Chapter 4:
Primitive Objects Using Graphics

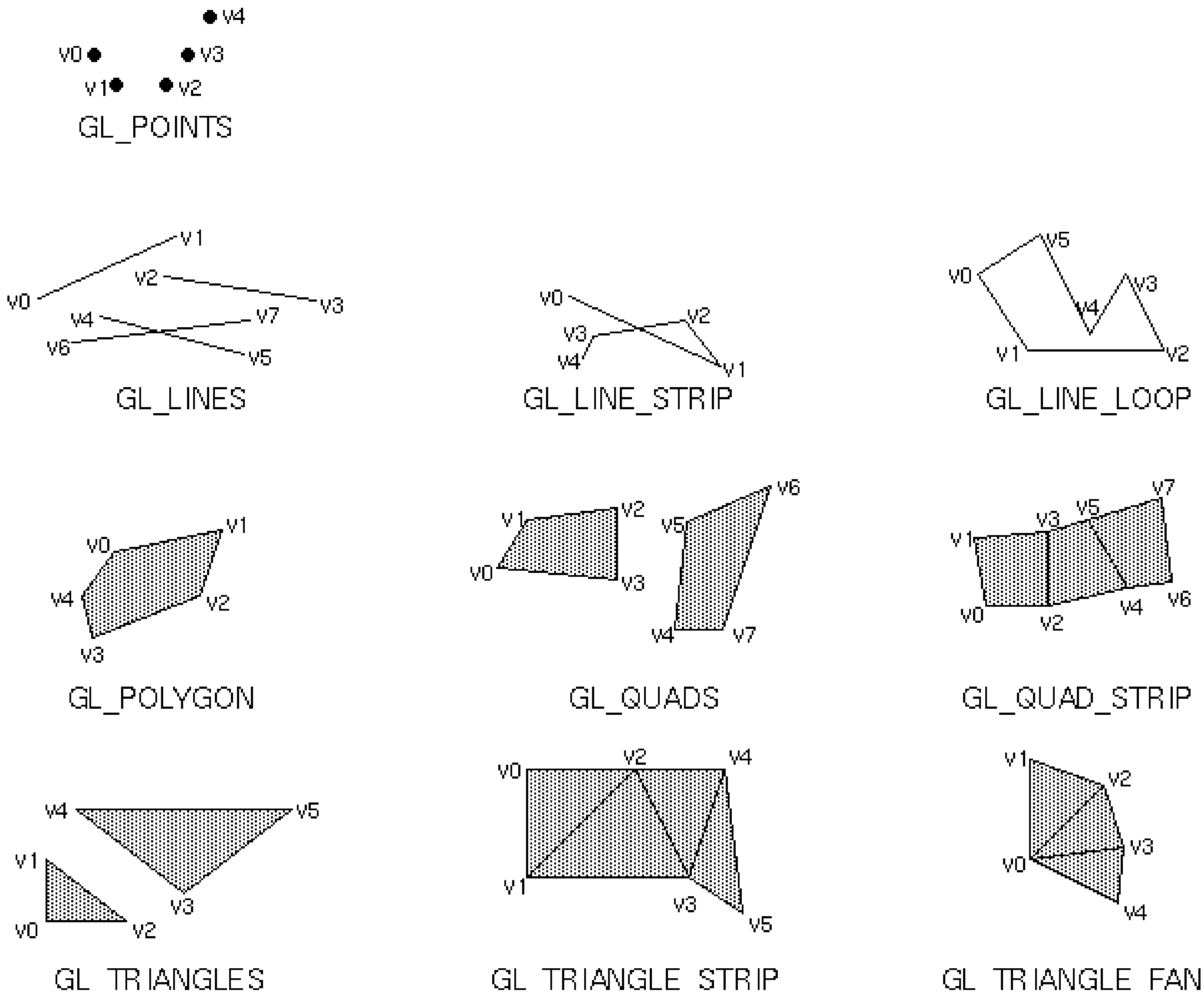
Objective

- To create primitive objects using graphics.
- To produce output primitives:
  - Point
  - Line
  - Circle
  - Rectangle
  - Polygon
- To introduce the graphics system using OpenGL.

Output Primitives

- Graphic SW and HW provide subroutines to describe a scene in terms of basic geometric structures called output primitives.
- Output primitives are combined to form complex structures
- Simplest primitives
  - Point (pixel)
  - Line segment

Geometric Primitive Types



Scan Conversion

- Converting output primitives into frame buffer updates. Choose which pixels contain which intensity value.
- Constraints
  - Straight lines should appear as a straight line
  - primitives should start and end accurately
  - Primitives should have a consistent brightness along their length
  - They should be drawn rapidly

Point

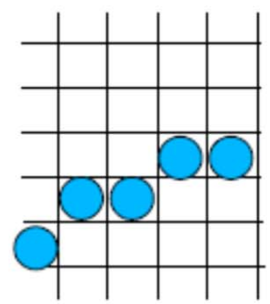
- Coordinate position is represented by 1 pixel or more.
- Screen coordinate is integer: point p(x, y) = pixel {int(x), int(y)}.
- Example $P_1(2.5, 3.25) \rightarrow P_{pixel} = (2, 3)$
- Instruction example:
  - Draw one point → *setPixel (x,y)*
  - Take point position → *getPixel (x,y)*

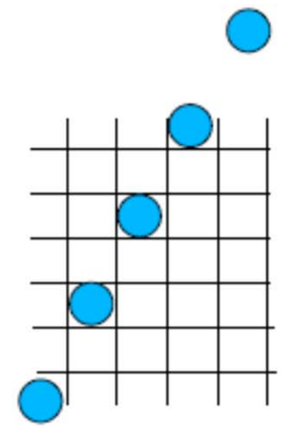
Line Drawing

- Simple approach:
  sample a line at discrete positions at one coordinate from start point to end point, calculate the other coordinate value from line equation.

$$y = mx + b \qquad x = \frac{1}{m}y + \frac{b}{m}$$

$$m = \frac{y_{start} - y_{end}}{x_{start} - x_{end}}$$





If $m > 1$, increment y and find x

If $m \leq 1$, increment x and find y

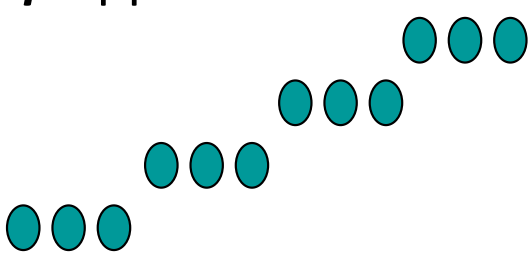
Algorithms

- **Line drawing**
  - Digital Differential Analyzer (DDA) algorithm
  - Bresenham's line algorithm

- **Circle-generating algorithm**
  - Midpoint circle algorithm

Approaches

- To display a line on a raster monitor, the graphics system must first
  - Project the endpoints to **integer** screen coordinates.
  - Determine the **nearest pixel positions** along the line path between the two end point.
  - Then the line color is loaded into the **frame buffer** at the corresponding pixel coordinates.
  - Reading from the frame buffer , the **video controller** plots the screen pixels.
- This process digitizes the line into a set of discrete integer positions that, in general, only **approximates the actual line path**.
  - A computed line position of *(10.48,20.51)* for example, is converted into *(10,21)*.
  - This rounding of coordinate values causes all (except horizontal and vertical) lines to be displayed with ***stair-step*** appearance.



Stair-step effect produced when a line is generated as a series of pixel positions.

Line Equations

- The Cartesian *slope-intercept equation* for a straight line is

$$y = m \cdot x + b \qquad (1)$$

with *m* as the slope and *b* as the intercept of the line.

- If *2* endpoints *(x0,y0)* and *(x1,y1)* is given, the values of slope *m* and *y* intercept *b* can be computed as:

$$m = \frac{y1 - y0}{x1 - x0} \qquad (2)$$

$$b = y0 - m \cdot x0 \qquad (3)$$

- Algorithms for displaying straight lines are based on the line equation (1).

- For any given *x* interval, δx, along a line, we can compute the corresponding *y* interval δy as:

$$\delta y = m \cdot \delta x \qquad (4)$$

- Similarly, the *x* interval δx corresponding to a specified δy as:

$$\delta x = \frac{\delta y}{m} \qquad (5)$$

- These equations form the basis for determining **deflection voltages** in analog displays, such as *vector-scan* system, where arbitrarily small changes in deflection voltage are possible.

Digital Differential Analyzer (DDA) Algorithm

- **DDA** is a scan-conversion line algorithm based on calculating either δy or δx using (4) and (5).
- Steps in the **DDA** algorithm are:
  1. Determine 2 endpoints to draw a line.
  2. Set (x0,y0) as the first point and (x1,y1) as the last point.
  3. Compute the difference of $\Delta x = x1 - x0$ and $\Delta y = y1 - y0$.
  4. The difference with greater magnitude determines the value of parameter *steps*. i.e.

     if $|\Delta x| > |\Delta y|$ then

     $steps = |\Delta x|$

     else $steps = |\Delta y|$
  5. Starting with pixel position (x0,y0), we determine the offset needed at each step to generate the next pixel position along the line path.

     i.e. x_increment = $|\Delta x|/steps$ and y_increment= $|\Delta y|/steps$.
  6. SetPixel at location (round(x0), round(y0)).
  7. Set k=0.
  8. Determine the next coordinate (x_{k+1}, y_{k+1}). $x_{k+1} = x_k + x\_increment$ and $y_{k+1} = y_k + y\_increment$. And setpixel at (round(x_{k+1}), round(y_{k+1})).
  9. Repeat step 8. until $x=x1$ and $y=y1$.

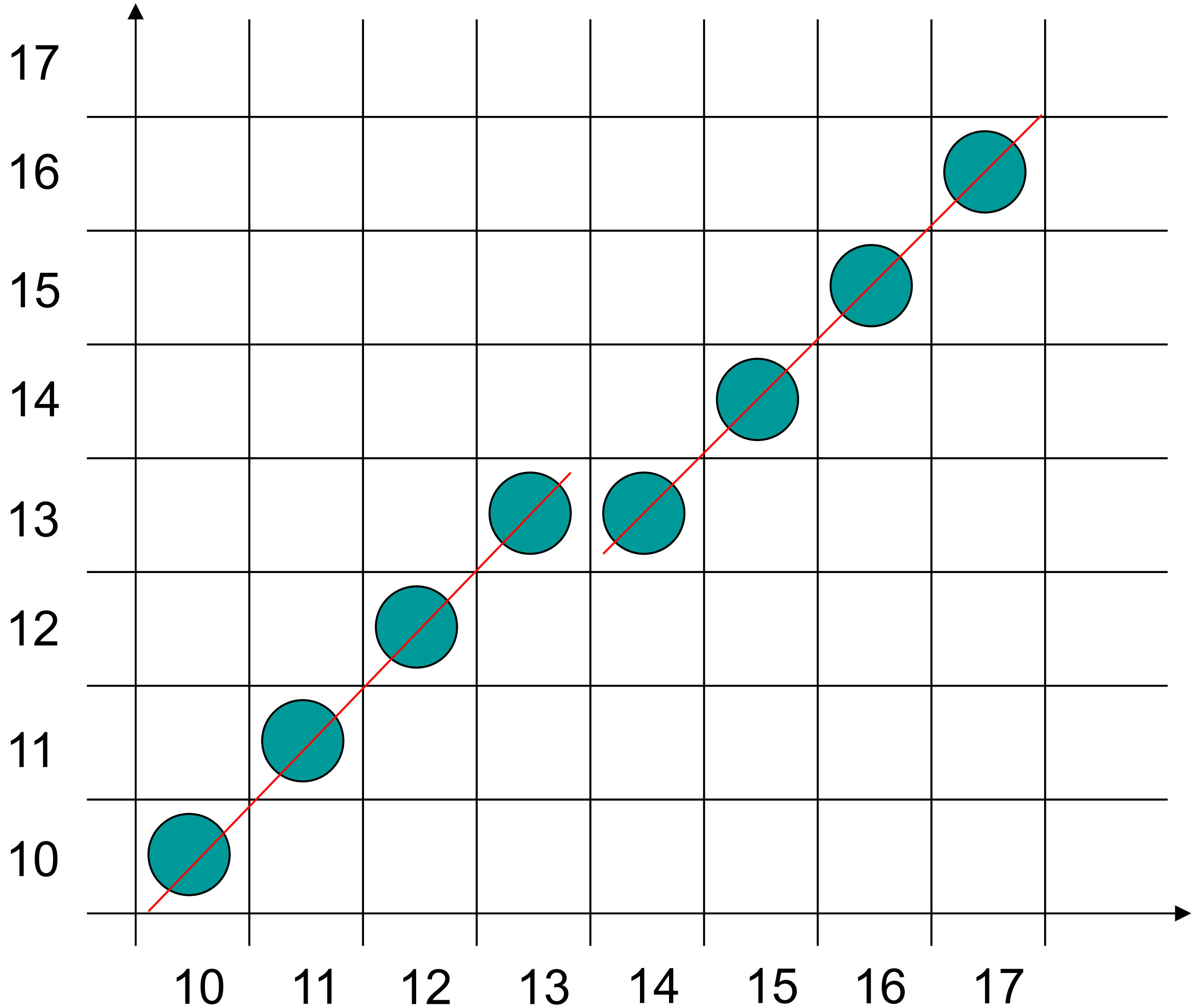
DDA Algorithm: Example

- Two endpoints of a line are given as A(10,10) and B(17,16). Compute the points generated by the **DDA** algorithm.
  1. A(10,10) and B(17,16)
  2. (x_0,y_0) =(10,10) and (x_1,y_1)=(17,16)
  3. $\Delta x=7$ and $\Delta y=6$.
  4. $\Delta x > \Delta y$, thus $step=7$.
  5. x_increment=$\Delta x \div step$=1 and y_increment=$\Delta y \div step = 0.86$.
  6. $(x_{k+1},y_{k+1})=(x_k + x\_increment,\ y_k + y\_increment) = (10+1,\ 10+0.86) = (11,10.86)$
  7. (round(x_{k+1}), round(y_{k+1})) = (11,11)
  8. Repeat steps 6. and 7. until $(x_{k+1},y_{k+1}) = (17,16)$

when $k=0$

k	x	y	(round(x), round(y))
0	10	10	(10,10)
1	11	10.86	(11,11)
2	12	11.72	(12,12)
3	13	12.58	(13,13)
4	14	13.44	(14,13)
5	15	14.30	(15,14)
6	16	15.16	(16,15)
7	17	16.02	(17,16)

Line Produced by DDA Algorithm



DDA Code in C

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a)  { return int (a + 0.5); }

void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
  int dx = xEnd - x0,  dy = yEnd - y0,  steps,  k;
  float xIncrement, yIncrement, x = x0, y = y0;

  if (fabs (dx) > fabs (dy))
    steps = fabs (dx);
  else
    steps = fabs (dy);
  xIncrement = float (dx) / float (steps);
  yIncrement = float (dy) / float (steps);

  setPixel (round (x), round (y));
  for (k = 0; k < steps; k++) {
    x += xIncrement;
    y += yIncrement;
    setPixel (round (x), round (y));
  }
}
```

Pros and Cons of DDA Algorithm

- **DDA** is a faster algorithm for calculating pixel positions compared to the one that directly implements Eq. (1).
- It eliminates the multiplication in Eq.(1) by appropriate increments in the *x* or *y* directions to step from one pixel to another along the line path.
- The accumulation of **round-off error** can drift away from the true line path for long line segments.
- The rounding operations and floating-point arithmetic is still **time consuming**.
- Its performance can be improved by separating the increments m and $\frac{1}{m}$ into integer and fractional parts. Thus, all calculations will be reduced to integer operations.

Bresenham's Line Algorithm

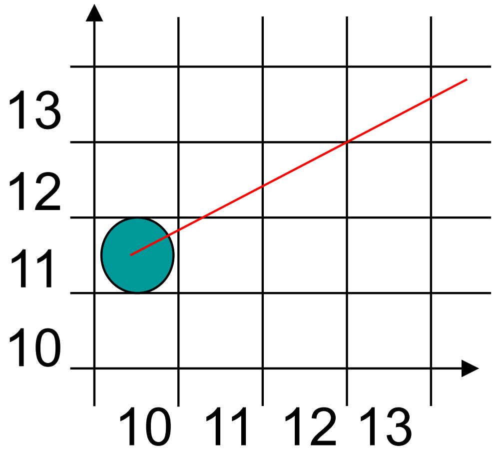


Figure 1 (positive slope)

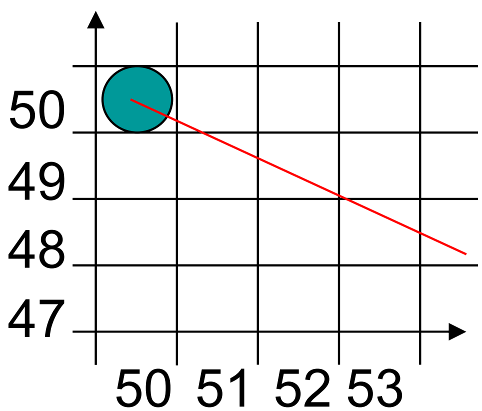


Figure 2 (negative slope)

- Is a more general scan-line approach that can be applied to both **line** and **curves**.
- This raster-line generating algorithm is developed by Bresenham, that uses only **incremental integer calculations**.
- Figure 1 and Figure 2 shows section of display screen where a straight line is to be drawn.
  - The vertical axis shows the scan-line positions.
  - The horizontal axis identify pixel columns.
- On figure 1, starting from the left endpoint we need to determine whether to plot the next pixel at position (11,11) or the one at (11,12).
- Similarly for Figure 2,starting from left endpoint , we want to determine whether to plot the next pixel at position (51,50) or (51,49).
- These questions are answered in Bresenham's algorithm.

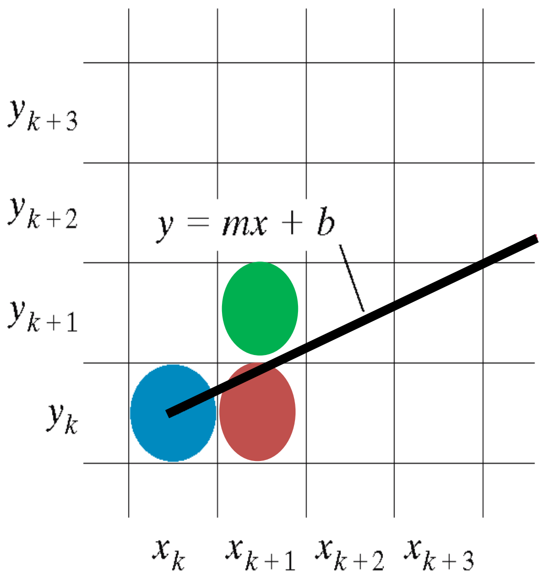


Figure 3-10

A section of the screen showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0 < m < 1$.

- We first look at the scan-conversion process for lines with **positive slope less than 1.0**.
- Pixel positions are determined by sampling at **unit** *x* intervals.
- Starting from left endpoint (x_0, y_0) we move to each successive column and plot the pixel whose scan-line *y* value is **closest** to the line path.
- Figure 3-10 shows the first pixel at point (x_k, y_k), we must determine the position of next pixel
  - i.e. either (x_{k+1}, y_k) or (x_{k+1}, y_{k+1}).

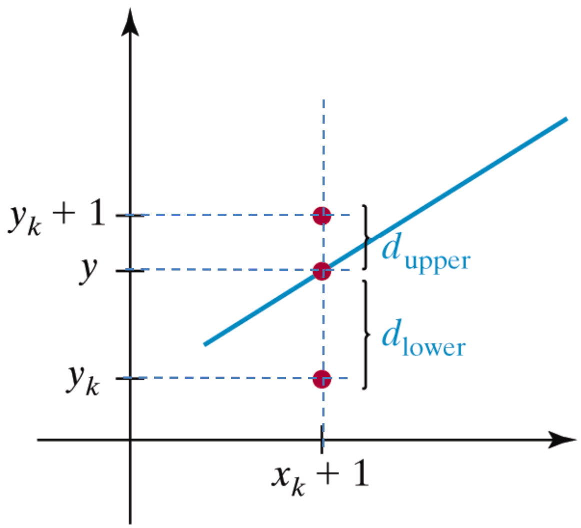


Figure 3-11

Vertical distances between pixel positions and the line y coordinate at sampling position $x_k + 1$.

Here, $x_{k+1} = x_k+1$
$y_{k+1} = y_k+1$

- The y coordinate on the mathematical line at pixel column position x_k+1 is calculated as,

$$y=m(x_k+1)+b \tag{6}$$

- Then,

$$\begin{aligned} d_{lower} &= y-y_k \\ &= m(x_k+1)+b-y_k \end{aligned} \tag{7}$$

and

$$\begin{aligned} d_{upper} &= (y_k+1)-y \\ &= y_k+1-m(x_k+1)-b \end{aligned} \tag{8}$$

Computer Graphics with Open GL, Third Edition, by Donald Hearn and M.Pauline Baker.

- To find which of the two pixels is closest to the point

  $d_{lower}-d_{upper}=2m(x_k+1)-2y_k+2b-1$

  where ,

$$m = \frac{\Delta y}{\Delta x} \qquad (9)$$

- Simplifying (9) and defining **decision parameter, p_k,** we get,

  $p_k=\Delta x(d_{lower} - d_{upper})$

  $=2\Delta y \cdot x_k -2\Delta x \cdot y_k +c$

where,

$$c=2\Delta y+\Delta x(2b-1) \qquad (10)$$

- If $p_k >=0$

  choose: (x_k+1 ,y_k+1)

  else

  choose: (x_k+1,y_k)

- At step ***k+1***, the decision parameter is evaluated as,

$$p_{k+1} = 2\Delta y \cdot x_{k+1}-2\Delta x \cdot y_{k+1}+ c \qquad (11)$$

- So,

$$p_{k+1} - p_k = 2\Delta y(x_{k+1}-x_k)-2\Delta x(y_{k+1}-y_k) \qquad (12)$$

- Since, $x_{k+1}=x_k+1$

$$p_{k+1} = p_k +2\Delta y-2\Delta x(y_{k+1}-y_k) \qquad (13)$$

- Thus,

  $p_{k+1} = p_k +2\Delta y$ $\quad$ *if* $p_k < 0$

  $p_{k+1} = p_k +2\Delta y-2\Delta x$ $\quad$ *if* $p_k >= 0$

- The first parameter p_0 is evaluated from (10) at (x_0, y_0)

  From (6) ,

  $$y_0 = mx_0 + b$$

  so,

  $$b = y_0 - mx_0 \qquad (14)$$

  $$p_k = \Delta x(d_{lower} - d_{upper})$$

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

Substitute b in (14)

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2(y_0 - mx_0) - 1)$$

Simplify…

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

Bresenham's line drawing algorithm for $|m|<1.0$

1. Input 2 endpoints of the line and store the left endpoint in (x_0, y_0).
2. Set the color for the frame-buffer position (x_0, y_0); i.e. plot the first point.
3. Calculate the constants Δx, Δy, $2\Delta y$, *and* $2\Delta y - 2\Delta x$, and obtain the starting value for the decision parameter as

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

4. At each x_k, along the line, starting at $k=0$, perform the following test.

   if $p_k<0$, the next point to plot is (x_{k+1}, y_k) and

   $$p_{k+1} = p_k + 2\Delta y$$

   if $p_k \geq 0$

   the next point to plot is (x_{k+1}, y_{k+1}) and

   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Perform step 4. $\Delta x - 1$ *times.*

Bresenham's Algorithm :Example

- Two endpoints of a line is given as (20,10) and (30,18). This line has slope of 0.8, with

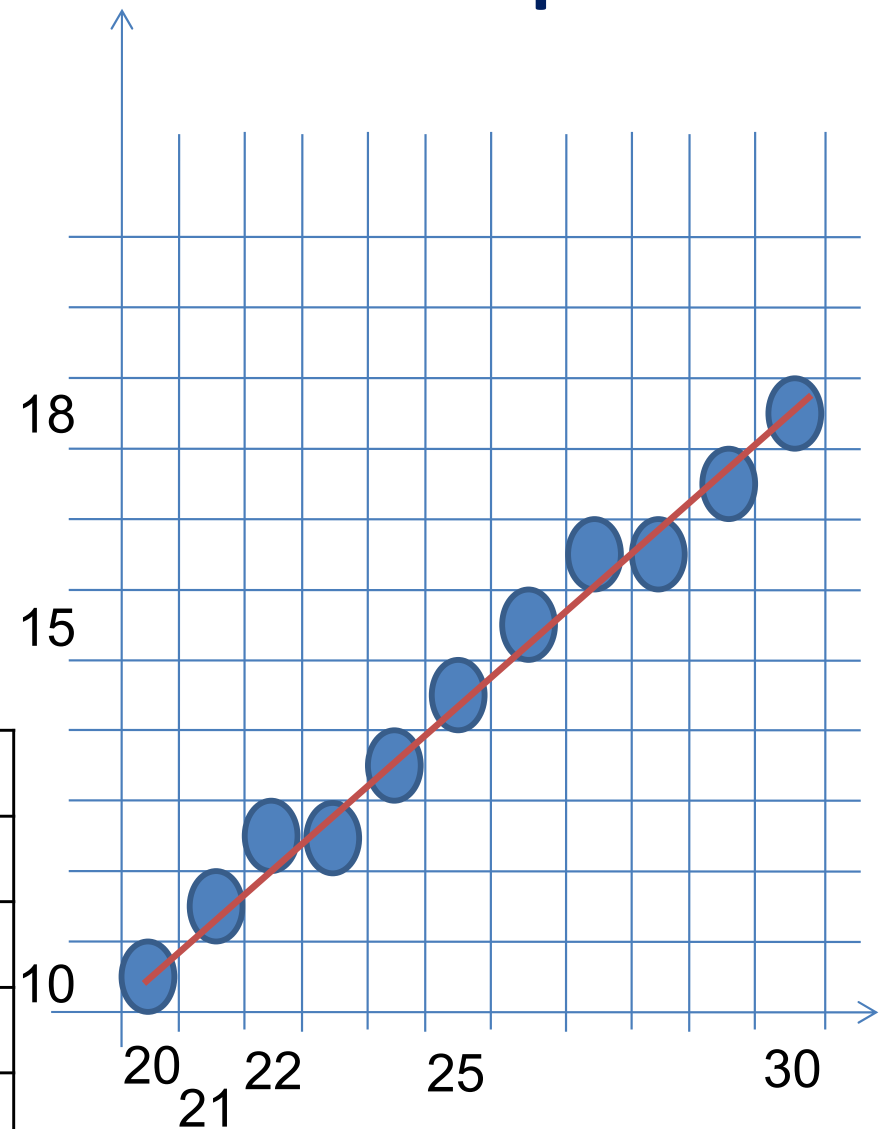
$\Delta x=10,\ \Delta y=8$

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

$2\Delta y=16,$

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

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



Bresenham’s Code in C

```
#include <stdlib.h>
#include <math.h>

/*  Bresenham line-drawing procedure for |m| < 1.0.  */
void lineBres (int x0, int y0, int xEnd, int yEnd)
{
  int dx = fabs (xEnd - x0),  dy = fabs(yEnd - y0);
  int p = 2 * dy - dx;
  int twoDy = 2 * dy,  twoDyMinusDx = 2 * (dy - dx);
  int x, y;

  /* Determine which endpoint to use as start position.  */
  if (x0 > xEnd) {
    x = xEnd;
    y = yEnd;
    xEnd = x0;
  }
  else {
    x = x0;
    y = y0;
  }
  setPixel (x, y);

  while (x < xEnd) {
    x++;
    if (p < 0)
      p += twoDy;
    else {
      y++;
      p += twoDyMinusDx;
    }
    setPixel (x, y);
  }
}
```

Bresenham's Algorithm for $|m| \not< 1.0$

- i.e m=0 (when Δy=0), m is undefined (when Δx=0), m=1 (when $|\Delta y|=|\Delta x|$), m<0 and m>1.
- For lines with m>1, interchange the roles of *x* and *y* directions. i.e. step along *y* direction in unit steps and calculate successive *x* values nearest the line path.
- For m<0 (negative slope), the procedure is similar, except that now one coordinate decreases and as the other increases.
- Special cases: horizontal line (Δy=0), vertical lines (Δx=0), and diagonal lines ($|\Delta y|=|\Delta x|$) can each be loaded directly into the frame buffer without processing them through the line-plotting algorithm.

Curve generating Algorithm

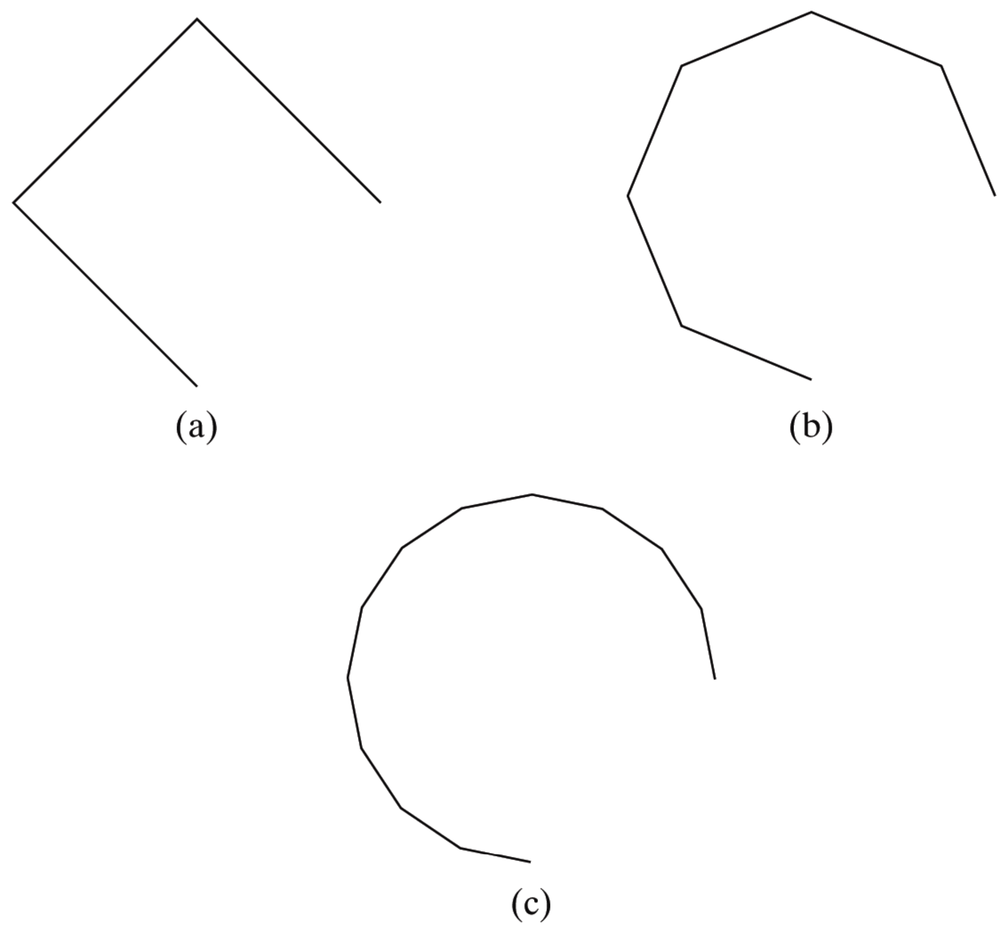


Figure 3-15

A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

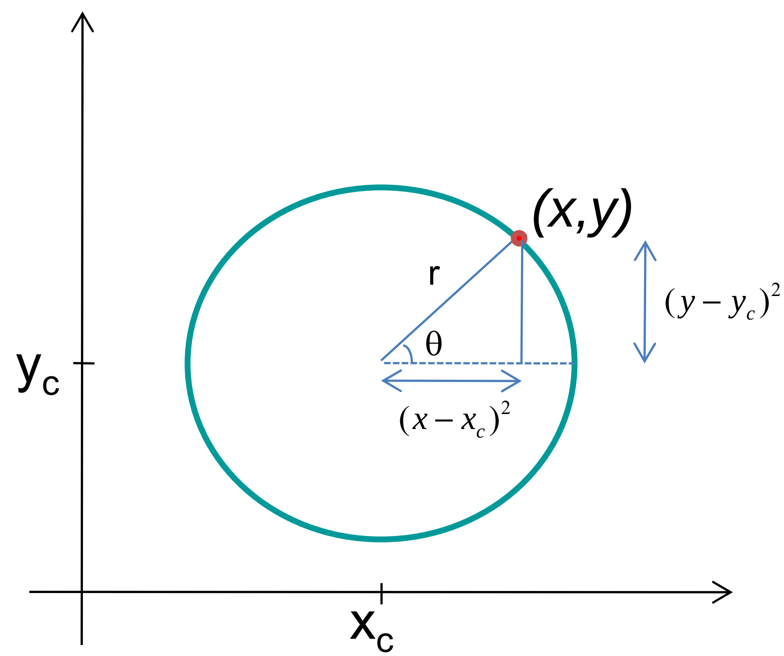
Computer Graphics with Open GL, Third Edition, by Donald Hearn and M.Pauline Baker.

Circle generating algorithms

- For any circle point *(x,y)*, with radius, *r*, from a center point *(x_c,y_c)*, the distance relationship is expressed by the **Pythagorean theorem** in Cartesian coordinates as:

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

- But using this equation to generate a circle is time consuming.
- We consider the **midpoint circle** algorithm here.
- There are many other techniques to reduce the calculation time. Read page 103-104 (Hearn & Baker)



Midpoint Circle Algorithm

- Uses the same technique used in the raster line algorithm.
- For a given radius, *r*, and screen center position (x_c, y_c).
  - Set up the algorithm to compute pixel positions around the circle path centered at coordinate *(0,0)*.
  - Then each computed position *(x,y)* is moved to proper screen position by adding x_c to *x* and y_c to *y*.
  - Symmetry consideration can be used to reduce computations.
- Here we look at the first quadrant, Figure 3.18 that is from *x=0* to *x=y*, the slope of the curve varies from *0* to *-1.0*.
- To apply the midpoint method, a circle function is defined as

$$f_{circ}(x,y)=x^2+y^2-r^2$$

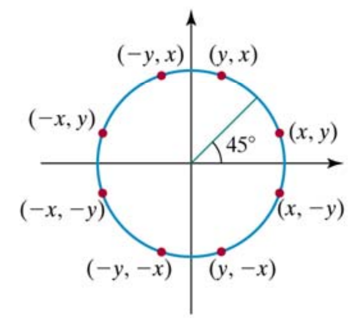


Figure 3-18

Symmetry of a circle. Calculation of a circle point (x, y)in one octant yields the circle points shown for the other seven octants.

The function below gives the value of *fcirc(x,y)* for different *(x,y)* positions.

$$f_{circ}(x,y)=\begin{cases} < 0, & \text{if (x,y) is inside the circle boundary.} \\ =0, & \text{if (x,y) is on the circle boundary.} \\ > 0, & \text{if (x,y) is outside the circle boundary.} \end{cases} \quad \textbf{(15)}$$

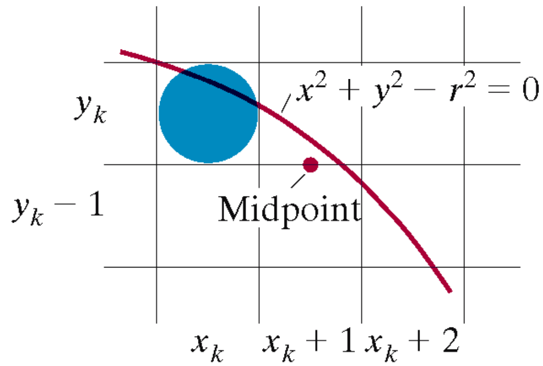


Figure 3-19

Midpoint between candidate pixels at sampling position $x_k + 1$ along a circular path.

- We want to determine whether to choose (x_{k+1}, y_k) or (x_{k+1}, y_{k-1}).

- The **decision parameter** is the circle function (15) evaluated at the **midpoint**.

$$p_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$

$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2} \right)^2 - r^2$$

- If $p_k < 0$,

  choose (x_{k+1}, y_k)

  if $p_k \geq 0$

  choose (x_{k+1}, y_{k-1}).

- Next to find pixel position at $x_{k+1}+1=x_k+2$

$$p_{k+1} = f_{circ}(x_{k+1}+1, y_{k+1}-\tfrac{1}{2})$$

$$= (x_{k+1}+1)^2 + \left(y_{k+1}-\frac{1}{2}\right)^2 - r^2$$

Can be simplified to:

$$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

where y_{k+1} is taken as either y_k or y_k*-1*, depending on the sign of $p_{k.}$

- Increments for obtaining p_{k+1} are either $2x_{k+1}+1$ (if p_k is negative) or $2x_{k+1}+1-2y_{k+1}$ (if p_k is non-negative)

- With $2x_{k+1}=2x_k+2$ and $2y_{k+1}=2y_k-2$

- The initial decision parameter is obtained by evaluating the circle function at $(x_0,y_0)=(0,r)$:

$$p_0 = f_{circ}(1, r-\tfrac{1}{2})$$

$$= 1+(r-\tfrac{1}{2})^2 - r^2$$

$$p_0 = \tfrac{5}{4} - r$$

- If radius is specified as an integer, we can simply round p_0 to :

  $p_0=1-r$ (for r integer)

Midpoint Circle algorithm

1. Input radius r and circle center (x_c,y_c) then set the coordinates for the first point on the circumferences of a circle centered on the origin as

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

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, starting at $k=0$, perform the following test.

   If $p_k < 0$, the next point along the circle centered on $(0,0)$ is (x_{k+1},y_k) and

$$p_{k+1}=p_k+2x_{k+1}+1$$

   Otherwise, the next point along the circle is (x_{k+1},y_{k-1}) and

$$p_{k+1}=p_k+2x_{k+1}+1-2y_{k+1}$$

   where $2x_{k+1}=2x_k+2$ and $2y_{k+1}=2y_k-2$

4. Determine symmetry points in the other 7 octants.

5. Move each calculated pixel position ***(x,y)*** onto the circular path centered at (x_c,y_c) and plot the coordinate values

$$x=x+x_c\,,\quad y=y+y_c$$

6. Repeat steps 3 through 5 until $x \geq y$

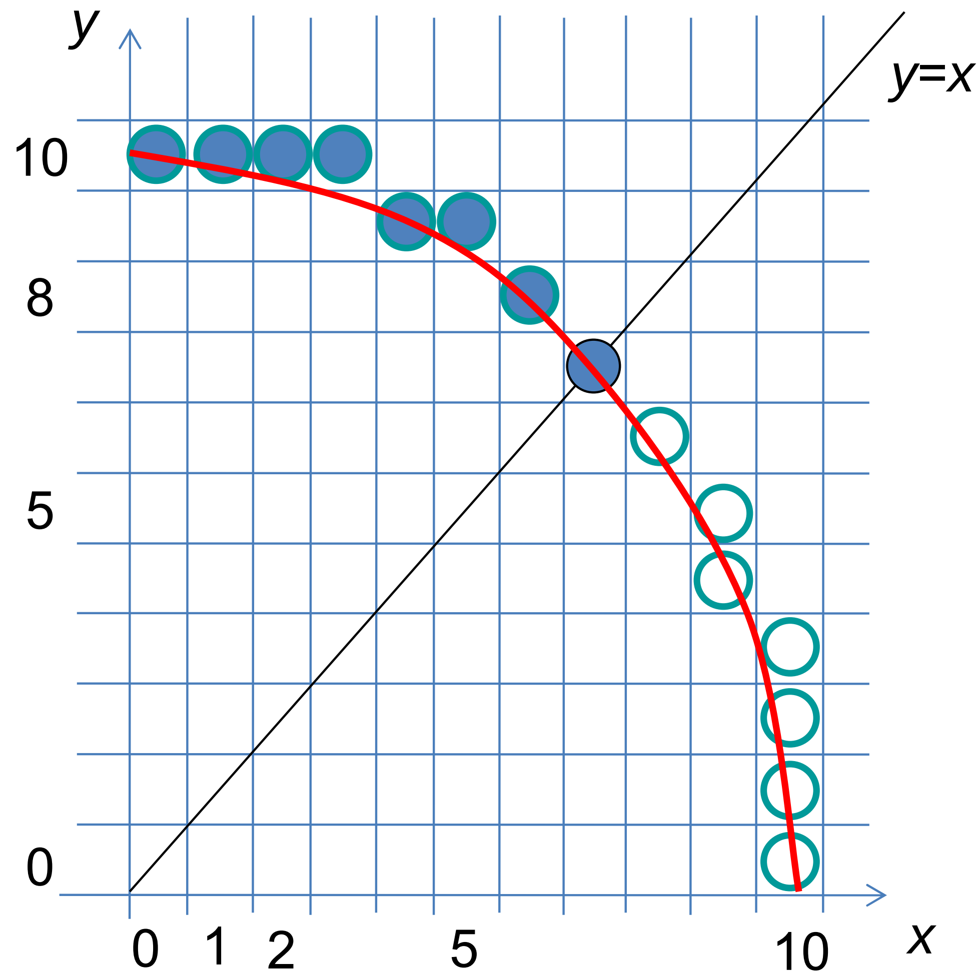
Mid point circle drawing : Example

- Given a circle radius r=10 centered at (0,0). Using the midpoint circle algorithm determine positions along the circle in the first quadrant.
  - Using the symmetrical property, we find the positions along the circle octant in the first quadrant from $x=0$ to $x=y$.
  - The initial value of the decision parameter is

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

  - For the circle centered on the coordinate origin, the initial point is $(x_0, y_0)=(0,10)$,
  - The initial increment terms for calculating decision parameters are: $2x_0=0$, $2y_0=20$.

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1,10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14



```
#include <GL/glut.h>

  class scrPt {
    public:
      GLint x, y;
  };
  void setPixel (GLint x, GLint y)
  {
    glBegin (GL_POINTS);
      glVertex2i (x, y);
    glEnd ( );
  }
void circleMidpoint (scrPt circCtr, GLint radius)
  {
    scrPt circPt;

    GLint p = 1 - radius;        // Initial value of midpoint parameter.

    circPt.x = 0;            // Set coordinates for top point of circle.
    circPt.y = radius;

    void circlePlotPoints (scrPt, scrPt);
    /* Plot the initial point in each circle quadrant. */
    circlePlotPoints (circCtr, circPt);
    /* Calculate next points and plot in each octant. */
    while (circPt.x < circPt.y) {
      circPt.x++;
      if (p < 0)
        p += 2 * circPt.x + 1;
      else {
        circPt.y--;
        p += 2 * (circPt.x - circPt.y) + 1;
      }
      circlePlotPoints (circCtr, circPt);
    }
  }

void circlePlotPoints (scrPt circCtr, scrPt circPt);
  {
    setPixel (circCtr.x + circPt.x, circCtr.y + circPt.y);
    setPixel (circCtr.x - circPt.x, circCtr.y + circPt.y);
    setPixel (circCtr.x + circPt.x, circCtr.y - circPt.y);
    setPixel (circCtr.x - circPt.x, circCtr.y - circPt.y);
    setPixel (circCtr.x + circPt.y, circCtr.y + circPt.x);
    setPixel (circCtr.x - circPt.y, circCtr.y + circPt.x);
    setPixel (circCtr.x + circPt.y, circCtr.y - circPt.x);
    setPixel (circCtr.x - circPt.y, circCtr.y - circPt.x);
  }
```