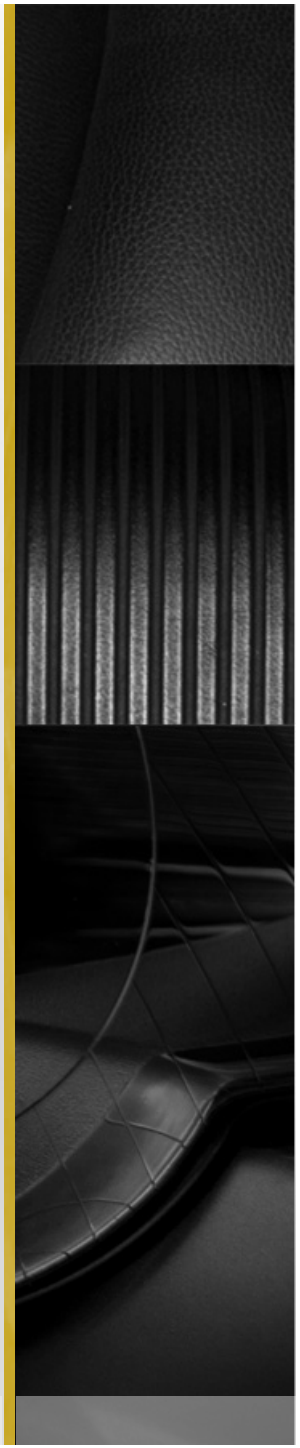
2D Primitives

Computer Graphics



Definitions

- **Pixel**
  - Smallest homogeneous color (intensity) unit of a digital image
  - Screen area with an assigned to a memory position (8 bits pixel has 256 color variations)
- **Display primitives**
  - Basic geometric structures (points, lines, curves, areas…) used to generate images
- **Conversion to the "raster"** (drawing figures into the raster)
  - Approximate primitives through a discrete set of points (**pixels map**)
    - Locate image pixel positions which are closest to an object (point, line..)
    - Store intensity/color values into the buffer (PIXMAP)
    - Trace the image on the screen (intensity values→ pixels)

Definitions

- **Image Buffer** → memory area where image definition is stored (set of intensity values for all screen points)
  - **B/W monitor** → 1 bit/pixel = 2 color intensities

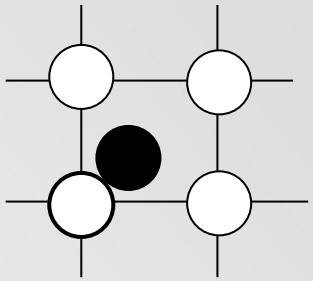
    BITMAP = set of bit values stored in the buffer
  - **Color monitor** → n bit/pixel = 2^n color intensities

    PIXMAP = set of pixel intensity values stored in the buffer
  - **High quality systems** → up to 24 bits/pixel

    24 bit RGB color system = real color system
- **Display processor/ Graphics controller** – frees CPU up from graphic work load

Drawing a Point

- Problem – convert a point (defined by coordinates) to the raster by storing color with which it will be stored

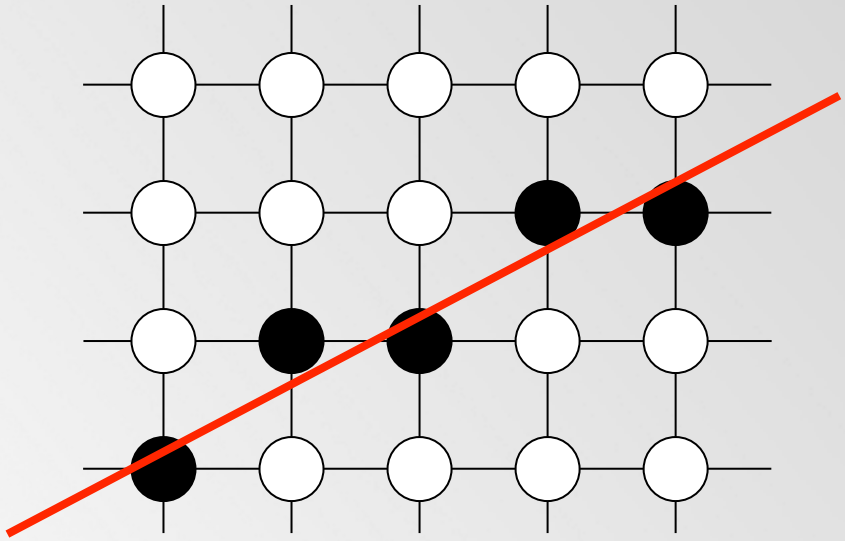


- Simple algorithm to convert point to the raster
  - **Let (x , y) be a real point**
  - **(x' , y') = (round(x) , round(y))** represents its position in the raster

    where **round()** rounds a real value to the closest integer

Drawing a Line

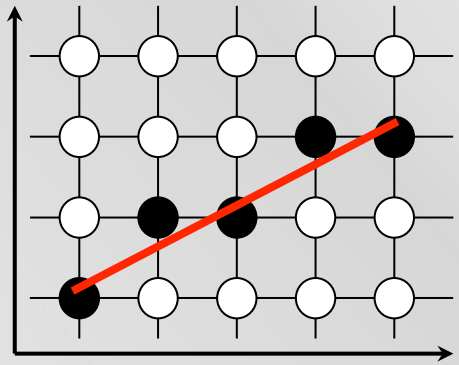
- Problem – calculate coordinates for pixels representing a straight line on a grid within a 2D raster



Drawing a Line

- **Brute-force algorithm**
  - Line equation between end points (x_1,y_1) and (x_2,y_2) → **$y = m\,x + b$**
    - Compute the slope of the line **$m = (y_2 - y_1) / (x_2 - x_1)$**
    - For each point x between **x_1 and x_2** , calculate the closest pixel to the line determined by the end points **$y = m\,x + b$**
  - Simple algorithm to display the line on the raster

```
for x from x1 to x2 (Δx unitary)
    y = x m + b
    draw pixel ( x, round(y) )
end for
```



Drawing a Line

- **Brute-force algorithm**
  - **Restriction**
    m needs to be between -1 and 1
  - **Solution**
    - Exchange x by y for all points in the line
    - Convert to the raster
    - Exchange again before visualizing
    - Calculate x as a function of unitary increments of y
  - **Disadvantages**
    Inefficient → each iteration requires:
    - Floating point calculation
    - Call the round() function

Drawing a Line

- **Digital differential analyzer (DDA) algorithm**
  - Algorithm that traces out successive (x, y) values by simultaneously incrementing x and y by small steps
  - Advantage – eliminates the floating point product
  - Disadvantage – still calls the round function
  - Select an axis with unitary steps (depending on $|m|$)
  - Represent the line from **left to right** → positive increments
  - Represent the line from **right to left** → negative increments

Drawing a Line

- **Digital differential analyzer (DDA) algorithm**
  - $\mathbf{y_{i+1}} = m \cdot x_{i+1} + b = m \cdot (x_i + \Delta x) + b = \mathbf{y_i + m\,\Delta x}$
  - $\mathbf{x_{i+1}} = (y_{i+1} - b) / m = (y_i + \Delta y - b) / m = \mathbf{x_i + \Delta y / m}$
  - $\mathbf{m = \Delta y / \Delta x} \rightarrow \mathbf{\Delta y = m\,\Delta x} \rightarrow \mathbf{\Delta x = \Delta y / m}$
  - $|m| < 1$
    - $\Delta x = 1 \quad \Delta y = m$
    - $\Delta x = -1 \quad \Delta y = -m$
  - $|m| > 1$
    - $\Delta y = 1 \quad \Delta x = 1/m$
    - $\Delta y = -1 \quad \Delta x = -1/m$

Drawing a Line

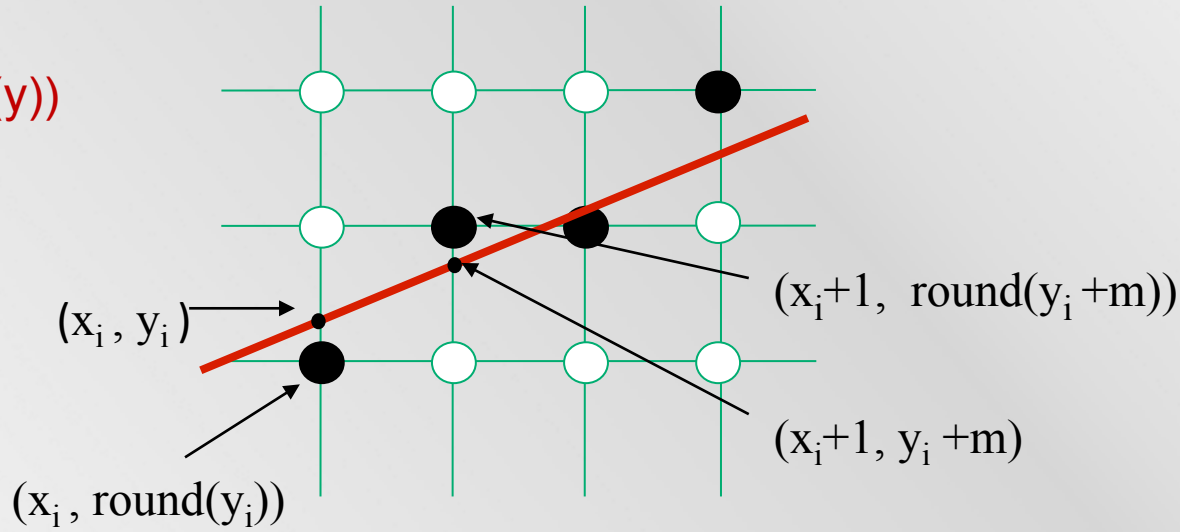
- **Traced line from left to right with slope m between 0 and 1**

For x from x_1 to x_2

    y = y + m

    draw pixel (x, round(y))

End for



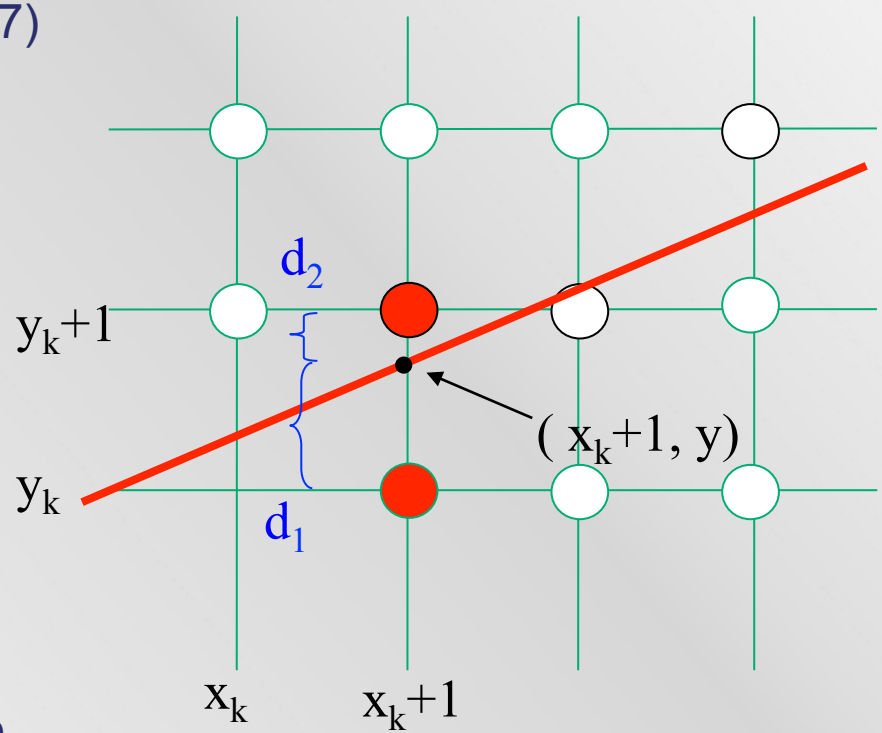
Drawing a Line

- **Complete DDA algorithm for the segment (x_1 , y_1) - (x_2 , y_2):**

```
dx = x2 – x1
dy = y2 – y1
If abs (dx) > abs ( dy ) then
      steps= abs ( dx )
else
      steps = abs ( dy )
End if
xInc = dx / steps
yInc = dy / steps
x = x1
y = y1
Draw pixel ( round(x) , round(y) )
For k from 1 to steps
      x = x + xInc
      y = y + yInc
      draw pixel ( round(x) , round(y) )
End for
```

Drawing a Line

- **Bresenham Midpoint algorithm** (1967)
  - After dislaying pixel **(x_k, y_k)**
    ¿ should we display pixel
    (x_k+1 , y_k) or **(x_k+1 , y_k +1)** ?
  - Coordinate y at **x_k+1** is:
    $y = m(x_k+1) + b$
  - Therefore, we define:
    $d_1 = y - y_k = m(x_k+1) + b - y_k$
    $d_2 = (y_k+1) - y = y_k + 1 - m(x_k+1) - b$
    $d_1 - d_2 = 2m(x_k+1) - 2y_k + 2b - 1$



Drawing a Line

- **Bresenham Midpoint algorithm**
  - We calculate the decision factor $\mathbf{p_k}$ using $m = \Delta y / \Delta x$ (obtaining an expression with integer operators only):

    $\mathbf{p_k} = \Delta x\,(d_1 - d_2)$

    $= \Delta x\,(\,2m\,(x_k+1) - 2y_k + 2b - 1\,)$

    $= 2\Delta y\,x_k + 2\Delta y - 2\Delta x\,y_k + \Delta x\,(\,2b - 1\,)$

    $= \mathbf{2\,(\Delta y\,x_k - \Delta x\,y_k\,) + \Delta x\,(\,2b - 1\,) + 2\Delta y}$

    Constant independent from the pixel $(\,\mathbf{x_k, y_k}\,)$

  - $\mathbf{p_k}$ **sign** determines the pixel $(\,\mathbf{x_k+1, y_k}\,)$
    = sign of $d_1 - d_2$ because $\Delta x > 0$ $(\,0 < m < 1\,)$
    - If $\mathbf{p_k < 0}$ → inferior pixel is drawn
    - If $\mathbf{p_k > 0}$ → superior pixel is drawn

Drawing a Line

- **Bresenham Midpoint algorithm** (continuation)
  - At step k+1 the decision factor would be

    $p_{k+1} = 2\Delta y\, x_{k+1} - 2\Delta x\, y_{k+1} + \text{constant}$
  - Subtracting consecutive decision factors

    $p_{k+1} - p_k = 2\Delta y\,(x_{k+1} - x_k) - 2\Delta x\,(y_{k+1} - y_k)$
  - Since $x_{k+1} = x_k + 1$ we have

    $\mathbf{p_{k+1} = p_k + 2\Delta y - 2\Delta x\,(y_{k+1} - y_k)}$

    $(y_{k+1} - y_k)$ is either 0 ($p_k < 0$) or 1 ($p_k > 0$)
  - p_0 evaluated at (x_0, y_0) with $\mathbf{m = \Delta y / \Delta x}$

    $\mathbf{p_0 = 2\Delta y - \Delta x}$

Drawing a Line

- **Bresenham Midpoint algorithm**

1. We capture line end points and store (x_0,y_0) into the buffer
   → first pixel (x_0,y_0) is drawn
2. Calculate constants Δx, Δy, $2\Delta x$, $2\Delta y$
3. Calculate $p_0 = 2\Delta y - \Delta x$
4. For each x_k (starting at k=0)

   If $\mathbf{p_k < 0}$ the point $\mathbf{(x_k+1 , y_k)}$ is drawn and $\mathbf{p_{k+1} = p_k + 2\Delta y}$

   If $\mathbf{p_k > 0}$ the point $\mathbf{(x_k+1 , y_k+1)}$ is drawn and $\mathbf{p_{k+1} = p_k + 2\Delta y - 2\Delta x}$
5. Repeat step 4 $\mathbf{\Delta x}$ **times**

Drawing a Line

- **Bresenham Midpoint algorithm**
  - Example – draw a line between (20,10) and (30,18)

    $\Delta x = 10$

    $\Delta y = 8$

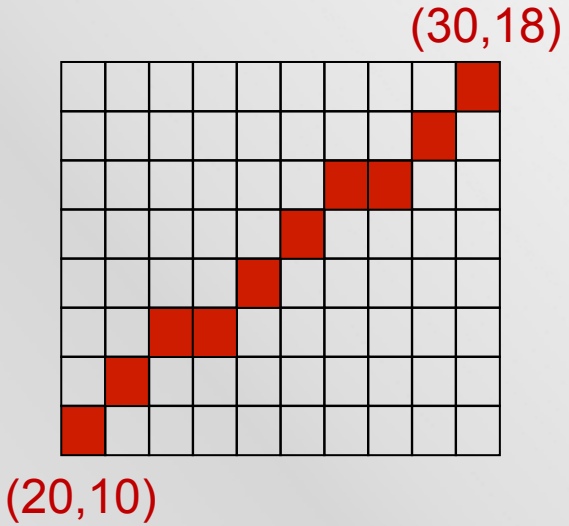
    $2\Delta y = 16$

    $2\Delta y - 2\Delta x = -4$

    $p_0 = 2\Delta y - \Delta x = 6$

    $(x_0 , y_0) = (20 , 10)$

k	p_k	(x_{k+1},y_{k+1})
0	6	(21,11)
1	2	(22,12)
2	-2	(23,12)
3	14	(24,13)
4	10	(25,14)
5	6	(26,15)
6	2	(27,16)
7	-2	(28,16)
8	14	(29,17)
9	10	(30,18)



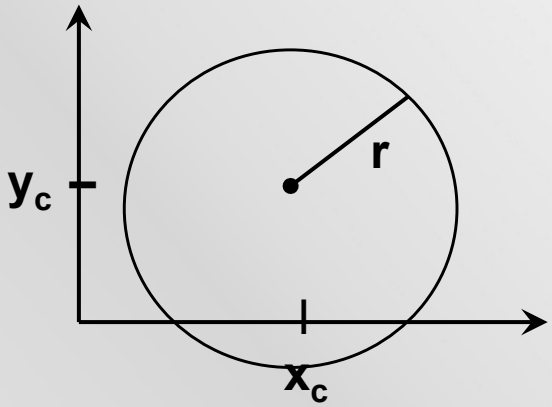
Drawing a Circumference

- We could use the circumference equation:

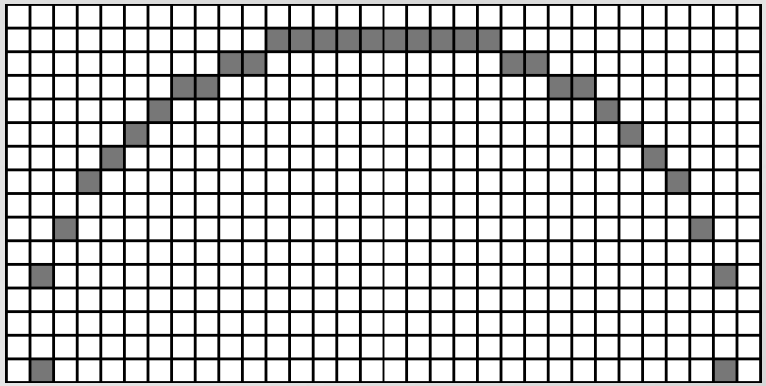
$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

to calculate the position of circumference points along the x axis with unitary steps from $x_c - r$ to $x_c + r$. Then calculate y values in each position

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$



- Drawback
  - Computational calculations
    - Floating point
    - Square root & rounding
  - Non-uniform distance between drawn pixels



Drawing a Circumference

- Polar coordinates to obtain uniform distance between pixels

$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$

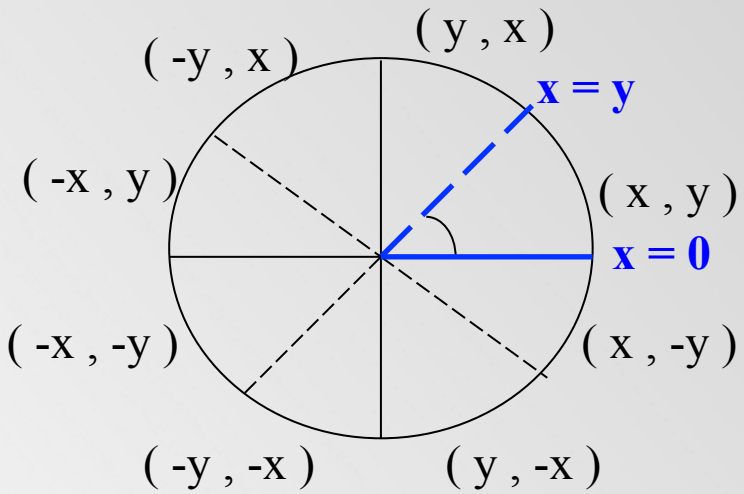
- Fix angular step size (θ) → equidistant points along the circunference (θ depends on the device)

- More continuous line → Step size = 1 / r
  → pixel positions separated by 1 unit ($\Delta\theta = 1$)

- Drawbacks → Trigonometric functions calculations

Drawing a Circumference

- Can be improved considering the circumference symmetry → requires to be drawn within the [0,π/4] range (symmetry between octants)



Drawing a Circumference

- **Midpoint algorithm**
  - Similar reasoning to that used for drawing lines is used to draw circumferences
  - Utilizes integer arithmetic
  - Utilizes circle symmetry
  - If (x_k , y_k) was the last point drawn, then the following point will be either (x_{k+1} , y_k) or (x_{k+1} , y_{k-1})
  - The decision parameter is a midpoint between those two

Drawing a Circumference

- Midpoint algorithm (algorithm complete discussion at Foley and Hearn)

1. First point $\mathbf{(x_0 , y_0) = (0 , r)}$
2. If **r** is:
   - Real → $\mathbf{p_0 = 5/4 - r}$
   - Integer → $\mathbf{p_0 = 1 - r}$
3. If $\mathbf{p_k}$
   - < 0 → select $\mathbf{(x_{k+1} , y_k)}$ and $\mathbf{p_{k+1} = p_k + 2x_{k+1} + 1}$
   - ≥ 0 → select $\mathbf{(x_{k+1} , y_{k-1})}$ and $\mathbf{p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}}$

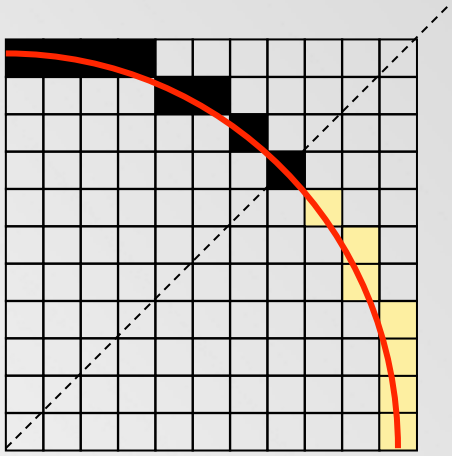
   With $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$
4. Calculate symmetric points
5. Move the points (calculated for an origin centered circumference) to a circumference centered at (x_c , y_c)
6. Repeat steps 3 to 5 until $x \geq y$

Drawing a Circumference

- Midpoint algorithm
  - Example – circumference with radius r = 10 centered on (0,0)

$$p_0 = 1 - r = -9$$

$$(x_0, y_0) = (0, 10)$$



k	p_k	(x_{k+1},y_{k+1})
0	-9	(1,10)
1	-6	(2,10)
2	-1	(3,10)
3	6	(4,9)
4	-3	(5,9)
5	8	(6,8)
6	5	(7,7)

Drawing other geometric figures

- **The midpoint algorithm** is generalized to:
  - Ellipses
  - Conics
  - **Splines** – curves generated by linking segments of cubic polynomials

$$x = a_{x0} + a_{x1}u + a_{x2}u^2 + a_{x3}u^3$$

$$y = a_{y0} + a_{y1}u + a_{y2}u^2 + a_{y3}u^3$$

    - Constants are obtained forcing continuity and differentiability conditions at nodal points