Midpoint algorithm for circles

- if $p_i < 0$ then $y_{i+1} = y_i$, and

  $p_{i+1} = p_i + 2*x_{i+1} + 1$
- if $p_i > 0$ then $y_{i+1} = y_i - 1$, and

  $p_{i+1} = p_i + 2*x_{i+1} + 1 - 2*y_{i+1}$
- We need only p_0 from the very first pixel drawn at (0,R):

  $p_0 = F(1, R-0.5) = 1.25 - R$
- Note 1: p_i is incremented/decremented by integer values, hence round off $p_0 = round(p_0)$; if R integer then start with $p_0 = 1-R$
- Note 2: p_i is incremented with $2*x_i$ and $2*y_i$, hence $2*x_{i+1} = 2*x_i + 2$, $2*y_{i+1} = 2*y_i - 2$ or $2*y_i$

Computer Graphics

Spring 2009, #1

2D Graphics Primitives

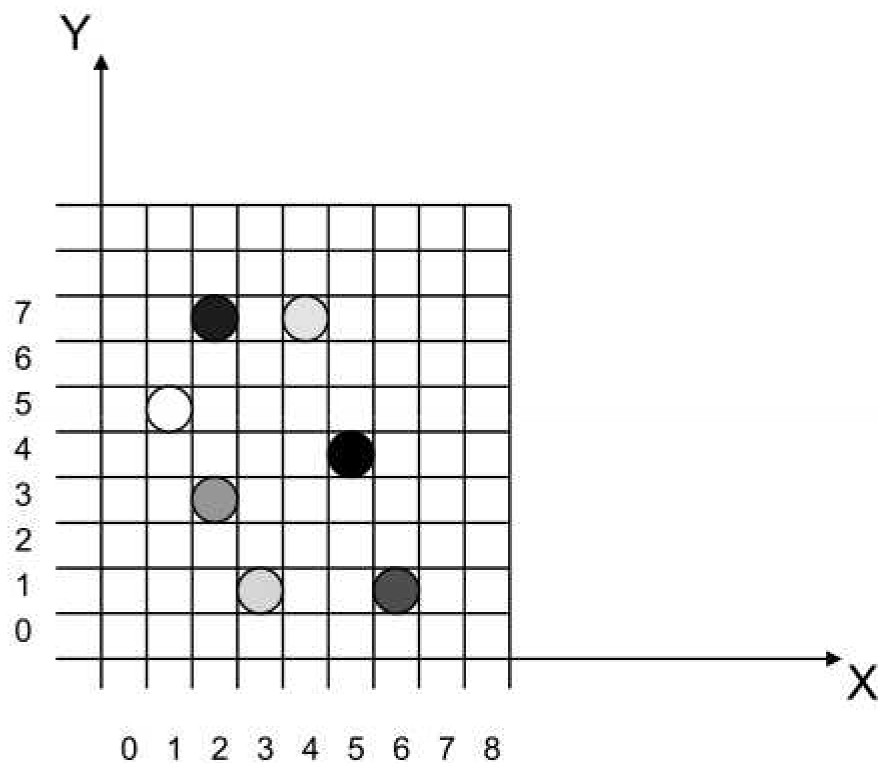
Contents

- Pixels
- Straight lines
- Line drawing algorithms, straight
  - Digital Differential Analyzer DDA
  - Bresenham
- Second order curves
  - midpoint algorithm

Pixels

- A pixel is the basic entity in computer graphics
- Represents a 0-dimensional geometrical point at position (x,y) with zero length, zero area, zero volume, ...
- A pixel has several properties
  - Pixel position $(x_{int}, y_{int}) = (round(x), round(y))$ where $round(x) = (int)(x+0.5)$
  - Colours (R,G,B)
  - Intensity
  - Has non-zero extension

Pixel Coordinate Systems



Pixels (1,5), (1.8,7.1), etc.

Straight Lines

- A line represents a 1-dimensional geometrical object with non-zero length, zero area, zero volume, ...
- A straight line **s** has two end points $\mathbf{p_1}$, $\mathbf{p_2}$ and a collection of points connecting the two end points
- Geometrically a line is defined using the parametric representation

$$s = p_1 + t*(p_2-p_1)\ ,\ t \in [0,1]$$

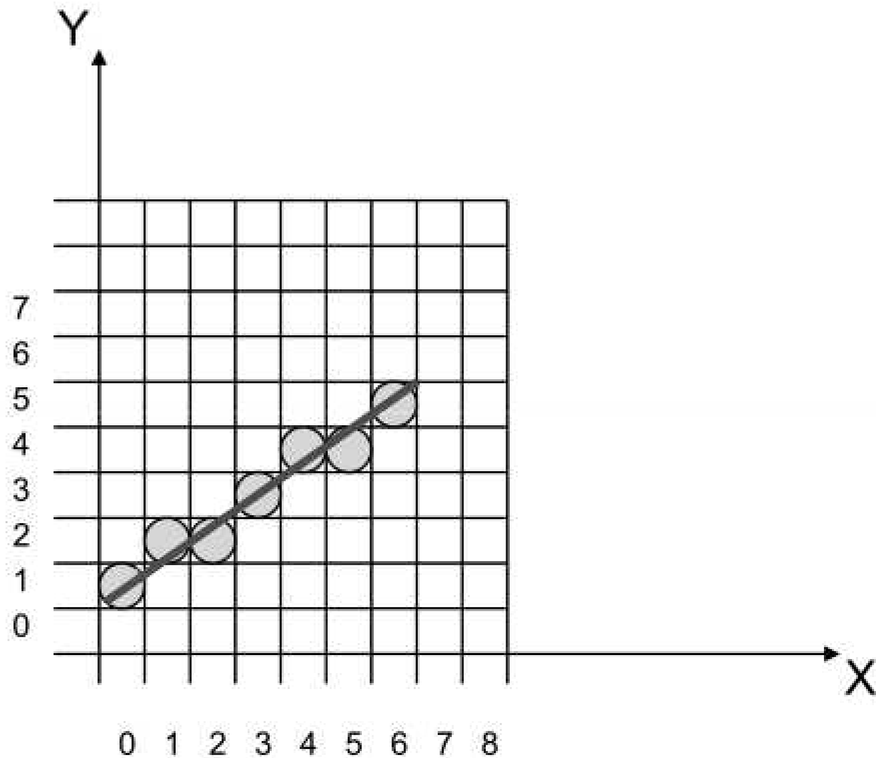
or an equation with constants a,b,c and endpoints:

$$a*x + b*y + c = 0 \text{ and } p_1,p_2$$

From Straight Lines to Pixels

- How can we draw a straight line using pixels?
- Requirements: The line
  - should be straight!
  - have fixed width (constant intensity)
  - have no jags (steps)
  - have well defined starting and end points
  - have the same intensity independent of angle w.r.t. the coordinate axes
  - be drawn quickly: integer arithmetic, additions and subtractions, possibly bit shifts

From Straight Lines to Pixels



$3*Y - 2*X - 3 = 0$: from pixel (0,1) to pixel (6,5)

DDA: Digital Differential Analyzer

- Basic idea: Calculate the (x,y)-coordinates of the pixels for the line using the parametric representation for the line and scan t from 0 to 1.

$$(x(t) , y(t)) = (p_{1x},p_{1y}) + t*(p_{2x}-p_{1x},p_{2y}-p_{1y})$$

$$= (x_1,y_1) + t*(x_2-x_1,y_2-y_1)$$

$$x(t) = x_1 + t*\Delta X$$

$$y(t) = y_1 + t*\Delta Y = y_1 + m*(t*\Delta X)$$

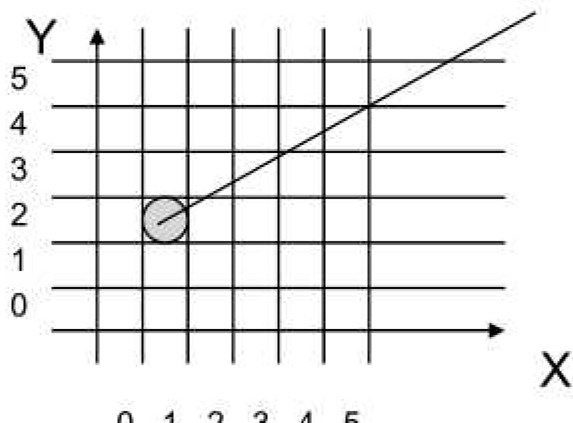
where $m = \Delta Y/\Delta X$

DDA: Digital Differential Analyzer

- Use suitable incremental values for t, calculate (x(t),y(t)) and round off to ints.
- Simplified version:
  - if $|m| \leq 1$ then increment $x \rightarrow x + 1$, that is, scan over t_n such that $t_n * \Delta X = 0,1,2,$ are integers; and increment y with m
  - if $|m| > 1$ then increment $y \rightarrow y + 1$, that is, scan over t_n such that $m * t_n * \Delta X = 0,1,2,$ are integers, and increment x by $1/m$.
- Potential problems:
  - floating point division at start up
  - floating point additions when incrementing
  - rounding at every increment

Bresenham algorithm

- For $|m| < 1$, x-increments are integer valued
- Try to eliminate rounding off floating point values by detecting when the rounding off gives a value which is different from the previously rounded off value
- Example: Will the next point be (2,2) or (2,3)?



Bresenham algorithm, $|m| < 1$

- x increases in integer steps: x = round(x)
- y has rounding "errors" d = round(y)-y
  $d \in (-0.5, 0.5]$
- Denote previously drawn pixel by
  (x, y') = (round(x),round(y)) = (x,y+d)
- Next point will be (x+1,y+m) which rounded off yields

$$\text{round}(x+1) = x+1$$

$$\text{round}(y+m) = \begin{cases} y' & \text{if } d+m < 0.5 \\ y'+1 & \text{if } d+m \geq 0.5 \end{cases}$$

Bresenham algorithm

- Bresenham: Enough to monitor the sign of

$$d + m - 0.5 \equiv s$$

- At the starting point d = 0 $\Rightarrow$ s = m - 0.5
- Algorithm:

```
s = m – 0.5; y' = round(y');
loop {
        s = s + m;
        if ( s > 0.0 )
                {
                s = s - 1.0;
                y'= y'+1;
                }
    }
```

Bresenham algorithm

- Computational trick: switch to integer arithmetic by scaling everything with $2*\Delta X$
- Scaled start up value $s*2*\Delta X = p = 2\Delta Y - \Delta X$

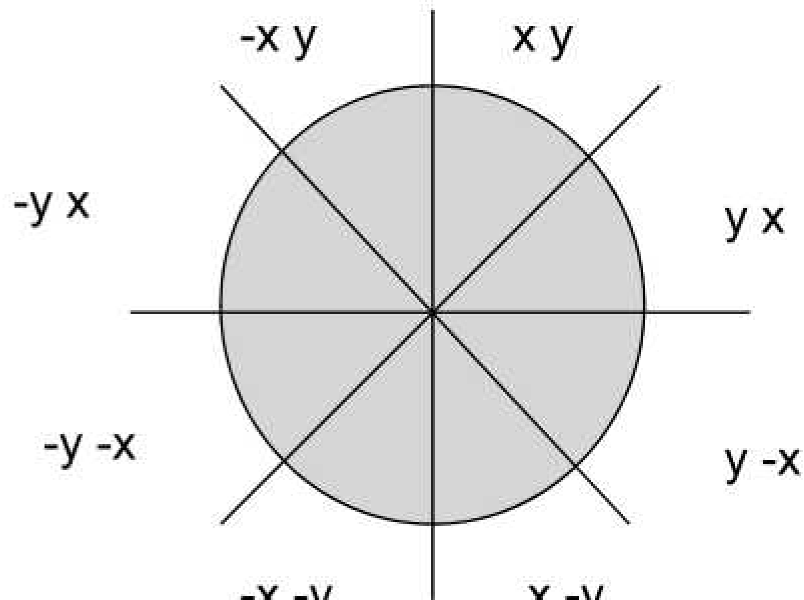
```
loop {
        p = p + 2*ΔY;
        if ( p > 0 )
                {
                p = p – 2*ΔX;
                y’ = y’ + 1;
                }
        }
```

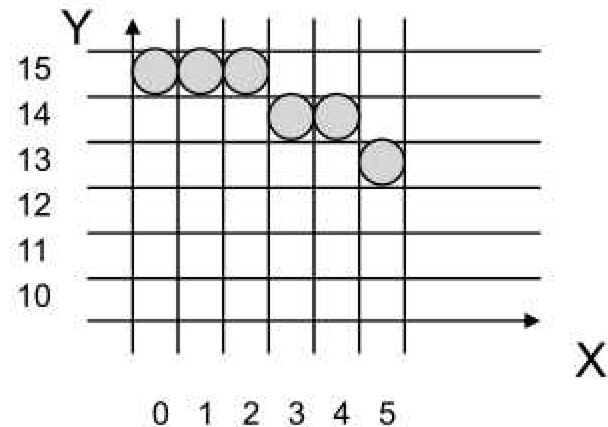
Bresenham algorithm

- Advantages:
  - no floating point divisions
  - no floating point numbers
  - only integer multiplication by 2, i.e. shift left!
- Note:
  - if $\Delta X < 0$ switch $p_1 \Leftrightarrow p_2$ or reprogram with diminishing x and y (subtract m or $2 * \Delta Y$)
  - if $|m| > 1$ switch the roles of $x \Leftrightarrow y$

2nd Order Curves

- Curves which are algebraically of second order (circle, ellipse, parabola, some splines) have special algorithms
- Example: for circles use octant symmetry





Midpoint algorithm for circles

- Assume that the circle in centered at the origin (0,0)
- Let (x_i,y_i) be the last drawn pixel.
- Now study $(x_{i+1},y_{i+1}) = (x_i+1,y_{i+1})$ and decide whether (x_i+1,y_i) or (x_i+1,y_i-1) will be chosen.
- Circle equation: $F(x,y) = x^2 + y^2 - R^2 = 0$
- For any point (x,y): if inside the circle then $F(x,y) < 0$, if outside the circle $F(x,y) > 0$

Midpoint algorithm for circles

- Introduce a decision variable p_i to be evaluated at (x_i,y_i) to decide where the next pixel will be.
- p_i evaluates $F(x,y)$ at the midpoint between y_i and y_i-1 for $x_{i+1} = x_i+1$:

$$p_i = F(x_i+1, y_i-0.5) = (x_i+1)^2 + (y_i-0.5)^2 - R^2$$

- if $p_i < 0$ then the midpoint is inside the circle, and we choose the pixel above the midpoint: $y_{i+1} = y_i$
- if $p_i > 0$ then the midpoint is outside the circle, and we choose the pixel below the midpoint:

$y_{i+1} = y_i-1$

Midpoint algorithm for circles

- Really neat trick: Calculate p_{i+1} using values already calculated at p_i:

  $p_{i+1} = F(x_{i+1}+1, y_{i+1}-0.5)$

  $= (x_{i+1}+1)^2 + (y_{i+1}-0.5)^2 - R^2$

  $p_{i+1} - p_i = 2(x_i+1)+1+(y_{i+1})^2 - (y_i)^2 - (y_{i+1}-y_i)$