GRAPHICS OUTPUT PRIMITIVES

Tara Qadir
2024-2025
3rd Stage
Computer department
College of science
University of Sulymaniah

Outline

- Points and Lines
- Line Drawing
- Line-Drawing Algorithms
  - Line Equation Algorithm
  - Digital Differential Analyzer (DDA) Algorithm
  - Bresenham’s Line Algorithm
- Polygons
- Rectangles
- Curves and Curved Surfaces
- Specifying Vertexes
- Geometric Drawing Primitives
- Point Details
- Line Details
- Polygon Details

Scan Conversion Definition

- It is a process of representing graphics objects from a collection of pixels.
- The graphics objects are continuous, The pixels used are discrete.
- Each pixel can have either on or off state.
- 0 is represented by pixel off. 1 is represented using pixel on. Using this ability graphics computer represent picture having discrete dots.
- For generating graphical object, many algorithms have been developed.

Advantage of developing algorithms for scan conversion

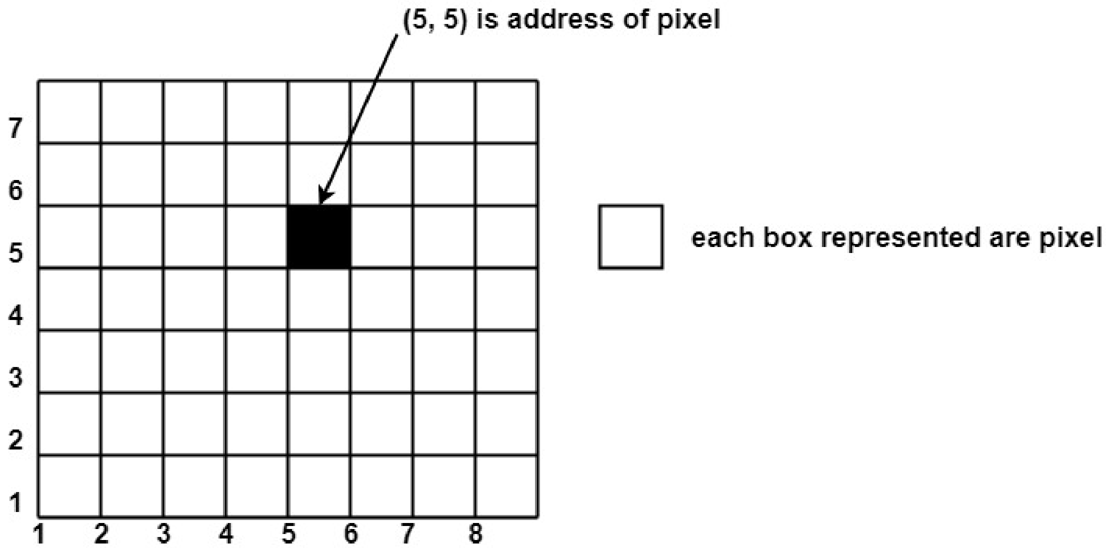
1. Algorithms can generate graphics objects at a faster rate.
2. Using algorithms memory can be used efficiently.
3. Algorithms can develop a higher level of graphical objects.

Examples of objects which can be scan converted

- ✓ Point
- ✓ Line
- ✓ Sector
- ✓ Arc
- ✓ Ellipse
- ✓ Rectangle
- ✓ Polygon مضلع
- ✓ Characters
- ✓ Filled Regions

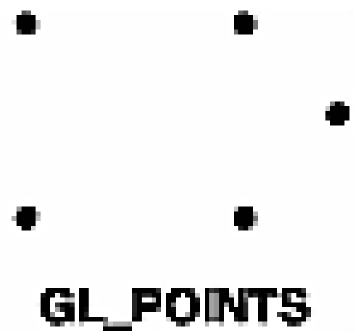
Pixel or Point:

- The term pixel is a short form of the picture element.
- It is also called a point or dot. It is the smallest picture unit accepted by display devices.
- Lines, circle, arcs, characters; curves are drawn with closely spaced pixels. To display the digit or letter matrix of pixels is used.



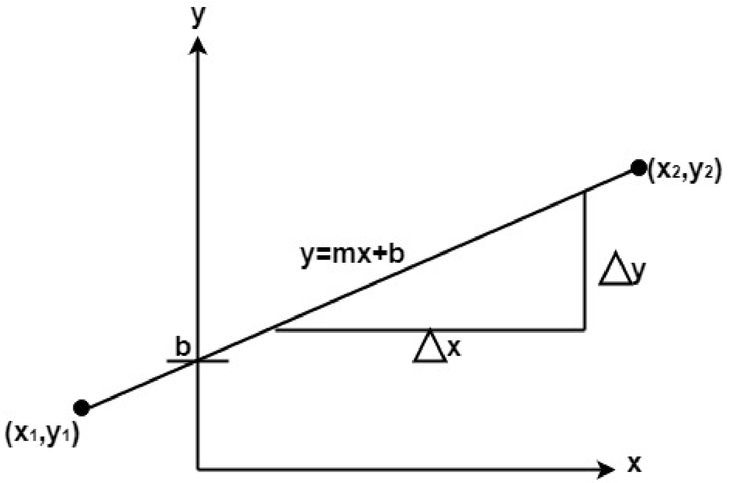
- Pixels are generated using commands.
- example in OpenCL :

```
glBegin(GL.GL_POINTS);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(0.0f, 3.0f);
    glVertex2f(4.0f, 3.0f);
    glVertex2f(6.0f, 1.5f);
    glVertex2f(4.0f, 0.0f);
glEnd()
```



Scan Converting a Straight Line

- A straight line may be defined by two endpoints & an equation.
- In figure bellow the two endpoints are described by (x_1,y_1) and (x_2,y_2). The equation of the line is used to determine the x, y coordinates of all the points that lie between these two endpoints.



Properties of Good Line Drawing Algorithm:

- Line should appear Straight
- Lines should have constant density
- Line density should be independent of line length and angle
- Line should be drawn rapidly

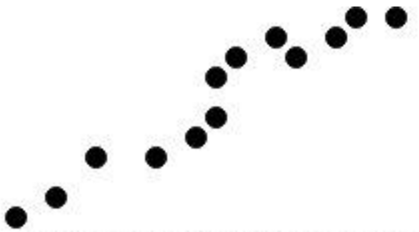
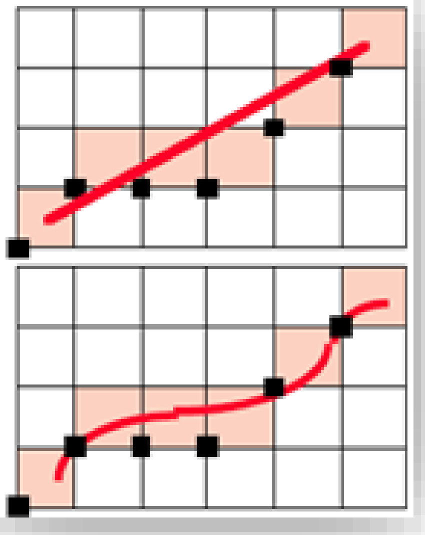
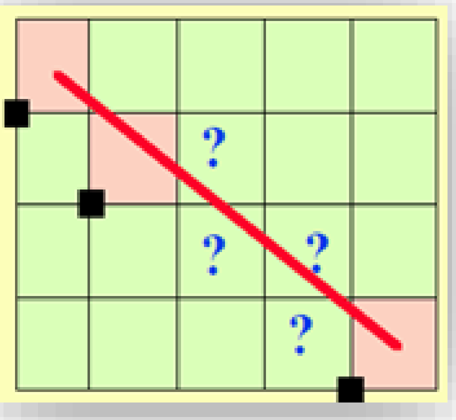


Fig: O/P from a poor line generating algorithm



Line Drawing

- ❑ Some useful definition:
  - ❑ Rasterization: process of determining which pixels provide the best approximation to a desired line on the screen.



Line-Drawing Algorithms

Algorithm for line Drawing:

✓ 1. Direct use of line equation

✓ 2. DDA (Digital Differential Analyzer)

✓ 3. Bresenham's Algorithm

Direct use of line equation

✓ $$y = x \cdot m + b$$

Cartesian slope-intercept equation

where **m** is the slope and **b** is the y intercept

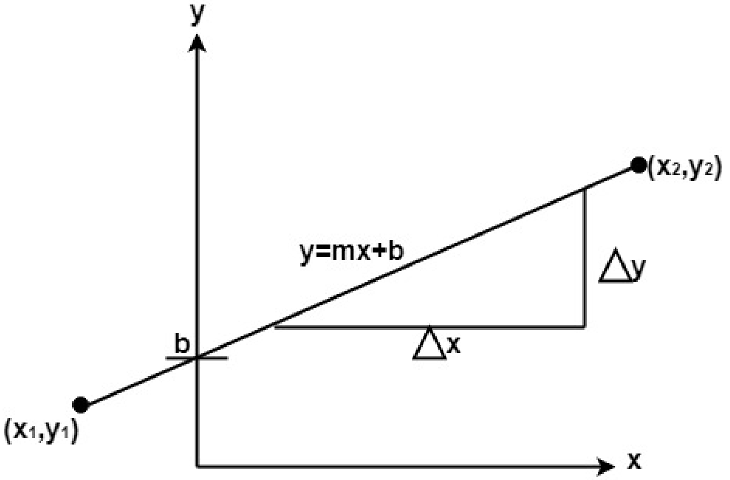
ميل

$$\boxed{m = \frac{y_{end} - y_0}{x_{end} - x_0}}$$

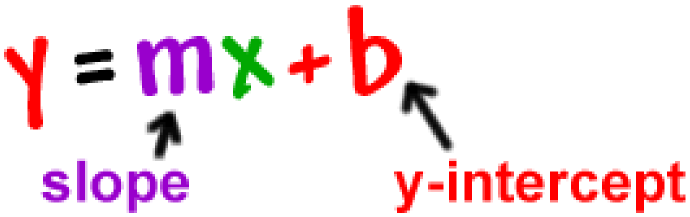
where (x_0, y_0) and (x_{end}, y_{end}) are the two endpoints of a line segment

$$\boxed{b = y_0 - m \cdot x_0}$$

✓ $$\boldsymbol{y - y1 = m(x - x1)}$$



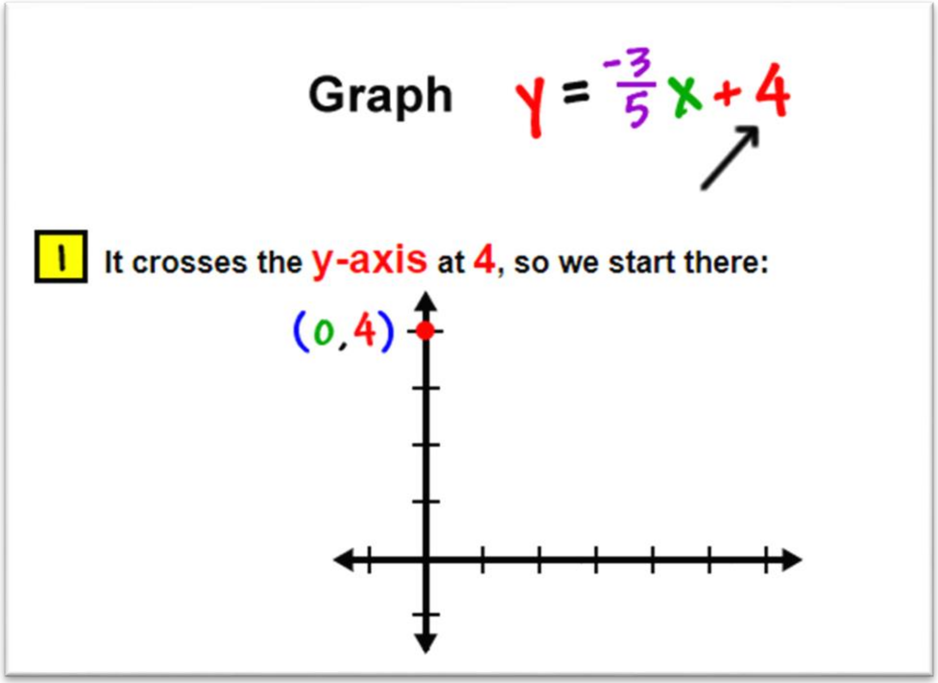
Example1 for line equation



$y = mx + b$

slope

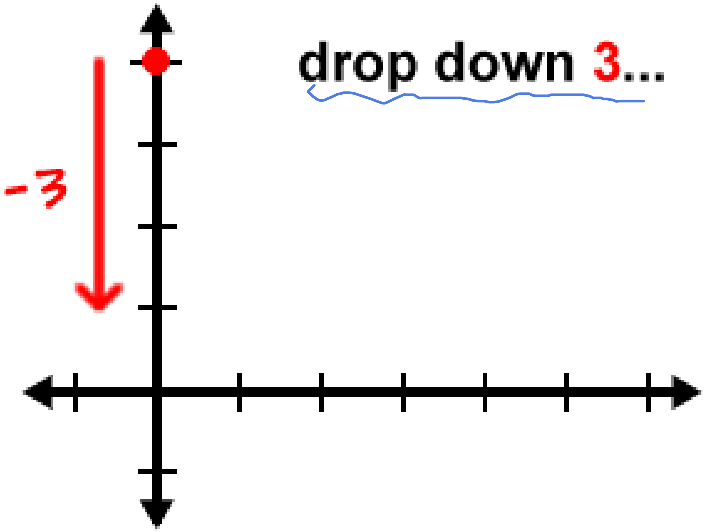
y-intercept

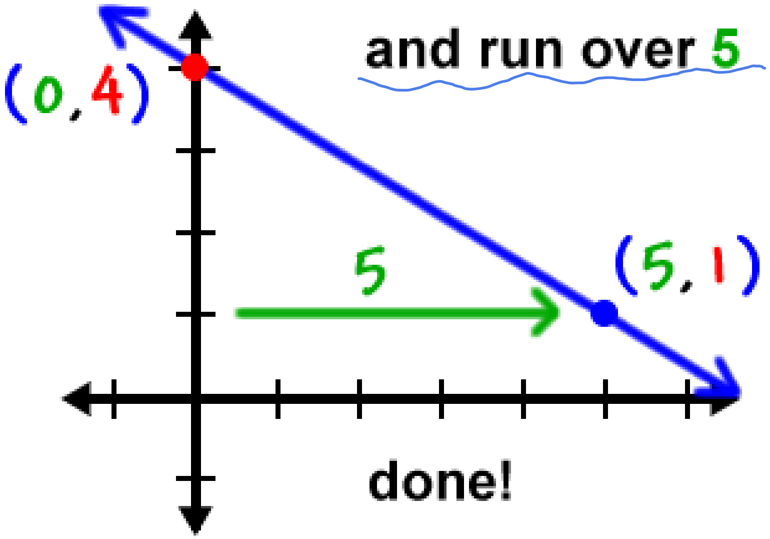


Graph $y = \frac{-3}{5}x + 4$

1 It crosses the y-axis at 4, so we start there:

2 the slope is $\frac{-3}{5}$ so we

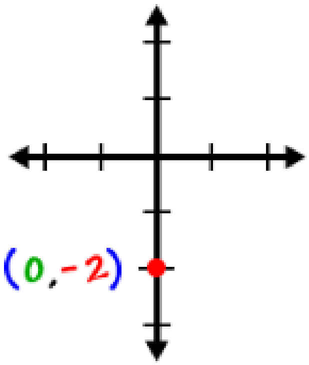

drop down 3...


and run over 5

done!

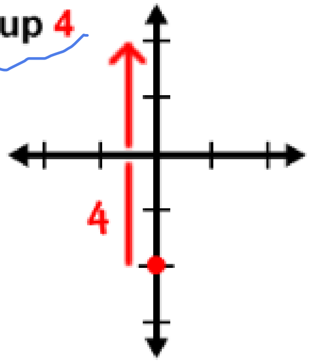
Graph $y = 4x - 2$

$\boxed{1}$ It crosses the y-axis at y = -2, so we start there:

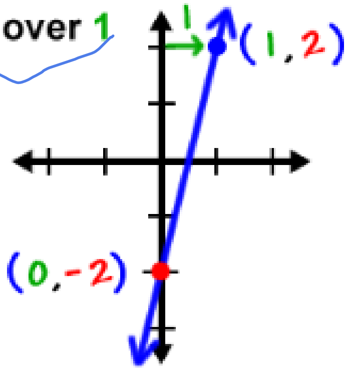


$\boxed{2}$ The slope is 4 which is really $\frac{4}{1}$...

rise up 4



run over 1



Example2 for line equation

By using line equation :

$$y - y_1 = m(x - x_1)$$

Let's find the equation of the line that passes through the points

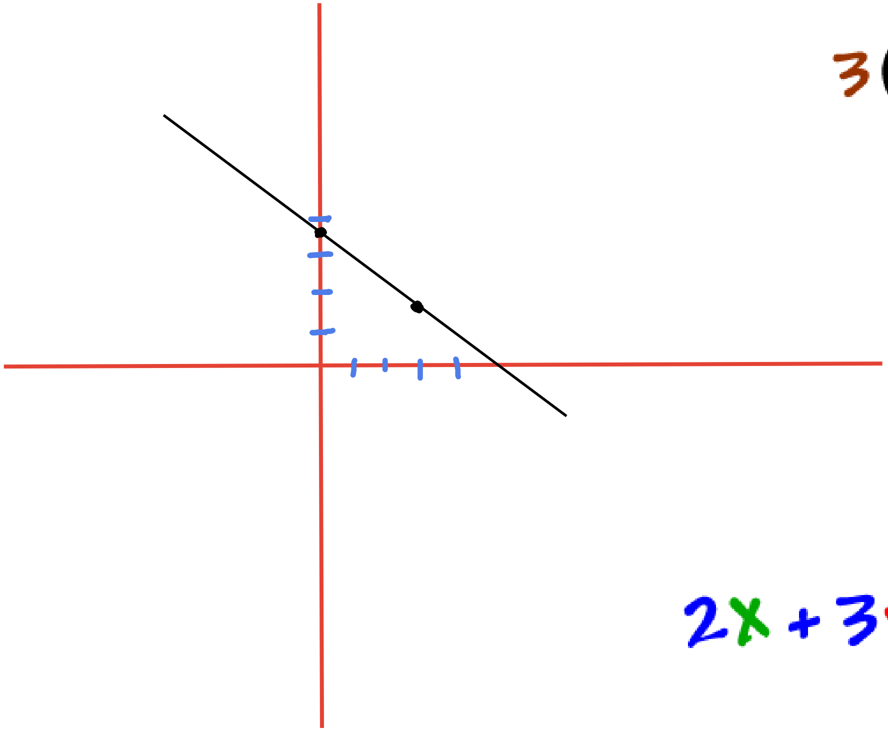
$$(\underset{x_1}{1}, \underset{y_1}{3}) \text{ and } (\underset{x_2}{-2}, \underset{y_2}{5})$$

STEP 1: Find the slope

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{5 - 3}{-2 - 1} = \frac{2}{-3} = \frac{-2}{3}$$

STEP 2: Now, use the point-slope formula with one of our points,

$(1, 3)$, and $m = \frac{-2}{3}$.



$$y - y_1 = m(x - x_1)$$

$$y - 3 = \frac{-2}{3}(x - 1)$$ multiply by 3

$$3(y - 3) = 3\left(\frac{-2}{3}\right)(x - 1)$$

$$3y - 9 = -2(x - 1)$$

$$\begin{aligned} 3y - 9 &= -2x + 2 \\ +9 &\qquad\quad +9 \\ \hline 3y &= -2x + 11 \end{aligned}$$

$$\swarrow \qquad \searrow$$

$$2x + 3y = 11$$ or $$y = \frac{-2}{3}x + \boxed{\frac{11}{3}} \approx 3667$$

Digital Differential Analyzer (DDA)

- Form left to right (dx = $x_{end} - x_0$ and dy = $y_{end} - y_0$)

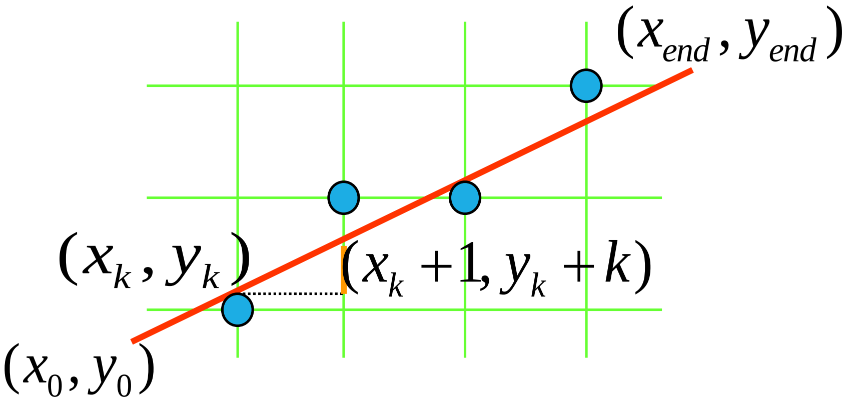
$m = \frac{\Delta y}{\Delta x}$

  - |m| <= 1
    - $x_{k+1} = x_k + 1$
    - $y_{k+1} = y_k + m$

      where k = 0, 1, 2, …, dx
    - pixel [x, round(y)]
  - |m| > 1
    - $x_{k+1} = x_k + \frac{1}{m}$
    - $y_{k+1} = y_k + 1$

      where k = 0, 1, 2, …, dy
    - pixel [round(x), y]

هواس العكس



Digital Differential Analyzer (DDA) (2)

- Form right to left (dx = $x_{end} - x_0$ and dy = $y_{end} - y_0$)
  - |m| <= 1
    - $x_{k+1} = x_k - 1$
    - $y_{k+1} = y_k - m$

      where k=0, 1, 2, …, dx
    - pixel [x, round(y)]
  - |m| > 1
    - $x_{k+1} = x_k - \frac{1}{m}$
    - $y_{k+1} = y_k - 1$

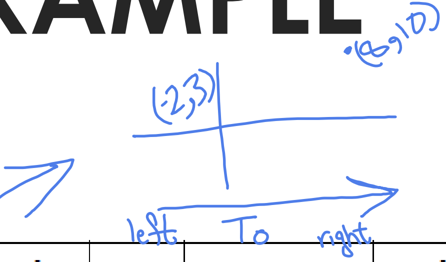
      where k=0, 1, 2, …, dy
    - pixel [round(x), y]

DDA Drawbacks:

(مهم) حرس مقارنة
س اسواع المبنودات لازم الحلول

1. pixel may go away from the line path due to round-off error
2. time consuming due to rounding and floating-point operations

DDA EXAMPLE



- For example: (-2, 3) and (8, 10)

  Vertex

  $x_0 = -2, y_0 = 3,$

  $x_{end} = 8, y_{end} = 10$

  $dx = 10, dy = 7$

  $m = 0.7$

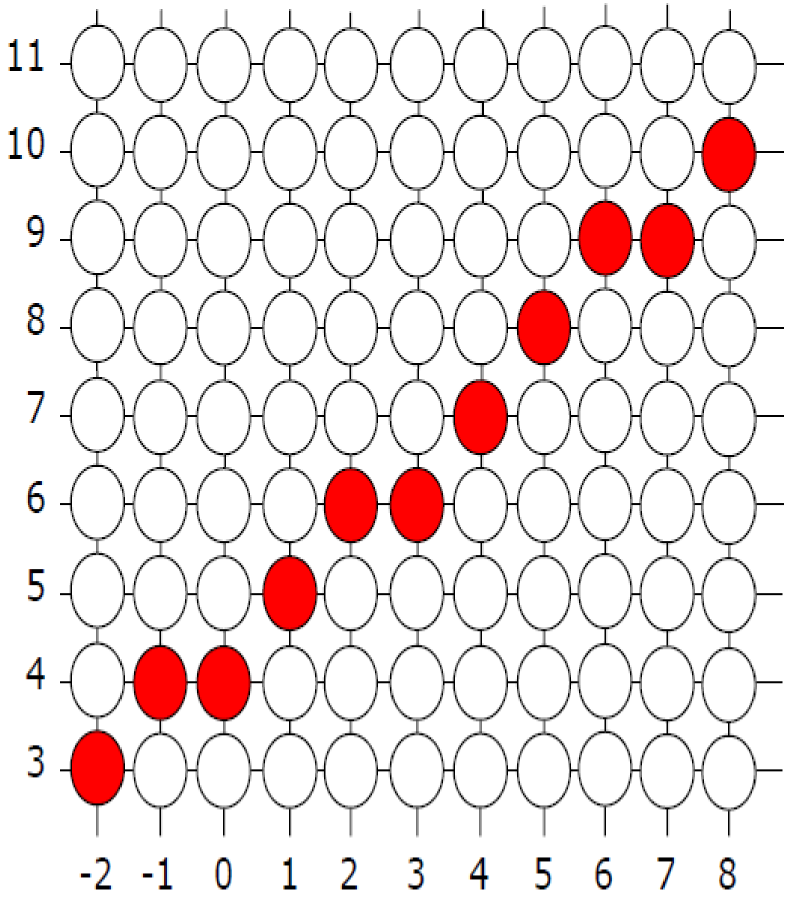
dx
$\Delta x = 8-(-2)=10$

dy
$\Delta y = 10-3 = 7$

$m = \frac{\Delta y}{\Delta x} = \frac{7}{10}$

k	x	y	pixel
0	-2	3.0 ≈ 3	(-2, 3)
1	-1	3.7 ≈ 4	(-1, 4)
2	0	4.4 ≈ 4	(0, 4)
3	1	5.1 ≈ 5	(1, 5)
4	2	5.8 ≈ 6	(2, 6)
5	3	6.5 ≈ 6	(3, 6)
6	4	7.2 ≈ 7	(4, 7)
7	5	7.9 ≈ 8	(5, 8)
8	6	8.6 ≈ 9	(6, 9)
9	7	9.3 ≈ 9	(7, 9)
10	8	10.0 ≈ 10	(8, 10)

+0.7
+0.7



Bresenham’s Line (1)

ما بع
rounding
ومشكل اي ميله

rounding → تقريب

- ❑ **Bersenham’s line** algorithm is an efficient method for scan converting straight lines , in that it use only integer addition ,subtraction and multiplication by 2 and floating point operations

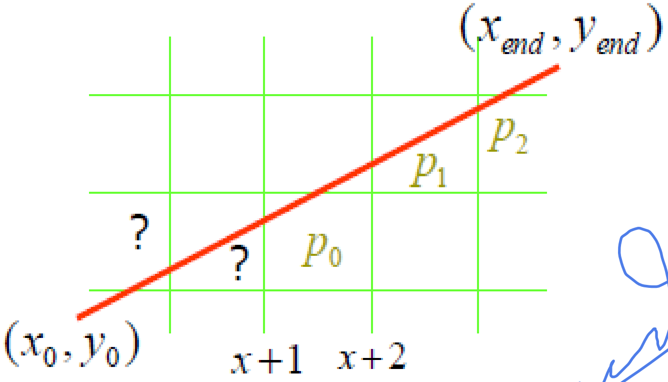
- ❑ The lines drawn are of superior quality as compared to DDA method

Bresenham's Line (2)

- Positive m < 1.0
- (x_0, y_0), (x_{end}, y_{end})
- $\Delta x = x_{end} - x_0$,

  $\Delta y = y_{end} - y_0$
- $\boxed{p_0 = 2\Delta y - \Delta x}$
- At each x_k

  where $k = 0, 1, 2, \ldots, \Delta x - 1$
- If $\boxed{p_0 < 0}$ then (x_k+1, y_k) and $\boxed{p_{k+1} = p_k + 2\Delta y}$
- Otherwise (x_k+1, y_k+1) and $\boxed{p_{k+1} = p_k + 2(\Delta y - \Delta x)}$
- Repeat Δx-1 times



2024/11/10

Bresenham's Line (3)

- For example:

  (-2, 3) and (8, 10)

  $x_0 = -2, y_0 = 3,$

  $x_{end} = 8, y_{end} = 10$

  $\Delta x = 10, \Delta y = 7$

  $m = 0.7$

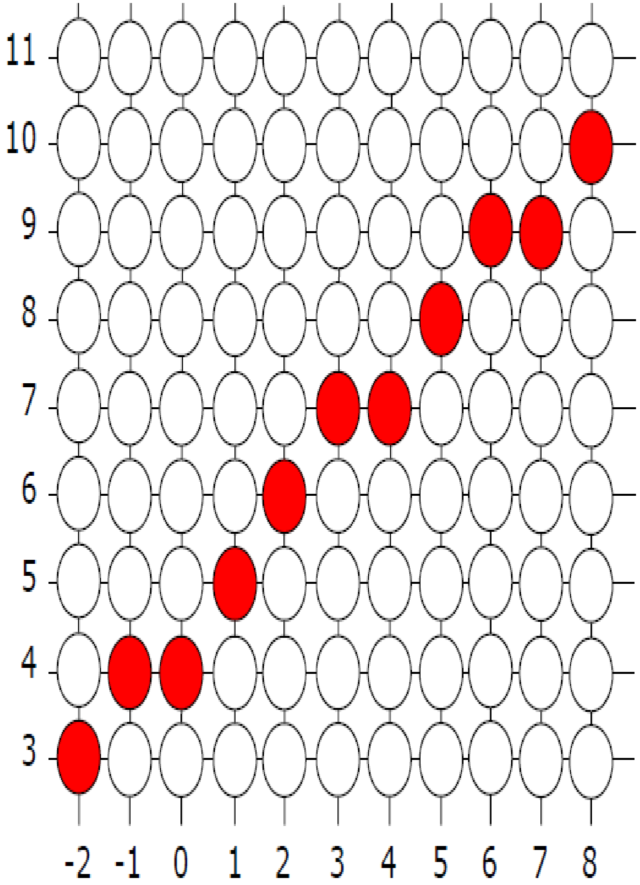
$p_0 = 2\,\Delta y - \Delta x =$

$14 - 10 = 4$

$2\Delta y = 14$

$2(\Delta y - \Delta x) = -6$

p	x	y	pixel
4	-2	3	(-2, 3)
-2	-1	4	(-1, 4)
12	0	4	(0, 4)
6	1	5	(1, 5)
0	2	6	(2, 6)
-6	3	7	(3, 7)
8	4	7	(4, 7)
2	5	8	(5, 8)
-4	6	9	(6, 9)
10	7	9	(7, 9)
4	8	10	(8, 10)



Exercise

1. Draw the line for following equation :

Graph $y = \frac{1}{2}x - 3$

2. Draw the following lines using DDA and Bresenham's algorithms :

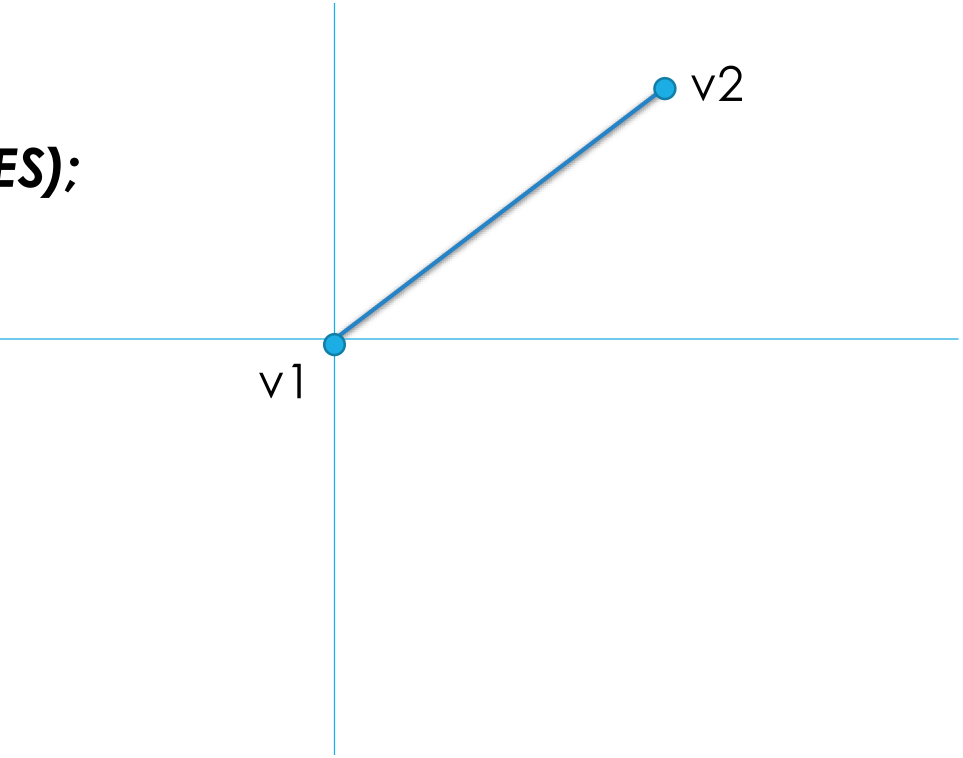
- (from left to right): (-1, 2) and (7, 8)
- (from right to left) : (6, 2) and (-4, -3)

عند استخدام bresenham راح نأخذ نقطة 2 حتى بعد الزيادة تصل الى نقطة ا

3. Draw a graph after finding equation of passing through the points (-4 , 5) and (2 , -3).

Line drawing using OpenGL

```
glBegin(GL.GL_LINES);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(5.0f, 5.0f);
glEnd();
```



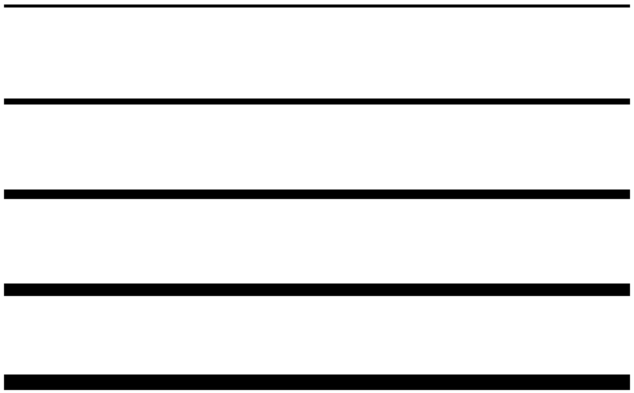
Line Details

❑Wide Lines

- ❑ *void glLineWidth(float width);*

  sets the width in pixels for rendered lines; width must be greater than 0.0 and by default is 1.0

- ❑ widths of 1, 2, and 3 draw lines 1, 2, and 3 pixels wide, and so on



Line Details (2)

❑Stippled Lines

- ❑ To make stippled (dotted or dashed) lines, use the command *glLineStipple()* , and then you enable line stippling with *glEnable()*

  glLineStipple(1, 0x3F07);

  glEnable(GL.GL_LINE_STIPPLE);

- ❑ *void glLineStipple(int factor, short pattern);*
  - ❑ the pattern is a 16-bit series of 0s and 1s
  - ❑ a 1 indicates that drawing occurs, and 0 that it does not
  - ❑ the pattern can be stretched out by using factor
  - ❑ factor is lie between 1 and 255
  - ❑ enabled by passing **GL_LINE_STIPPLE** to *glEnable()*, and disabled by *glDisable()*
  - ❑ if the pattern 0x3F07 (0011111100000111 in binary), a line would be 3 pixels on, then 5 off, 6 on, and 2 off
  - ❑ if factor = 2, the pattern would have been: 6 pixels on, 10 off, 12 on, and 4 off

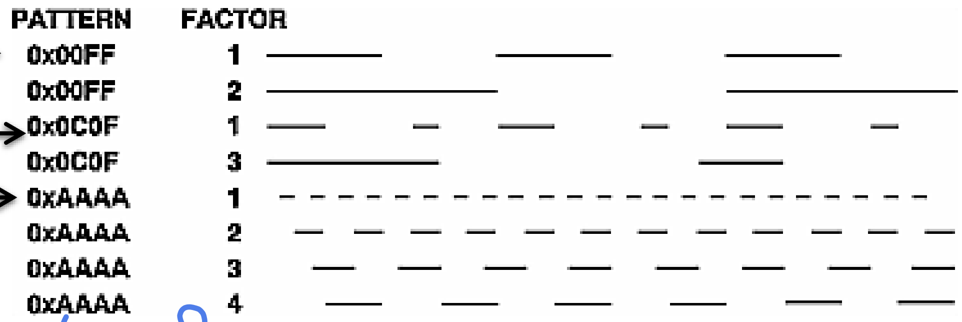
Line Details (3)

Stippled Lines

0000 0000111111111 →

1100 0000 1111 →

1010101010101010 →



PATTERN	FACTOR
0x00FF	1
0x00FF	2
0x0C0F	1
0x0C0F	3
0xAAAA	1
0xAAAA	2
0xAAAA	3
0xAAAA	4

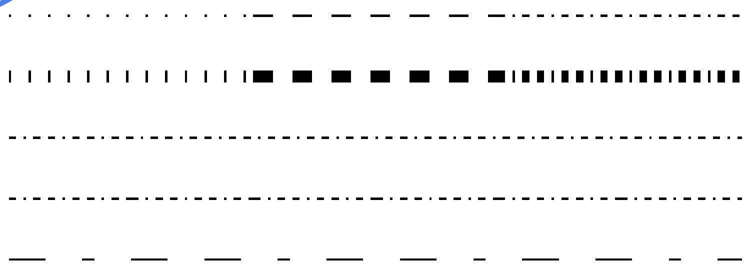
01011111 0101, factor=2

00 11 00

11111 11111

راح يضاعف

Wide Stippled Lines



Specifying Vertexes

- With OpenGL, all geometric objects are ultimately described as an ordered set of **vertexes**

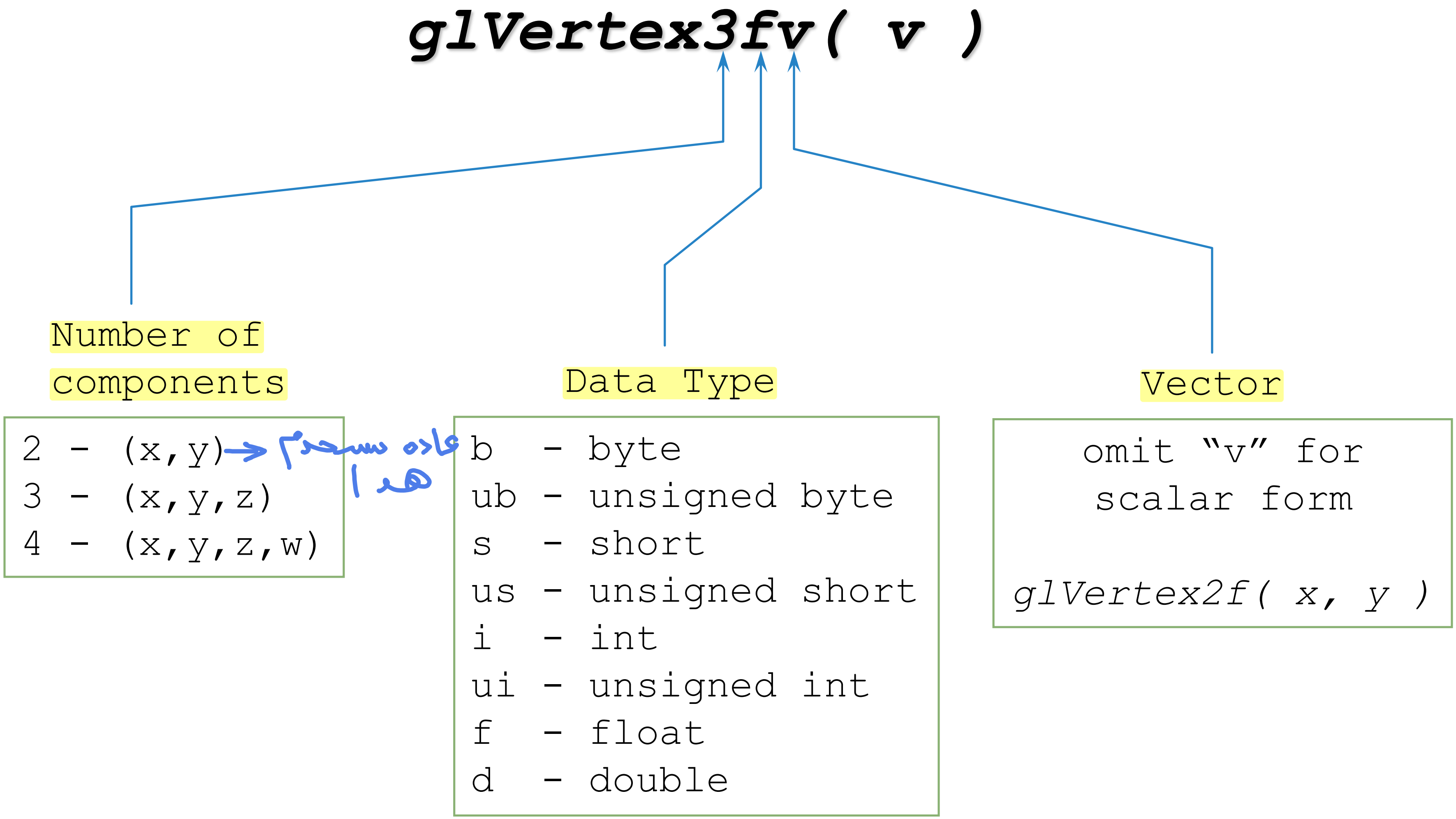
  void glVertex{234}{sifd}[v](TYPE coords);

  Specifies a vertex for use in describing a geometric object

- Legal Uses of *glVertex*()*

  glVertex2s(2, 3);
  glVertex3d(0.0 , 0.0, 3.1415926535898);
  glVertex4f(2.3f, 1.0f, -2.2f, 2.0f);
  GLdouble dvect[3] = {5.0, 9.0, 1992.0};
  glVertex3dv(dvect);

Specifying Vertexes (2)



glVertex3fv(v)

Number of components

```
2 - (x,y) → عاده نستخدم هدا
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

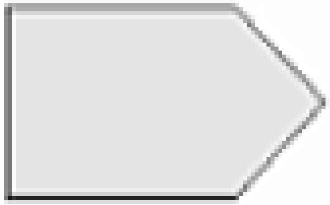
glVertex2f(x, y)

Geometric Drawing Primitives

- ❑ We bracket each set of vertexes between a call to *glBegin()* and a call to *glEnd()*
- ❑ The argument passed to *glBegin()* determines what sort of geometric primitive is constructed from the vertexes

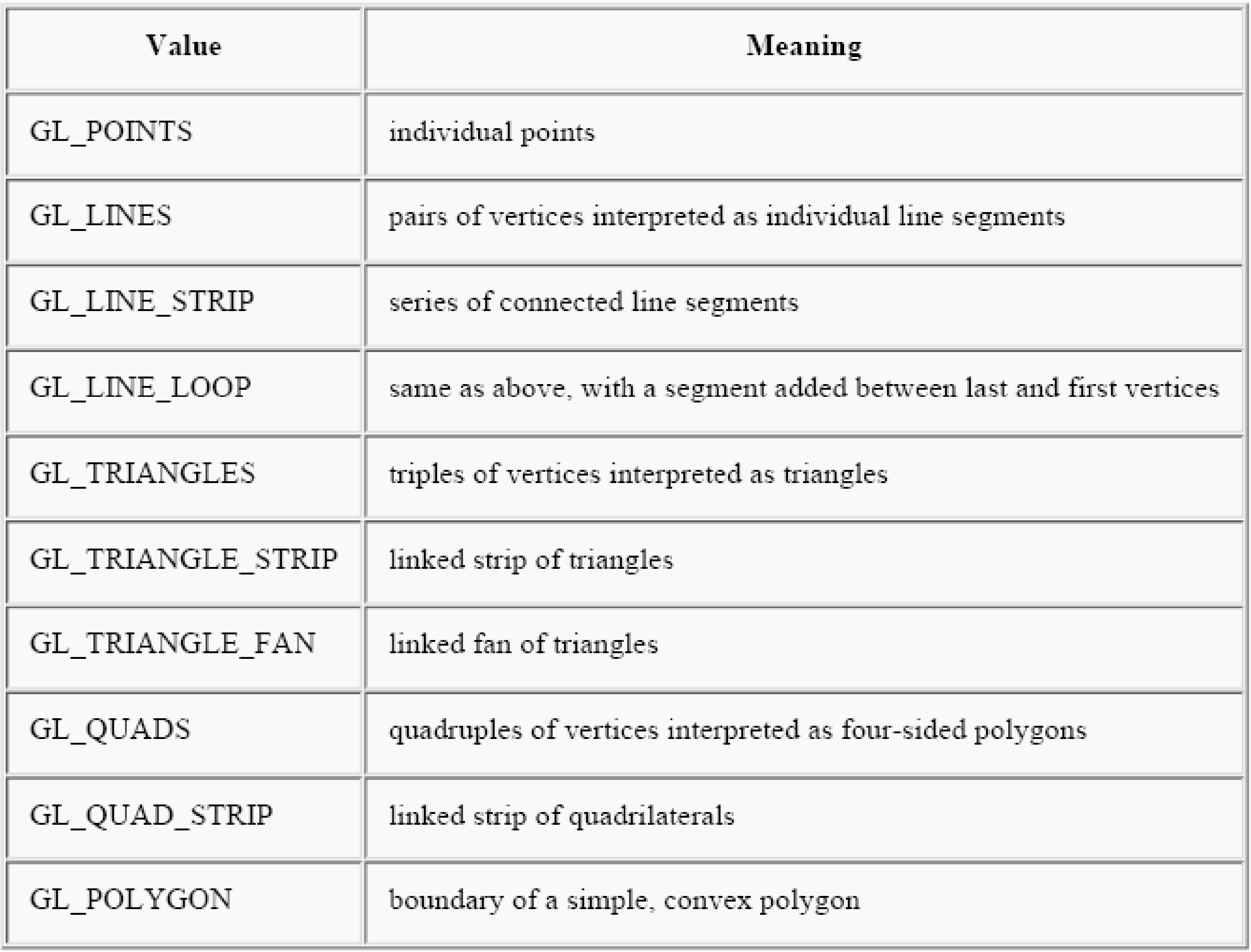
```
glBegin(GL.GL_POLYGON);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(0.0f, 3.0f);
    glVertex2f(4.0f, 3.0f);
    glVertex2f(6.0f, 1.5f);
    glVertex2f(4.0f, 0.0f);
glEnd();
```

[illegible] [illegible]



Geometric Drawing Primitives (3)

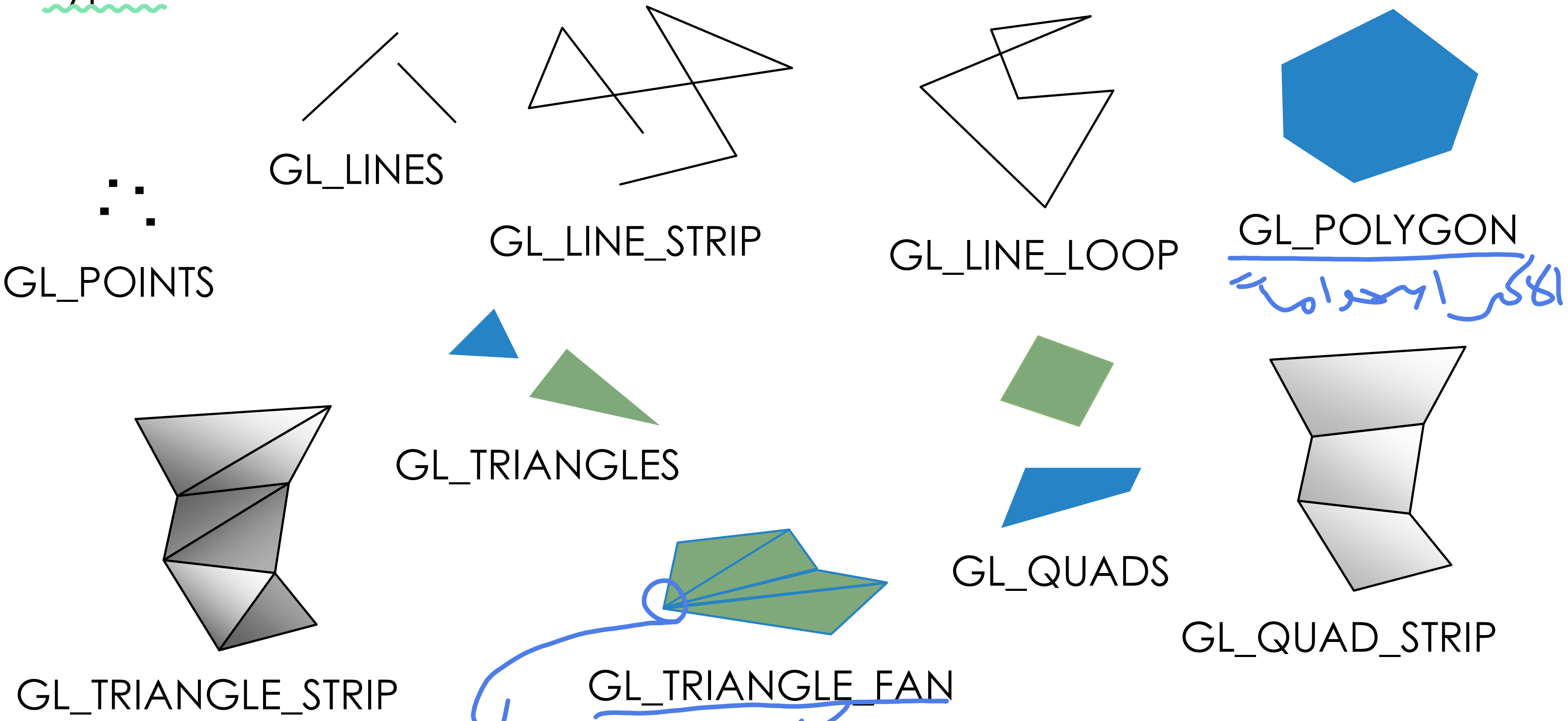
Geometric Primitive Names and Meanings



Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon

Geometric Drawing Primitives (4)

Geometric Primitive Types



الاكثر استخداماً

مع نقطة مشتركة

Basic State Management

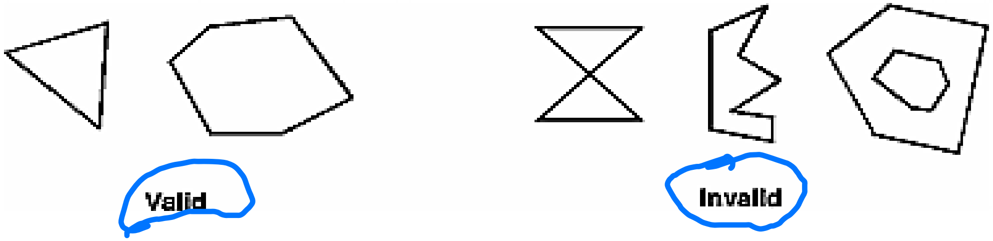
- OpenGL maintains many states and state variables
- By default, most of these states are initially inactive
- To turn on and off many of these states, use these two simple commands:
  - *void glEnable(GLenum cap);* → turns on
  - *void glDisable(GLenum cap);* → turns off
- There are over 40 enumerated values that can be passed as a parameter to *glEnable()* or *glDisable()*
- **Example:**

GL_BLEND GL_FOG GL_LINE_STIPPLE GL_POLYGON_STIPPLE

GL_LIGHTING …

Polygons

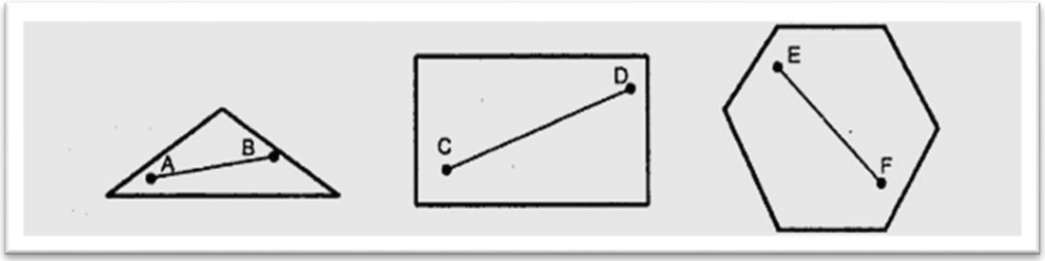
- **Polygons** : is a representation of the surface. It is primitive which is closed in nature. It is formed using a collection of lines. It is also called as many-sided figure. The lines combined to form polygon are called sides or edges. The lines are obtained by combining two vertices.
  - **First**, the edges of OpenGL polygons cannot intersect
  - **Second**, OpenGL polygons must be convex, meaning that they cannot have indentations



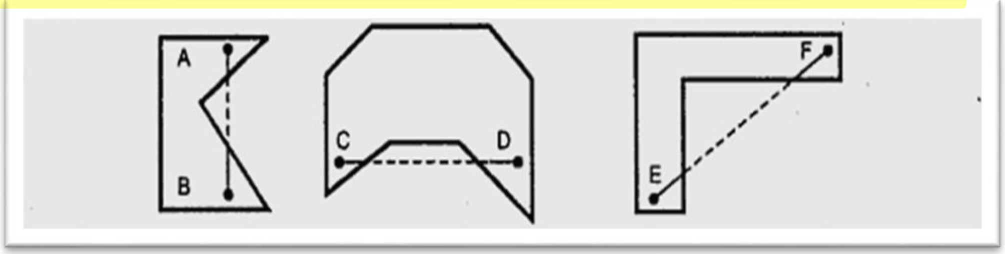
Valid and Invalid Polygons

Polygons (2)

- The classification of polygons is based on where the line segment joining any two points within the polygon is going to lie
  - **Convex** : is a polygon in which the line segment joining any two points within the polygon lies completely inside the polygon



  - **Concave:** is a polygon in which the line segment joining any two points within the polygon may not lie completely inside the polygon



Polygons (3)

- To add polygon to a graphic system, we must first decide how to represent it
- There are three approaches:

  First: Polygon drawing primitive approach

  Second: Trapezoid primitive approach

  Third: Line and Point approach → هذه الطريقة الي نستخدمها

- Most of the graphic devices do not provide any polygon support at all. In such cases polygons are represented using lines and points

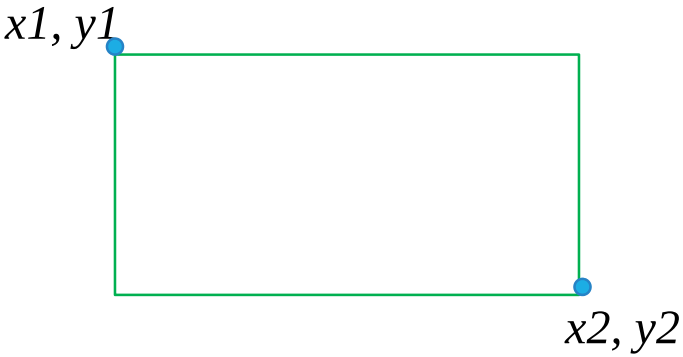
Rectangles

- OpenGL provides a filled-rectangle drawing primitive, **glRect**

  void glRect{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);

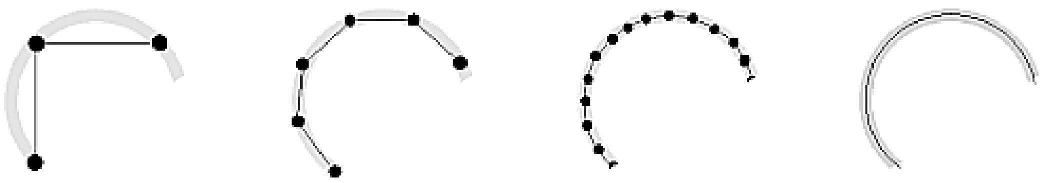
  *void glRect{sifd}v(TYPE*v1, TYPE*v2);*

  - Draws the rectangle defined by the corner points (x1, y1) and (x2, y2). The rectangle lies in the plane z=0 and has sides parallel to the x- and y-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an (x, y) pair



Curves and Curved Surfaces

- Any smoothly curved line or surface can be approximated by short **line segments** or **small polygonal regions**
- Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved



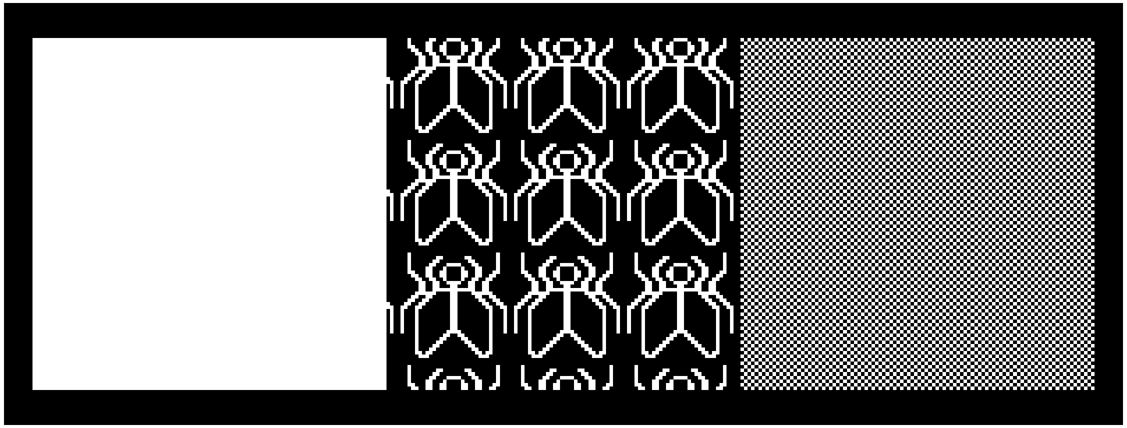
Approximating Curves

Polygon Details

- ❑Polygons drawn as **Points**, **Lines**, or **Solids** (by default solid)
- ❑A polygon has two sides - Front and Back
- ❑By default, both front and back faces are drawn in the same way
  - ❑ *void glPolygonMode(GLenum face, GLenum mode);*
    - ❑ controls the drawing mode for a polygon's front and back faces
    - ❑ the parameter face can be **GL_FRONT**, **GL_BACK**, or **GL_FRONT_AND_BACK**
    - ❑ mode can be **GL_POINT**, **GL_LINE**, or **GL_FILL** to indicate whether the polygon should be drawn as points, lined, or filled
    - ❑ by default, both the front and back faces are drawn filled
  - ❑ **Example**:
    - ❑ *glPolygonMode(GL_FRONT, GL_FILL);*
    - ❑ *glPolygonMode(GL_BACK, GL_LINE);*

Polygon Details (2)

- **Stippling Polygons**
  - By default, filled polygons are drawn with a solid pattern
  - can be filled with a 32 X 32-bit stipple pattern with *glPolygonStipple()*
  - *void glPolygonStipple(int mask, int offset);*
    - mask is a pointer to a 32 × 32 bitmap that's interpreted as a mask of 0s and 1s
    - where a 1 appears, the pixel is drawn, and where a 0 appears, nothing is drawn
  - enabled and disabled by using *glEnable()* and *glDisable()* with **GL_POLYGON_STIPPLE**



Polygon Details (3)

❑Reversing Polygon Faces

- ❑ vertexes of polygons appear in counterclockwise order called front-facing
- ❑ we can swap faces by using the function *glFrontFace()*

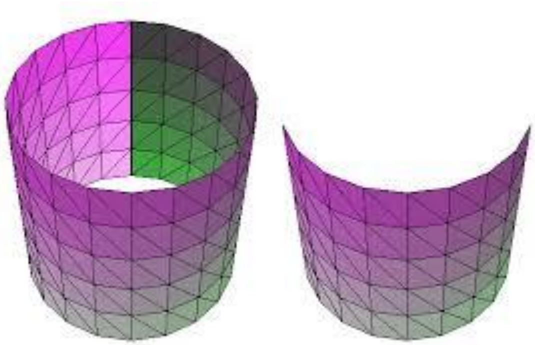
  void glFrontFace(GLenum mode);

- ❑ by default, mode is **GL_CCW** (counterclockwise orientation)
- ❑ if mode is **GL_CW**, a clockwise orientation are considered front-facing

Polygon Details (4)

Culling Polygon Faces

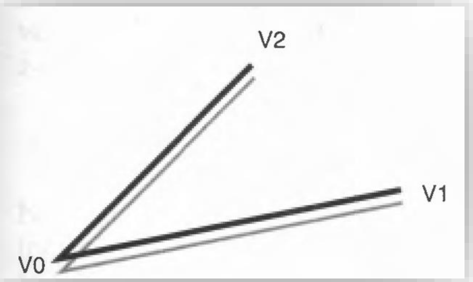
- to discard front- or back-facing polygons, use the command *glCullFace()* and enable culling with *glEnable()*
- *void glCullFace(GLenum mode);*
  - indicates which faces should be discarded (culled)
  - the mode is either **GL_FRONT**, **GL_BACK**, or **GL_FRONT_AND_BACK** to indicate front-facing, back-facing, or all polygons
- culling enabled using *glEnable()* with **GL_CULL_FACE**, and disabled using *glDisable()*



Polygon Details (5)

❑Marking Polygon Boundary Edges

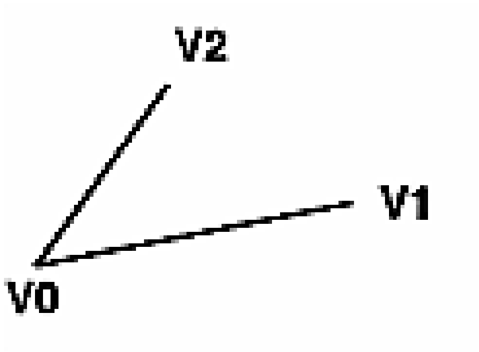
- ❑ *glEdgeFlag*()* used between *glBegin()* and *glEnd()*, and it affects all the vertexes specified after it
- ❑ it applies only to polygons, triangles, and quads, not to strips of triangles or quads
- ❑ *void glEdgeFlag(boolean flag);*
- ❑ *void glEdgeFlagv(boolean[] flag, int offset);*
  - ❑ if flag is **GL_TRUE**, and any vertexes are considered to precede bou function is called again with flag being **GL_FALSE**



Polygon Details (6)

❑ **Example:**

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glBegin(GL_POLYGON);
   glEdgeFlag(GL_TRUE);
   glVertex3fv(V0, 0);
   glEdgeFlag(GL_FALSE);
   glVertex3fv(V1, 0);
   glEdgeFlag(GL_TRUE);
   glVertex3fv(V2, 0);
glEnd();
```



Reference

- https://www.javatpoint.com/computer-graphics-scan-converting-a-point