Higher Technological Institute
Computer Science Department

Computer Graphics

Dr Osama Farouk
Dr Ayman Soliman
Dr Adel Khaled

Lecture two
Graphics Output Primitives

OpenGL Point Functions

- The default color for primitives is white and the default point size is equal to the size of one screen pixel.
- The form for an OpenGL specification of a point position is

```
glBegin (GL_POINTS);
 glVertex* ( );          //The coordinate values for a single position
glEnd ( );
```

- In the following example, three equally spaced points are plotted along a two-dimensional straight-line path with a slope of 2 (Figure). Coordinates are given as integer pairs.

```
glBegin (GL_POINTS);
    glVertex2i (50, 100);
    glVertex2i (75, 150);
    glVertex2i (100, 200);
glEnd ( );
```

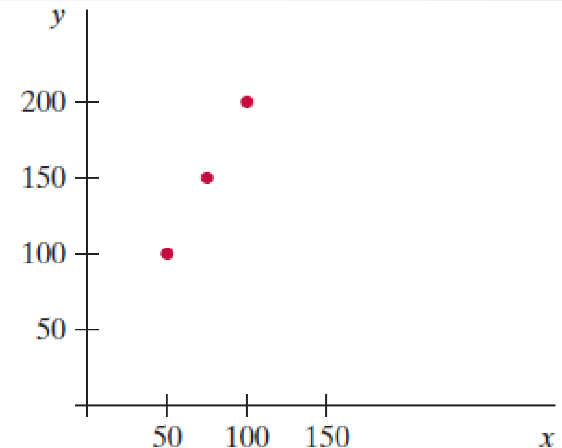


FIGURE Display of three point positions generated with glBegin (GL_POINTS).

Alternatively, we could specify the coordinate values for the preceding points in arrays such as

```
int point1 [ ] = {50, 100};
int point2 [ ] = {75, 150};
int point3 [ ] = {100, 200};
```

and call the OpenGL functions for plotting the three points as

```
glBegin (GL_POINTS);
glVertex2iv (point1);
glVertex2iv (point2);
glVertex2iv (point3);
glEnd ( );
```

And here is an example of specifying two point positions in a three dimensional world reference frame. In this case, we give the coordinates as explicit floating-point values.

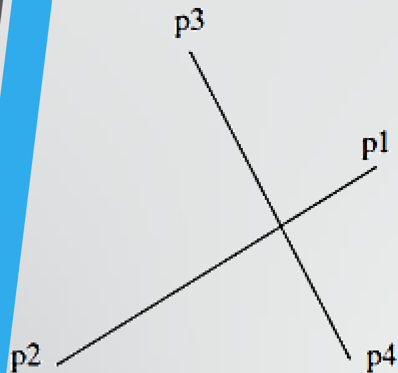
```
glBegin (GL_POINTS);
glVertex3f (-78.05, 909.72, 14.60);
glVertex3f (261.91, -5200.67, 188.33);
glEnd ( );
```

Using this class definition, we could specify two- dimensional, world-coordinate point position with the statements

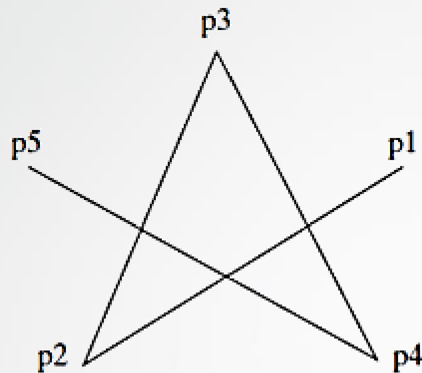
```
wcPt2D pointPos;
pointPos.x = 120.75;
pointPos.y = 45.30;
glBegin (GL_POINTS);
glVertex2f (pointPos.x, pointPos.y);
glEnd ( );
```

Look to pages in textbook "Computer Graphics with open GL" Pages(88-89)

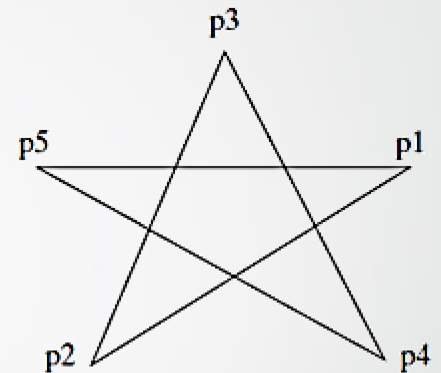
OpenGL LINE FUNCTIONS



(a)



(b)



(c)

The following code could generate the display shown in Figure.

```
glBegin (GL_LINES);
      glVertex2iv (p1);
      glVertex2iv (p2);
      glVertex2iv (p3);
      glVertex2iv (p4);
      glVertex2iv (p5);
glEnd ( );
```

```
glBegin (GL_LINES_STRIP);
      glVertex2iv (p1);
      glVertex2iv (p2);
      glVertex2iv (p3);
      glVertex2iv (p4);
      glVertex2iv (p5);
glEnd ( );
```

```
glBegin (GL_LINES_LOOP);
      glVertex2iv (p1);
      glVertex2iv (p2);
      glVertex2iv (p3);
      glVertex2iv (p4);
      glVertex2iv (p5);
glEnd ( );
```

LINE-DRAWING ALGORITHMS

- A straight-line segment in a scene is defined by the coordinate positions for the endpoints of the segment.
- The line color is loaded into the frame buffer at the corresponding pixel coordinates.
- Stair-step effect (jaggies) produced when a line is generated as a series of pixel positions.

Line Equations

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

$$m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} \qquad (2)$$

$$b = y_0 - m \cdot x_0 \qquad (3)$$

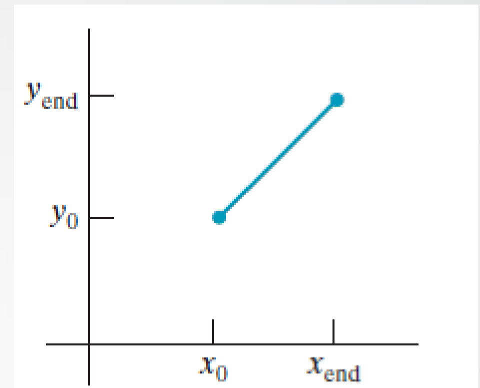


FIGURE Line path between endpoint positions (x_0, y_0) and $(x_{\text{end}}, y_{\text{end}})$.

For any given x interval δx along a line, we can compute the corresponding δy interval *By* from Eq. 2 as

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

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

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

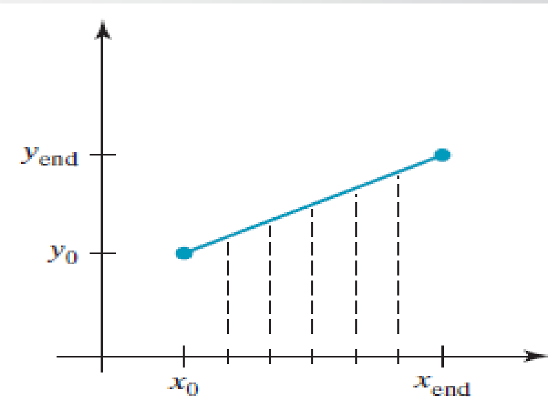


FIGURE Straight-line segment with five sampling positions along the x axis between x_0 and x_{end}.

The *Digital Differential Analyzer* (DDA)

We consider first a line with positive slope,

If the slope is less than or equal to 1, we sample at unit *x* intervals (δx = 1) and compute successive *y* values as

$$y_{k+1} = y_k + m \qquad (6)$$

Subscript *k* takes integer values starting from 0, for the first point, and increases by 1 until the final endpoint is reached. Since *m* can be any real number between 0.0 and 1.0, each calculated *y* value must be rounded to the nearest integer corresponding to a screen pixel position in the *x* column we are processing.

For lines with a positive slope greater than 1.0,we reverse the roles of *x* and *y*. That is, we sample at unit *y* intervals (δy = 1) and calculate consecutive *x* values as

$$x_{k+1} = x_k + \frac{1}{m} \qquad (7)$$

In this case, each computed *x* value is rounded to the nearest pixel position along the current *y* scan line.

Equations 6 and 7 are based on the assumption that lines are to be processed from the left endpoint to the right endpoint

If this processing is reversed, so that the starting endpoint is at the right, then either we have δx=−1 and

$$y_{k+1} = y_k - m$$

or (when the slope is greater than 1) we have δy = −1 with

$$x_{k+1} = x_k - \frac{1}{m}$$

This algorithm is summarized in the following procedure, which accepts as input two integer screen positions for the endpoints of a line segment . Horizontal and vertical differences between the endpoint positions are assigned to parameters **dx** and **dy**. The difference with the greater magnitude determines the value of parameter **steps**. Starting with pixel position **(xo, yo)** ,we determine the offset needed at each step to generate the next pixel position along the line path. We loop through this process **steps** times. If the magnitude of **dx** is greater than the magnitude of **dy** and **xo** is less than **xEnd**, the values for the increments in the *x* and *y* directions are 1 and *m*, respectively. If the greater change is in the *x* direction, but **xo** is greater than **xEnd**, then the decrements −1 and −*m* are used to generate each new point on the line. Otherwise, we use a unit increment (or decrement) in the *y* direction and an *x* increment $\frac{1}{m}$ (or decrement) of .

The *digital differential analyzer* (DDA) Algorithm: (textbook p94)

The *digital differential analyzer* (DDA) is a scan-conversion line algorithm based on calculating either δx or δy , using Eq. 4 or Eq. 5.

```
#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));
   }
}
```

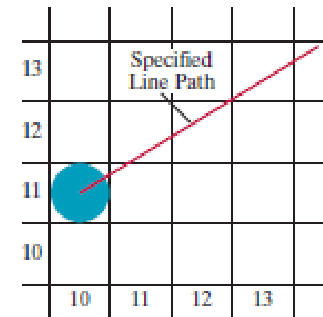


FIGURE 3-8 A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

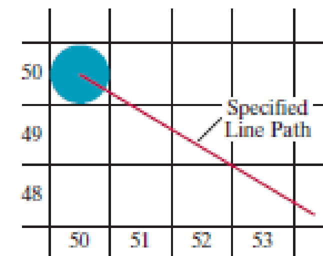


FIGURE 3-9 A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

EXAMPLE 1:

Apply The *digital differential analyzer* (DDA) Algorithm to compute which pixels should be turned on to represent the line from (2,2) to (8,7)

DDA- Digital Differential Analyser

This case is for slope (m) less than 1. Slope (m) =(7-1)/(8-1) = 6/7 .

S-1: x1=1; y1=1; x2=8; y2=7.

S-2: m=(7-1)/(8-1) = 6/7 which is less than 1.

S-3: As m (6/7) is less than 1 therefore x is increased and y is calculated.

S-4 : The step will be x1=x1+1 and y1= y1+6/7

S-5 : The points genrated would be x1=1+1 and Y1=1+(5/7) => 1+0.9=>1.9=> approx 2. So X 1=2 and Y1= ~

.	X1	Y1	Pixel Plotted
p1	2	2	2,2
p2	3	2+6/7 = 2.9	3,3
p3	4	2.9 + 6/7 = 3.8	4,4
p4	5	3.8 + 6/7 = 4.7	5,5
p5	6	4.7 + 6/7 = 5.6	6,6
p6	7	5.6 + 6/7 = 7.0	7,7

The algorithm will stop here as the x value has reached 7.

EXAMPLE 2:

Apply The *digital differential analyzer* (DDA) Algorithm to compute which pixels should be turned on to represent the line from (0,0) to (4,6)

This case is for slope (m) greater than 1. Slope (m) =(6-0)/(4-0) = 6/4 .

S-1: x1=0; y1=0; x2=4; y2=6

S-2: m=(6-0)/(4-0) = 6/4 which is more than 1.

S-3: As m (6/4) is greater than 1 therefore y is increased and x is calculated.

S-4 : Now increase the value of y and calculate value of x.

- To calculate x, take line equation and find x , x2=x1+1/m
- The step will be y1=y1+1 and x1 =x1+1/(6/4) , After Simplification, Every time y1=y1+1 and x1=x1+4/6

	Y1	X1	Pixel Plotted
p0	0	0	(0,0)
p1	1	x1= (0)+4/6=0.67 =1	(1,1)
p2	2	0.67+4/6 = 1.34	(1,2)
p3	3	1.34+4/6=2.01	(2,3)
p4	4	2.01+4/6= 2.68	(3,4)
p5	5	2.68+4/6=3.35	(3,5)
p6	6	3.35+4/6=4.02	(4,6)

EXAMPLE 3:

Apply The *digital differential analyzer* (DDA) Algorithm to compute which pixels should be turned on to represent the line from (2,3) to (9,8)

S-1: x1=2, y1=3 and x2=9 , y2=8.

S-2: Calculate Slope m = (8-3)/(9-2) = 5/7, which is less than 1.

S-3: Since m is less than one that means we would increase x and calculate y.

S-4: So new x would be equal to old x plus 1 🙂 and calculate y as new y = old y +m(slope). — Easy to understand, We mean the following

x1=x1+1 and y1=y1+(5/7)

	X1	Y1	Pixel Plotted
p0	2	3	2,3
p1	**3**	**3+5/7 => 26/7 => 26/7=>3.71**	**3,4**
p2	**4**	**3.71 + 5/7 = 4.42**	**4,4**
p3	**5**	**4.42 + 5/7 =5 .13**	**5,5**
p4	**6**	5.13 + 5/7 =5.84	**6,6**
p5	**7**	**5.84 + 5/7 = 6.55**	**7,7**
p6	8	6.55+5/7=7.26	8,7
p7	9	7.26+5/7=7.97	9,8

The algorithm would stop here as we have reached the end point of the line (9,8)

Exercise :

Apply The *digital differential analyzer* (DDA) Algorithm to compute which pixels should be turned on to represent the line from (20,10) to (30,18)

dx	dy	steps	k	X Increment	Y Increment	x	y	(x,y)
						20	10	(20,10)
10	8	10	0	1	0.8	21	10.8	(21,11)
			1			22	11.6	(22,12)
			2			23	12.3	(23,12)
			3			24	13.1	(24,13)
			4			25	13.9	(25,14)
			5			26	14.7	(26,15)
			6			27	15.5	(27,15)
			7			28	16.3	(28,16)
			8			29	17.1	(29,17)
			9			30	17.9	(30,18)

Bresenham's Line Algorithm

Bresenham's Line-Drawing Algorithm for $|m| < 1.0$

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Set the color for 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

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

   Otherwise, 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.

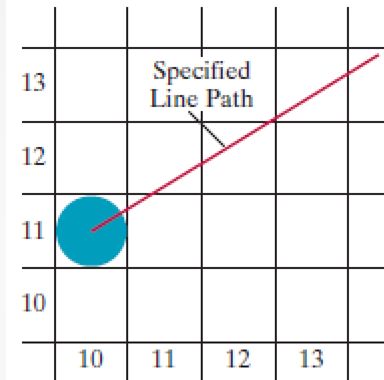


FIGURE A section of a display screen where a straight-line segment is to be plotted, starting from the pixel at column 10 on scan line 11.

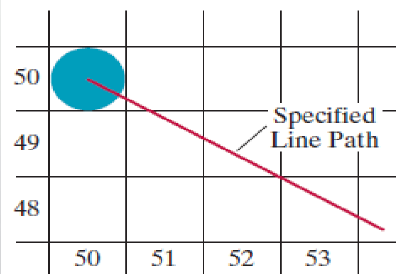


FIGURE A section of a display screen where a negative slope line segment is to be plotted, starting from the pixel at column 50 on scan line 50.

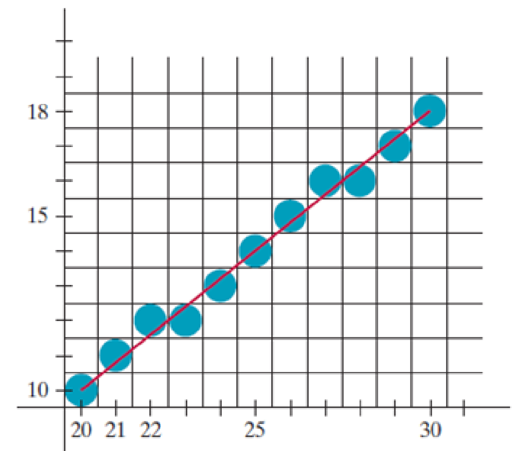
Bresenham's Line Algorithm

EXAMPLE :(textbook p136-137)

To illustrate the algorithm, we digitize the line with endpoints (20, 10) and (30, 18).

- Draw the line with endpoints (20,10) and (30, 18).
  - **Δx=30-20=10, Δy=18-10=8,**
  - **$p_0 = 2\Delta y - \Delta x$=16-10=6**
  - **$2\Delta y$=16**, and **$2\Delta y - 2\Delta x$=-4**
- Plot the initial position at (20,10), then

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 Line Algorithm

An implementation of Bresenham line drawing for slopes in the range $0 < m < 1.0$ is given in the following procedure

```
#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);
   }
}
```

End of Lecture
Good Luck!

See you

in next lecture...