Raster Graphics Algorithms

- Scan conversion
- Line rasterization
- Scan fill polygon line algorithm
- Midpoint circle algorithm
- Midpoint ellipse algorithm and more
- Filled primitives

Overview of Graphics Pipeline

Frame-buffer Model

- Raster Display: 2D array of picture elements (pixels)
- Pixel can be set (grayscale/color)
- Window coordinates: pixels centered at integers

2D Scan Conversion

- Geometric Primitives:
  - 2D: point, line, polygon, circle, ...
  - 3D: point, line, polyhedron, sphere, ...
- Problem: Primitives are continuous, screen is discrete

2D Scan Conversion (2)

- Solution: compute discrete approximation
- Scan Conversion:
  Algorithms for efficient generation of the samples comprising this approximation

Line Rasterization

- Scan converting 2D line segments
- Input: segment end points as integer values:
  \((x_1, y_1), (x_2, y_2)\)
- Output: set of pixels lighten up
Line Rasterization: Basic Math Review

- **Slope**: \( m = \frac{y_2 - y_1}{x_2 - x_1} \)
- **Solving for \( y \)**:
  \[ y = y_1 + \left( \frac{y_2 - y_1}{x_2 - x_1} \right) (x - x_1) \]
- **Parametric form of line equation**:
  \[ x = x_1 + t(x_2 - x_1), \quad y = y_1 + t(y_2 - y_1) \]

Basic Line Algorithms

- **Must**:
  - Computer integer coordinates of pixels which lie on or near a line
  - Be efficient
  - Create visually satisfactory images:
    - Lines should appear straight
    - Lines should terminate accurately
    - Lines should have constant density
    - Lines' density should be independent of length and angle
  - Always be defined

A Naïve Algorithm

- **Problems**:
  - Computing \( y \) value is expensive, for every \( x \):
    - 1 floating point multiplication
    - 1 floating point addition
    - 1 call to rounding function
  - Exception around vertical lines
  - Gets disconnected (strung out) for \( m > 1 \)

Basic Incremental Algorithm

- **Also called**: DDA
- **Based on parametric equation of line**:
  - Compute \( \Delta x \) and \( \Delta y \) so that one of them is 1 and the other is less than 1
  - Start from \((x_1, y_1)\)
  - In each step, increment \( x \) by \( \Delta x \) and \( y \) by \( \Delta y \)
  - Set Pixel (Round\( x \), Round\( y \))
Basic Incremental Algorithm (2)

```c
void DDA (int x1, int y1, int x2, int y2) {
    int length;
    float x, y, dx, dy;
    if abs(x2-x1) > abs(y2-y1)
        length = abs(x2-x1);
    else
        length = abs(y2-y1);
    dx = (x2-x1)/length;
    dy = (y2-y1)/length;
    x = x1;
    y = y1;
    for (int i = 0; i <= length; i++) {
        setpixel(rnd(x), rnd(y))
        x += dx;
        y += dy;
    }
}
```

Creates good lines, but still there are problems

Bresenham’s Midpoint Algorithm

Keynote: At each step along the line, there are only two choices for the next pixel to be filled in.

The set of which two pixels should be filled in depends on the slope of the line

Bresenham’s Midpoint Algorithm (2)

For a line with a slope between 0 and 1, the two possible pixels that might be filled in, are the E and NE neighbors

What are the possible next pixels for a line with:
-1 < slope < 0
slope > 1

Bresenham’s Midpoint Algorithm (3)

Let’s consider 0 < slope < 1

We can determine the next pixel by putting a decision point half-way between the two candidate points:

The next pixel is selected based on which side of the decision point, the “true line” is on, so we can define a decision variable (d).

Bresenham’s Midpoint Algorithm (4)

In each step of the algorithm:
- determine the coordinates of current based on the decision variable computed in previous step
- compute decision variable for next step

So, How to define decision variable (d)?

Bresenham’s Midpoint Algorithm (5)

\[ F(x, y) = 0 : \text{[x, y] is on the line} \]
\[ F(x, y) > 0 : \text{[x, y] is below the line} \]
\[ F(x, y) < 0 : \text{[x, y] is on top of the line} \]

decision variable:
\[ d = F(x_m, y_m) \]

where \([x_m, y_m]\) is the coordinates of the midpoint

e.g. the value of d in current step:
\[ d = F(x_0, y_0) = (y_0 + 1/2)dx - (x_0 + 1/2)dy + B.dx \]
Bresenham’s Midpoint Algorithm (6)

\[ d_{\text{old}} = (x_p + 1) dy - (y_p + 1/2) dx + B dx \]

\[ \begin{cases} \text{case 1): } d_{\text{old}} < 0 & \Rightarrow \text{next } = E \\ \text{case 2): } d_{\text{old}} > 0 & \Rightarrow \text{next } = \text{NE} \end{cases} \]

\[ d_{\text{new}} = \left[ (x_p + 1) dy - (y_p + 1/2) dx + B dx \right] \]

\[ \Rightarrow d_{\text{new}} = d_{\text{old}} + dy \]

Bresenham’s Midpoint Algorithm (7)

\[ d_{\text{old}} = (x_p + 1) dy - (y_p + 1/2) dx + B dx \]

\[ \begin{cases} \text{case 1): } d_{\text{old}} < 0 & \Rightarrow \text{next } = E \\ \text{case 2): } d_{\text{old}} > 0 & \Rightarrow \text{next } = \text{NE} \end{cases} \]

\[ d_{\text{new}} = \left[ (x_p + 1) dy - (y_p + 1/2) dx + B dx \right] \]

\[ \Rightarrow d_{\text{new}} = d_{\text{old}} + (dy - dx) \]

Bresenham’s Midpoint Algorithm (8)

So far, we computed an incremental function for the decision variable. What about the initial decision value?

\[ d_{\text{old}} = F(x_0 + 1, y_0 + 1/2) = (y_0 + 1/2) dx + B dx \]

\[ = (x_0 + 1) dy - (y_0 + 1/2) dx + B dx \]

\[ = F(x_0, y_0) + dy - dx = 0 \]

\[ \Rightarrow d_{\text{old}} = dy - dx \]

Oh, Nooo!!! floating point 😞

Bresenham’s Midpoint Algorithm (9)

How to get ride of the ½ fraction?

Solution: We only need the sign of d, so multiply d by 2!

\[ d_{\text{new}} = 2 \times d_{\text{old}} \]

\[ d_{\text{old}} = 2 \times d_{\text{new}} \]

\[ d_{\text{new}} = 2 \times (dy - dx) \]

Bresenham’s Midpoint Algorithm (Finally the code!!)

```c
void MidpointLine(int x0, int y0, int x1, int y1)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx; // initial value of d
    int x = x0;
    int y = y0;
    int incE = 2 * dy; // increment when choose E
    int incNE = 2 * (dy - dx); // increment when choose NE
    SetPixel (x, y); // the start pixel
    while (x < x1) {
        if (d <= 0) { // choose E
            d += incE ;
            x ++;
        } else { // choose NE
            d += incNE;
            x ++;
            y ++;
        }
        SetPixel (x, y);
    }
}
```

Lines with more general slopes?

Lines with more general slopes:

1) \( dx > dy \)
2) \( dy > dx \)
3) \( dy > -dx \)
4) \( -dx > dy \)
5) \( -dx > -dy \)
6) \( -dy > dx \)
7) \( -dy > dx \)
8) \( dx > -dy \)
Lines with more general slopes? (2)

- Solution 1: compute the proper decision variable for each 8 octants:
  e.g. o2): N and NE
  o5): W and SW
  
  redeem... no thanks! we have already had enough trouble for that one octant!

- Solution 2:
  convert all octants to octant1
  same Midpoint algorithm
  convert to original octant in SetPixel
  e.g. o2): Midpoint (y, x)
  SetPixel (x, y)

Scan Converting Circles

Assume center of circle is at (0, 0):
\[ x^2 + y^2 = R^2 \]
\[ d = F(x, y) = x^2 + y^2 - R^2 \]

- \( F(x, y) = 0 \): \( (x, y) \) is on the circle
- \( F(x, y) > 0 \): \( (x, y) \) is outside the circle
- \( F(x, y) < 0 \): \( (x, y) \) is inside the circle

What if center is not at (0, 0)?

Midpoint Circle Algorithm

\[ d_{old} = F(x_{p} + 1, y_{p} - \frac{1}{2}) = (x_{p} + 1)^2 + (y_{p} - \frac{1}{2})^2 - R^2 \]

- \( d_{old} < 0 \): next = E
- \( d_{old} > 0 \): next = SE

Midpoint Circle Algorithm (2)

Midpoint Circle Algorithm (3)
Midpoint Circle Algorithm (4)

Floating Point: to be or not to be?

\[ d_{old} = d_{old} + 2x + 3 \]
\[ d_{new} = d_{new} + 2x - 2y + 5 \]

\[ d_{new} \geq 0 \]

and \[ d = d - 1 \]
\[ d < 0 \] and \[ d > 0 \] \[ x \geq 0 \] \[ y \geq 0 \]

so why not use \[ d \]!!!
\[ d' = d - 1 \]

Midpoint Circle Algorithm (5)

\[ d_{old} = d_{old} + 2x + 3 \]
\[ d_{new} = d_{new} + 2x - 2y + 5 \]

\[ E \Rightarrow x \Rightarrow \delta_{SE} = \delta_{SE} + 2 \]
\[ \delta_{SE} = \delta_{SE} + 4 \]
\[ \delta_{E} = \delta_{E} + 2 \]
\[ \delta_{E} = \delta_{E} + 4 \]

Midpoint Ellipse Algorithm (2)

\[ F(x, y) = 0 : \] \[ (x, y) \] is on the ellipse
\[ F(x, y) > 0 : \] \[ (x, y) \] is outside the ellipse
\[ F(x, y) < 0 : \] \[ (x, y) \] is inside the ellipse

Scan Conversion Ellipses

What if not centered at \((0, 0)\)?

Midpoint Ellipse Algorithm

\[ F(x, y) = a x^2 + b y^2 + c \]

Another method to test the slope:

Gradient

\[ \text{Gradient}(F) = \frac{\partial F}{\partial x}, \frac{\partial F}{\partial y} \]

\[ \text{Region 1} : \] \( \text{Gradient}(F) < 0 \)
\[ \text{Region 2} : \] \( \text{Gradient}(F) > 0 \)

Midpoint Ellipse Algorithm (6)

void MidpointCircle(int xcenter, int ycenter, int R)
{
    int x = 0;
    int y = R;
    int d = 1 - R // initial value for decision variable
    int deltaE = 3;
    int deltaSE = -2*R + 5;

    CirclePoints (x, y, xcenter, ycenter);

    while (y > x) { // slope is between -1 and 0
        if (d < 0) { // Select E
            d+= deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else { // Select SE
            d += deltaSE;
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, xcenter, ycenter);
    } // while
}

Scan Conversion Ellipses

\[ F(x, y) = 0 : \] \[ (x, y) \] is on the ellipse
\[ F(x, y) > 0 : \] \[ (x, y) \] is outside the ellipse
\[ F(x, y) < 0 : \] \[ (x, y) \] is inside the ellipse

What if not centered at \((0, 0)\)?
Midpoint Ellipse Algorithm (3)

The rest is more straightforward:
- calculate \( d_{\text{new}} \) as a function of \( d_{\text{old}} \)
- calculate the initial values
- then we have two main loops:
  - while \( a^2y > b^2x \), scan along \( x++ \)
  - while \( y > 0 \), scan along \( y-- \)

Rotated Ellipses?

\[
d_1 + d_2 = K \quad \text{(constant)}
\]
\[
dist(P,C_1) + dist(P,C_2) = K
\]
\[
\sqrt{(x-C_1_x)^2 + (y-C_1_y)^2} + \sqrt{(x-C_2_x)^2 + (y-C_2_y)^2} = K
\]
\[
F(x,y) = \sqrt{(x-C_1_x)^2 + (y-C_1_y)^2} + \sqrt{(x-C_2_x)^2 + (y-C_2_y)^2} - K
\]

Please, stop right here!!! 😓

Why stop? – We want more!

General procedure to develop a midpoint method for an arbitrary shape:
- formulate the shape (parabola, sinus, …)
- determine the decision variable
- determine the scanning path(s) so that the decision in each step is limited
- try to formulate the equations as incremental and avoid floating point operations as much as possible, or at least convert them to easier ones (mult to add, sin to mult, …)

Filled Primitives – Rectangle (Box)

for (x = x1; x <= x2; x++)
for (y = y1; y <= y2; y++)
  SetPixel (x,y, fillcolor);

? 

Filled Primitives – Circle (Dist)

void CirclePointsFill (x, y, center, xcenter, ycenter)
{
  int i;
  for (i = x+center; i <= -x + center; i++) {
    SetPixel (i,y+center, fillcolor);
    SetPixel (i,-y+center, fillcolor);
  }
  for (i = y+center; i <= -y + center; i++) {
    SetPixel (i,x+center, fillcolor);
    SetPixel (i,-x+center, fillcolor);
  }
}

Filled Primitives – Triangle

The most frequently used primitive in 3D graphics!

Just, keep track of two Bresenham lines in each step

any optimization?
yup: skip steps when \( y \) does not change!