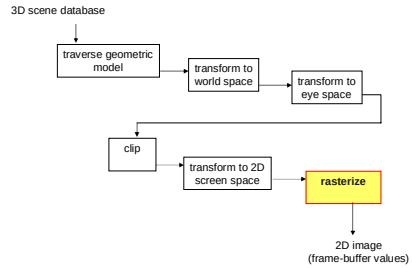
Raster Graphics Algorithms

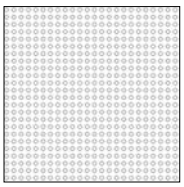
- scan conversion
- Line rasterization
- Bresenham's Midpoint line algorithm
- Midpoint circle algorithm
- Midpoint ellipse algorithm and more
- Filled primitives

Overview of Graphics Pipeline



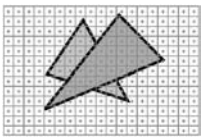
Frame-buffer Model

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



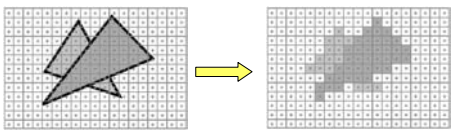
2D Scan Conversion

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



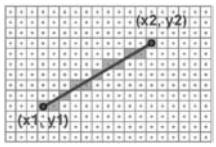
2D Scan Conversion (2)

- 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

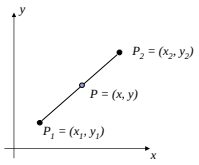
$$Slope = \frac{y_2 - y_1}{x_2 - x_1} = \frac{y - y_1}{x - x_1}$$

Solving for y

$$y = \frac{y_2 - y_1}{x_2 - x_1} x + (-\frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1)$$

or

$$y = mx + b$$



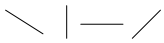
Line Rasterization: Basic Math Review

- Length of line segment between P_1 and P_2
  $L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- Midpoint of a line segment between P_1 and P_2
  $P_{mid} = (\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$
- Two lines are perpendicular iff
  $m_2 = -\frac{1}{m_1}$
- Parametric form of line equation:
  $x = x_1 + t(x_2 - x_1)$
  $y = y_1 + t(y_2 - y_1)$

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

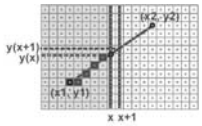
- increment x in each step
- Compute y as a function of x
- Round y (why?)
- Set Pixel (x, Rnd(y(x)))

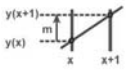
$$y = \frac{y_2 - y_1}{x_2 - x_1} x + (-\frac{y_2 - y_1}{x_2 - x_1} x_1 + y_1)$$

or

$$y = mx + b$$

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{dy}{dx}$$

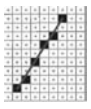




A Naïve Algorithm (2)

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: Digital Differential Analyzer (DDA)
- Based on parametric equation of line:
  - Compute Δx and Δy so that one of them is 1 and the other is less than 1
  - Start from (x1, y1)
  - in each step, increment x1 by Δx and y1 by Δy
  - Set Pixel (Rnd(x), Rnd(y))

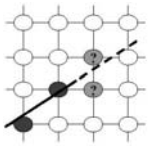
Basic Incremental Algorithm (2)

```
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

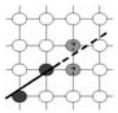
- Keynote: At each step along the line, there are only two choices for the next pixel to be filled in.

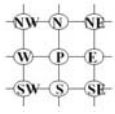


- The set of which two pixels should be filled in depends on the slop of the line

Bresenham's Midpoint Algorithm (2)

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

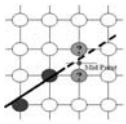




- What are the possible next pixels for a line with:
  - -1 < slop <0
  - slop > 1

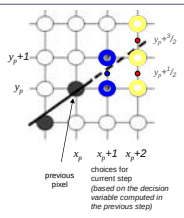
Bresenham's Midpoint Algorithm (3)

- 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)

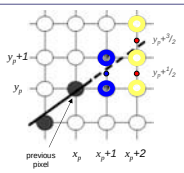


choices for next step *(based on the decision variable that will be computed in the current step)*

- 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)



$$y = mx + B = \frac{dy}{dx}x + B$$

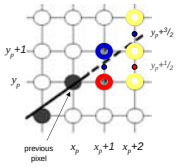
$$F(x, y) = x.dy - y.dx + B.dx$$

F (x, y) = 0 : *[x, y] is on the line*
F (x, y) > 0 : *[x, y] is below the line*
F (x, y) < 0 : *[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:
$d = F(x_p+1, y_p+1/2) = (x_p+1).dy - (y_p+1/2).dx+B.dx$

Bresenham's Midpoint Algorithm (6)



$d_{old} = (x_p+1).dy - (y_p+1/2).dx + B.dx$

case 1) $d_{old} < 0$: next = E
case 2) $d_{old} > 0$: next = NE

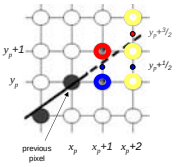
$case1) \quad d_{old} < 0 \Rightarrow next = E$

$d_{new} = F(x_p + 2, y_p + \frac{1}{2}) = (x_p + 2).dy - (y_p + \frac{1}{2}).dx + B.dx$

$d_{new} = [(x_p + 1).dy + dy] - (y_p + \frac{1}{2}).dx + B.dx$

$d_{new} = d_{old} + dy$

Bresenham's Midpoint Algorithm (7)



$d_{old} = (x_p+1).dy - (y_p+1/2).dx + B.dx$

case 1) $d_{old} < 0$: next = E
case 2) $d_{old} > 0$: next = NE

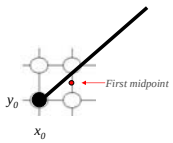
$case2) \quad d_{old} > 0 \Rightarrow next = NE$

$d_{new} = F(x_p + 2, y_p + \frac{3}{2}) = (x_p + 2).dy - (y_p + \frac{3}{2}).dx + B.dx$

$d_{new} = [(x_p + 1).dy + dy] - [(y_p + \frac{1}{2}).dx + dx] + B.dx$

$d_{new} = d_{old} + (dy - dx)$

Bresenham's Midpoint Algorithm (8)



So far, We computed an incremental function for the decision variable.

What about the initial decision value?

$d_{initial} = F(x_0 + 1, y_0 + \frac{1}{2}) = (x_0 + 1).dy - (y_0 + \frac{1}{2}).dx + B.dx$

$= [x_0.dy + dy] - [y_0.dx + \frac{1}{2}dx] + B.dx$

$= F(x_0, y_0) + dy - \frac{1}{2}dx \qquad F(x_0, y_0) = 0$

$\Rightarrow d_{initial} = dy - \frac{1}{2}dx$

Oh, Noooo!!! 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!

$d_{new(E)} = d_{old} + 2 \times dy$

$d_{new(NE)} = d_{old} + 2 \times (dy - dx)$

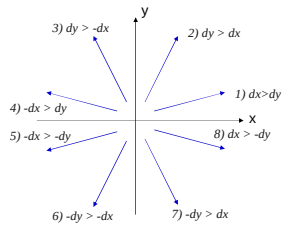
$d_{initial} = 2 \times dy - dx$

☺

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

```
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 {
      d += incNE;                      // choose NE
      x++;
      y++;
    }
    SetPixel (x, y);
  }
}
```


Lines with more general slopes?



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

  mmm… 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)

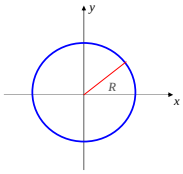
Scan Converting Circles

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*

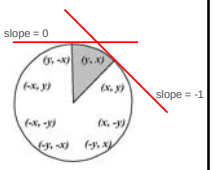


What if center is not at (0, 0)?

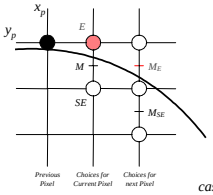
Scan Converting Circles (2)

- Take advantage of symmetries to minimize cases (slopes) and amount of curve drawing
- Only one eighth of a circle needs to be computed!

```
void CirclePoints (x, y, xcenter, ycenter)
{
  SetPixel (x + xcenter, y + ycenter);
  SetPixel (y + xcenter, x + ycenter);
  SetPixel (-x + xcenter, y + ycenter);
  SetPixel (y + xcenter, -x + ycenter);
  SetPixel (-x + xcenter, -y + ycenter);
  SetPixel (-y + xcenter, -x + ycenter);
  SetPixel (x + xcenter, -y + ycenter);
  SetPixel (-y + xcenter, x + ycenter);
}
```



Midpoint Circle Algorithm



$d_{old} = F(x_p+1, y_p-½)$
$= (x_p+1)^2 + (y_p-½)^2 - R^2$

-1< slope < 0:

case 1) $d_{old} < 0$: next = E
case 2) $d_{old} > 0$: next = SE

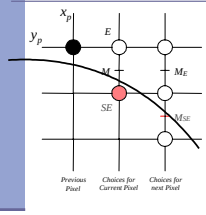
case1) $d_{old} < 0 \Rightarrow next = E$

$d_{new} = F(x_p + 2, y_p - \frac{1}{2}) = (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2$

$d_{new} = [(x_p + 1)^2 + 2x_p + 3] + (y_p - \frac{1}{2})^2 - R^2$

$d_{new} = d_{old} + 2x_p + 3$

Midpoint Circle Algorithm (2)



$d_{old} = F(x_p+1, y_p-½)$
$= (x_p+1)^2 + (y_p-½)^2 - R^2$

case 1) $d_{old} < 0$: next = E
case 2) $d_{old} > 0$: next = SE

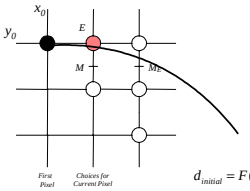
case2) $d_{old} > 0 \Rightarrow next = SE$

$d_{new} = F(x_p + 2, y_p - \frac{3}{2}) = (x_p + 2)^2 + (y_p - \frac{3}{2})^2 - R^2$

$d_{new} = [(x_p + 1)^2 + 2x_p + 3] + [(y_p - \frac{1}{2})^2 - 2y_p + 2] - R^2$

$d_{new} = d_{old} + 2x_p - 2y_p + 5$

Midpoint Circle Algorithm (3)



$d_{initial} = F(x_0 + 1, y_0 - \frac{1}{2}) = (x_0 + 1)^2 + (y_0 - \frac{1}{2})^2 - R^2$

$= [x_0^2 + 2x_0 + 1] + [y_0^2 - y_0 + \frac{1}{4}] - R^2$

$x_0 = 0, y_0 = R \Rightarrow$

$d_{initial} = \frac{5}{4} - R$ ☹

Midpoint Circle Algorithm (4)

Floating Point: to be or not to be?

$$d_{new(E)} = d_{old} + \underbrace{2x_p + 3}_{\text{int}}$$

$$d_{new(SE)} = d_{old} + \underbrace{2x_p - 2y_p + 5}_{\text{int}}$$

$$d_{initial} = \frac{5}{4} - R$$

int $d' = d - \frac{1}{4}$

$d' < 0 \Leftrightarrow d < 0$ and $d' > 0 \Leftrightarrow d > 0$

so why not use d'?!!

$d' = 1 - R$

Midpoint Circle Algorithm (5)

$$d_{new(E)} = d_{old} + \underbrace{2x_p + 3}_{\text{int deltaE}}$$

$$d_{new(SE)} = d_{old} + \underbrace{2x_p - 2y_p + 5}_{\text{int deltaSE}}$$

$$E \Rightarrow x{+}{+} \Rightarrow \begin{cases} deltaE_{new} = deltaE_{old} + 2 \\ deltaSE_{new} = deltaSE_{old} + 2 \end{cases}$$

$$SE \Rightarrow x{+}{+}, y{-}{-} \Rightarrow \begin{cases} deltaE_{new} = deltaE_{old} + 2 \\ deltaSE_{new} = deltaSE_{old} + 4 \end{cases}$$

Midpoint Circle Algorithm (6)

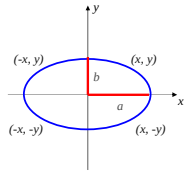
```
void MidpointCircle(int xcenter, int ycenter, int R)
{
  int x = 0;
  int y = R;
  int d = 1 - R              // initial value for decision variable
  int deltaE = 3;
  int deltaSE = -2*R + 5;

  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

$$b^2x^2 + a^2y^2 = a^2b^2$$

$$d = F(x,y) = b^2x^2 + a^2y^2 - a^2b^2$$

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*



What if not centered at (0, 0)?

Midpoint Ellipse Algorithm

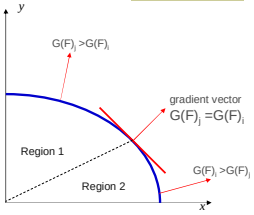
$$F(x,y) = b^2x^2 + a^2y^2 - a^2b^2$$

Another method to test the slope :
Gradient

$$Gradient(F) = \frac{\partial F}{\partial x}i + \frac{\partial F}{\partial y}j$$

$$Gradient(F) = 2b^2x\ i + 2a^2y\ j$$

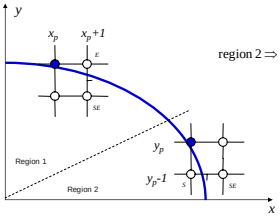
$$\begin{cases} G(F)_j > G(F)_i : 2a^2y > 2b^2x \Rightarrow \text{region 1} \\ G(F)_i > G(F)_j : 2b^2x > 2a^2y \Rightarrow \text{region 2} \end{cases}$$



Midpoint Ellipse Algorithm (2)

region 1 $\Rightarrow$ x++ $\Rightarrow \begin{cases} E \Rightarrow d_{new} = F(x_p + 2, y_p - \frac{1}{2}) \\ SE \Rightarrow d_{new} = F(x_p + 2, y_p - \frac{3}{2}) \end{cases}$

region 2 $\Rightarrow$ y--$\Rightarrow \begin{cases} S \Rightarrow d_{new} = F(x_p + \frac{1}{2}, y_p - 2) \\ SE \Rightarrow d_{new} = F(x_p + \frac{3}{2}, y_p - 2) \end{cases}$

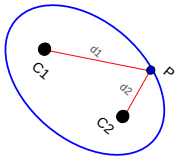


Midpoint Ellipse Algorithm (3)

The rest is more straightforward:

- calculate d_{new} as a function of d_{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?



$$d_1 + d_2 = K \text{ (constant)}$$

$$dist(P, C_1) + dist(P, C_2) = K$$

$$\sqrt{(Px - C_1x)^2 + (Py - C_1y)^2} + \sqrt{(Px - C_2x)^2 + (Py - C_2y)^2} = K$$

$$F(x, y) = \sqrt{(Px - C_1x)^2 + (Py - C_1y)^2} + \sqrt{(Px - C_2x)^2 + (Py - C_2y)^2} - K$$

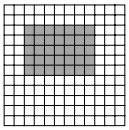
Pleeese, stop right here!!! ☹

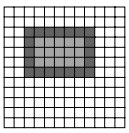
Why stop? – We want more!

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)

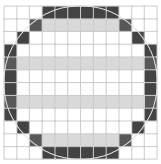




```
for (x = x1; x <= x2; x++)
  for (y = y1; y <= y2; y++)
    SetPixel (x,y, fillcolor);
```

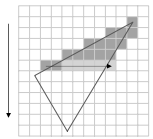
?

Filled Primitives – Circle (Dist)



```
void CirclePointsFill (x, y, 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!