once we know the value of the implicit function at one midpoint, we can get the value at the next midpoint by the same differencing technique:
  - If we’re going E: \( F(x+2, y) - F(x+1, y) = 2x + 3 \)
  - If we’re going SE: \( F(x+2, y-1) - F(x+1, y) = 2(y-x) + 5 \)
Drawing the 2nd Octant

```c
void Circle( int cx, int cy, int radius )
{
    int x = 0, y = radius, e = 1-radius;
    DrawPoint( x + cx, y + cy );
    while (y > x)
    {
        if (e < 0) // Go "east"
        {
            e += 2*x + 3;
        }
        else // Go "south-east"
        {
            e += 2*(x-y) + 5;
            y--;
        }
        x++;
        DrawPoint( x + cx, y + cy );
    }
}
```

- Look at all those expensive multiplies! Can we get rid of them too?
- How about doing finite differencing on the "e" variable itself?
  - If we go E: \( e_{new} - e_{old} = 2 \)
  - If we go SE: \( e_{new} - e_{old} = 4 \)

Final Circle Drawer

```c
void Circle( int cx, int cy, int radius )
{
    int x = 0, y = radius, e = 1-radius;
    int incrE = 3, incrSE = -2*radius + 5
    DrawPoint( x + cx, y + cy );
    while (y > x)
    {
        if (e < 0) // Go "east"
        {
            e += incrE;
            incrE += 2; incrSE += 2;
        }
        else // Go "south-east"
        {
            e += incrSE;
            incrE += 2; incrSE += 4;
            y--;
        }
        x++;
        DrawPoint( x + cx, y + cy );
    }
}
```

This looks pretty fast!

Flood Fill

- The idea: fill a "connected region" with a solid color
- Term definitions:
  
  ![4-connected Flood Fill](image)

  - The center "1" pixel is 4-connected to the pixels marked "4", and 8-connected to the pixels marked "8"

Simple 4-connected Fill

- The simplest algorithm to fill a 4-connected region is a recursive one:

  ```c
  FloodFill( int x, int y, int inside_color, int new_color )
  {
      if (GetColor( x, y ) == inside_color)
      {
          SetColor( x, y, new_color );
          FloodFill( x+1, y  , inside_color, new_color );
          FloodFill( x-1, y  , inside_color, new_color );
          FloodFill( x,   y+1, inside_color, new_color );
          FloodFill( x,   y-1, inside_color, new_color );
      }
  }
  ```

A Span-based Algorithm

- Definition: a run is a horizontal span of identically colored pixels

  ![Span-based Algorithm](image)

  Start at pixel “s”, the seed. Find the run containing “s” (“h” to “a”). Fill that run with the new color. Then search every pixel above that run, looking for other pixels of the interior color. For each one found, find the right side of that run (“c”), and push that on a stack. Do the same for the pixels below (“d”). Then pop the stack and repeat the procedure with the new seed. The algorithm finds runs ending at “e”, “f”, “g”, “h”, and “i”

Fractals

- Fractal: any curve or shape exhibiting self-similarity at any scale
- Fractals appear in nature all the time
- Part of the general science of chaos
The Koch Curve

- Start with a line
- Replace the line with four lines, and repeat:

Properties of the Koch Curve

- It occupies a finite area in the plane
- It has infinite length
- It is self-similar
- It has a dimension of $\log_3 4$

- It’s easier to draw in polar coordinates (hint, hint…)

Non-Integral Dimension???

- Let’s look at other “self-similar” shapes:

<table>
<thead>
<tr>
<th>Shape</th>
<th>#Parts</th>
<th>Scale</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>@</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>4</td>
<td>3</td>
<td>$\log_3 4 \approx 1.26$</td>
</tr>
</tbody>
</table>

$p = S^D \Rightarrow D = \log_p p$

Self Similar Curves in Nature

- How long is a coastline?
  - Measure on a map
  - Walk around big boulders
  - Measure each grain of sand
- Cellular structure of leaves
- Blood vessel structure
- Ethernet packet traffic (okay, that’s not really Nature)

Drawing Fractal Trees

```c
#define ANGLE 45

void tree(float X1, float Y1, float R, float Theta, int Level) {
    float SX, SY, EX, EY;
    if (Level > 0) {
        EX = X1 + (R * cos(Theta));
        EY = Y1 + (R * sin(Theta));
        line(X1, Y1, EX, EY);
        SX = X1 + ((R * 0.5) * cos(Theta));
        SY = Y1 + ((R * 0.5) * sin(Theta));
        tree(SX, SY, R*0.8, Theta-ANGLE, Level-1);
        tree(SX, SY, R*0.8, Theta+ANGLE, Level-1);
    }
}
```

Fractal Trees
Better Fractal Trees

```c
#define RT_ANGLE 45
#define LFT_ANGLE 65

void non_sym_tree (float X1, float Y1, float R, float Theta, int Level)
{
    float SX, SY, EX, EY;
    if (Level > 0)
    {
        EX = X1 + (R * cos(Theta));
        EY = Y1 + (R * sin(Theta));
        line (X1, Y1, EX, EY);
        SX = X1 + ((R * 0.4) * cos(Theta));
        SY = Y1 + ((R * 0.4) * sin(Theta));
        non_sym_tree (SX, SY, R*0.8, Theta-RT_ANGLE, Level-1);
        SX = X1 + ((R * 0.6) * cos(Theta));
        SY = Y1 + ((R * 0.6) * sin(Theta));
        non_sym_tree (SX, SY, R*0.8, Theta+LFT_ANGLE, Level-1);
    }
}
```

Non Symmetric Tree

Even Better Trees

- The tree still doesn’t really look like a tree.
- To get a better looking picture, we should apply randomness to the recursion.
- Allow the length of branches to vary (between .4 and .6 of the parent branch, say).
- Allow the angle of the branch to vary (between 35 and 65 angles, say).
- Allow the join point of the branch to the parent to vary.

Other Fractal Curves

- C-curve (dimension = 2!)

Julia Sets

- Each 2D point can be thought of as a complex number \((x + yi)\).
- Repeatedly replace each point \(z\) with \(z^2 + c\) until the point goes to infinity or comes to a “fixed point.”
- Color the point based on how long it took the magnitude of the point to become > 2.

Julia Set for \(C = .3+.6i\)
Mandelbrot Set

- Almost like the Julia set, but use the original \((x, y)\) point instead of a constant \(c\)

Julia Set for \(C = .2 + .6i\)

Julia Set for \(C = .1 + .6i\)