Computer Graphics

Unit 1

Contents

- Introduction, graphics primitives - pixel, resolution, aspect ratio, frame buffer. Display devices, applications of computer graphics.
- Introduction to OpenGL - OpenGL architecture, primitives and attributes, simple modelling and rendering of two and three dimensional geometric objects, GLUT, interaction, events and call-backs picking.(Simple Interaction with the Mouse and Keyboard)
- Scan conversion: Line drawing algorithms: Digital Differential Analyzer (DDA), Bresenham. Circle drawing algorithms: DDA, Bresenham, and Midpoint.

Graphics Primitives

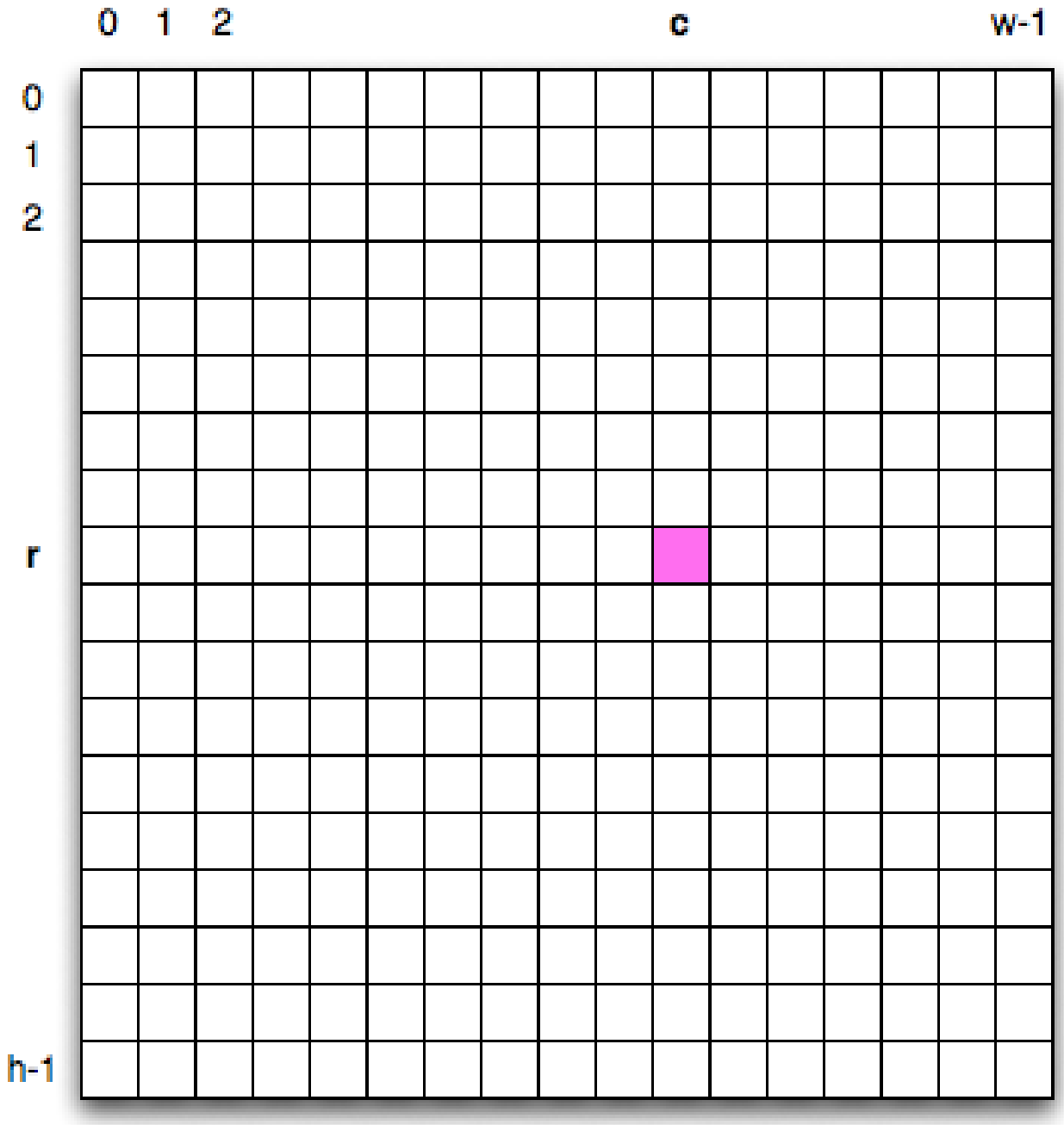
- Computer graphics primitives are basic geometric shapes or elements that serve as the building blocks for creating more complex images in computer graphics.
- Some common computer graphics primitives include:

- **Points:** These are single pixels that represent a location in space.
- **Lines:** Lines are sequences of connected pixels that extend in a particular direction. They can be defined by two endpoints or by a point and a direction vector.
- **Line Segments:** These are finite sections of lines that have a definite starting and ending point.

- **Polygons:** Polygons are closed geometric shapes with straight sides. They can be regular (all sides and angles are equal) or irregular (sides and angles can vary). Common examples include triangles, rectangles, and pentagons.
- **Circles:** Circles are round shapes defined by a center point and a radius. They can be used to represent curves and arcs.
- **Ellipses:** Similar to circles, ellipses are elongated round shapes defined by a center point, major axis, and minor axis.
- **Curves:** Curves represent smooth or nonlinear paths. Bezier curves and splines are examples of commonly used curves in computer graphics.
- **Surfaces:** Surfaces are two-dimensional representations of shapes. These can be used to create 3D models by combining multiple surfaces.

Basic Concepts

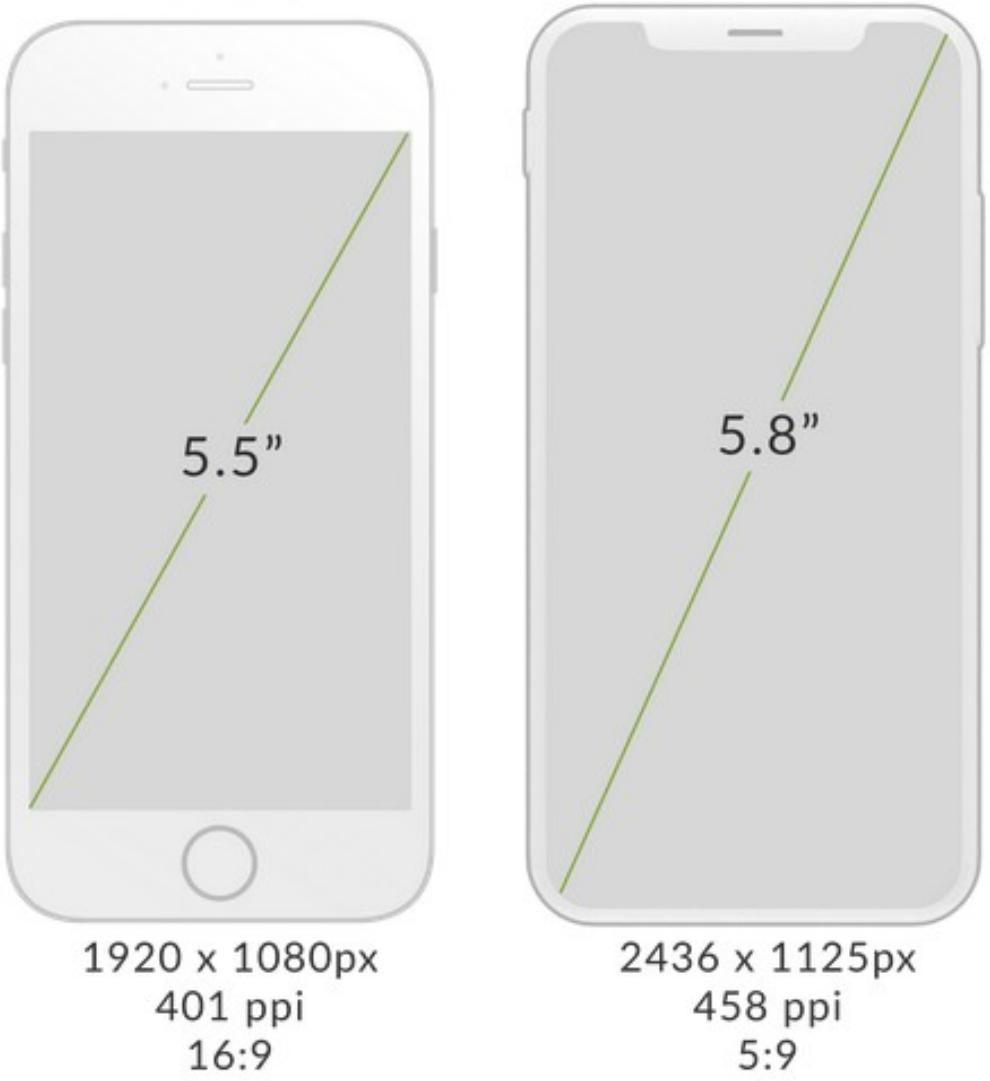
- **Screen Size :** The physical dimensions of a screen. It is the length, in inches, of the screen from one corner to the diagonal corner.
- **Pixel:** Screens display images through pixels. A pixel, pel or dots, or picture element is a physical point in a raster image, or the smallest addressable element in raster display device; so it is the smallest controllable element of a picture represented on the screen.
- Pixels are arranged in a grid to form images on screens, such as computer monitors, TVs, and mobile devices.
- Pixels are not always the same size from device to device.


0
1
2
c
w-1
0
1
2
r
h-1

- **Resolution:** This is the number of pixels displayed on the screen.
  - It is often formatted as width x height or pixels per inch.
  - Because pixels aren’t always the same size, it is possible to have two devices with the same screen size and different resolutions.
  - Resolution affects the clarity and level of detail in images.
  - A higher resolution image has more pixels and therefore can display finer details, but it may also require more processing power and storage space.

- **Aspect Ratio:** Aspect ratio measures width to height ratio of screen.

- For example, if a computer graphic has an aspect ratio of 3:1, this means the width of the graphic is three times of the height of the image.
- Common aspect ratios include 4:3 (standard for older TVs and monitors), 16:9 (widescreen HD), and 21:9 (ultrawide).
- The aspect ratio is significant because it affects how images are displayed on different screens. For instance, a 16:9 aspect ratio corresponds to a widescreen display, while a 4:3 aspect ratio is more square.



Screen sizes, resolutions, pixels, and aspect ratios

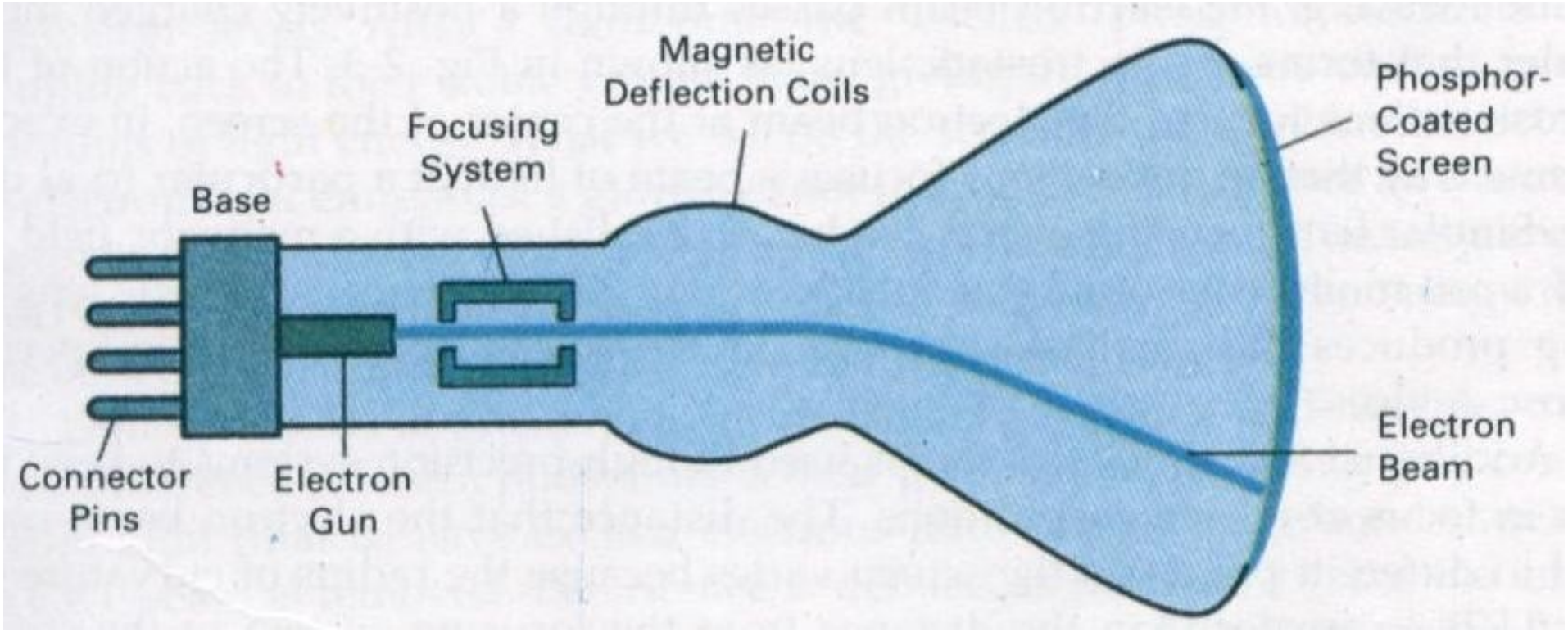
Frame Buffer

- A frame buffer (Refresh Buffer) is a portion of computer memory used to store and manage the data necessary for displaying images or video.
- The frame buffer stores pixel values for each position on the screen. Each pixel's value represents its color and intensity.
- The frame buffer essentially acts as a 2D array or grid, where each cell corresponds to a pixel on the display.

- The contents of the frame buffer are periodically read by the display hardware to illuminate the corresponding pixels on the physical display. This process happens rapidly, creating the illusion of continuous motion.
- The frame buffer's capacity (the number of pixels it can store) and bit depth (the number of bits used to represent the color of each pixel) directly impact the quality and complexity of the images that can be displayed. Higher resolutions and more colors require larger frame buffer capacities.

Display Devices

CRT

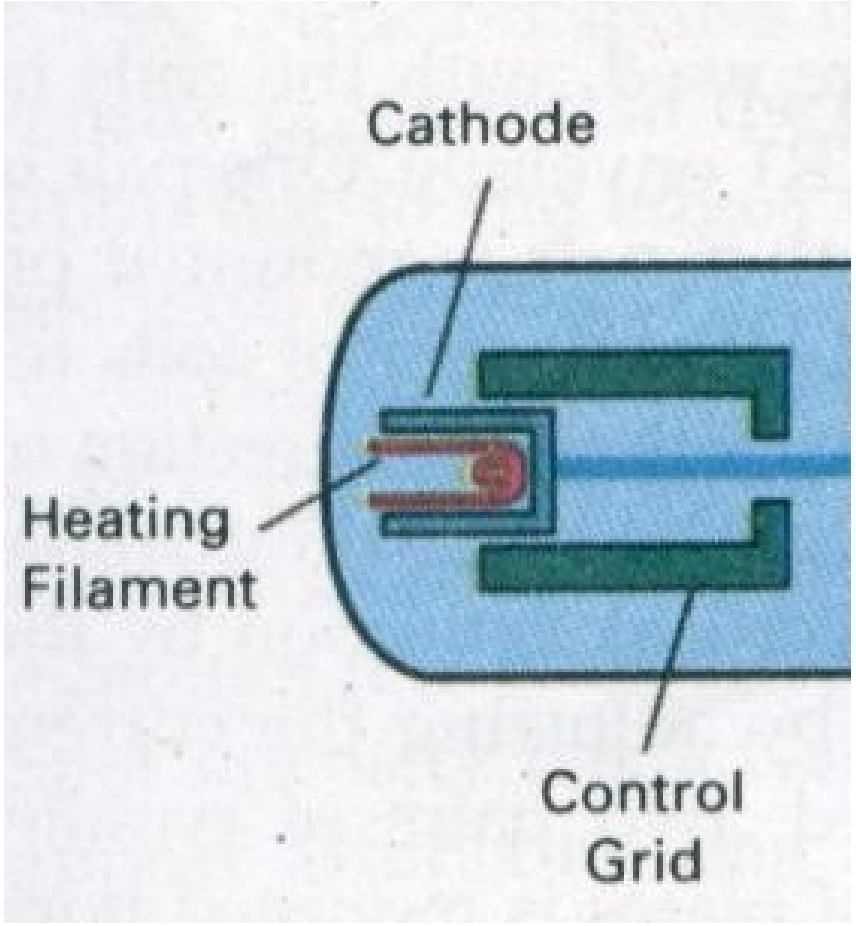

Magnetic
Deflection Coils
Phosphor-
Coated
Screen
Focusing
System
Base
Electron
Beam
Connector
Pins
Electron
Gun

- The electron gun emits a beam of electrons (**cathode rays**).

- The electron beam passes through focusing and deflection systems that direct it towards specified positions on the **phosphor-coated screen.**

- When the beam hits the screen, the phosphor emits a small spot of light at each position contacted by the electron beam.

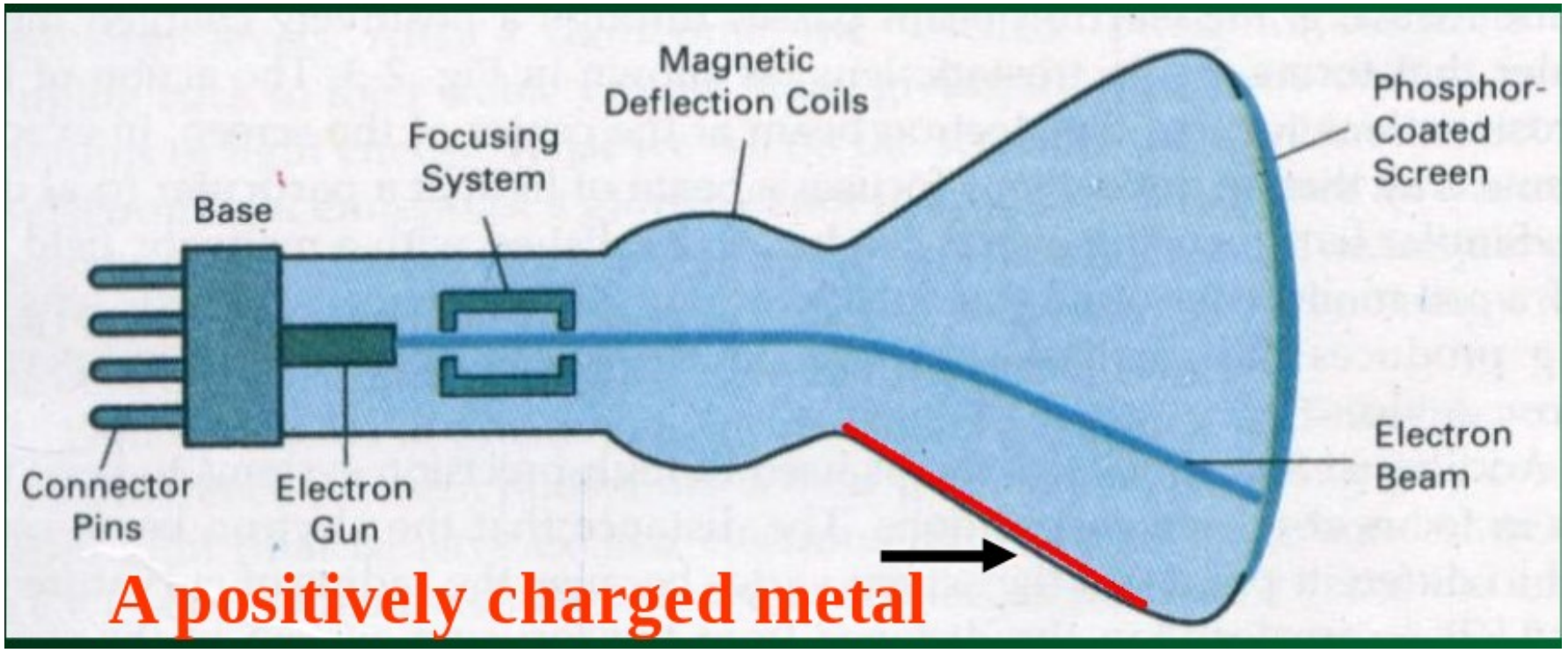
Electron Gun



- Heat is supplied to the cathode by the **filament**.
- The free electrons are then accelerated toward the phosphor coating by a **high positive voltage**.

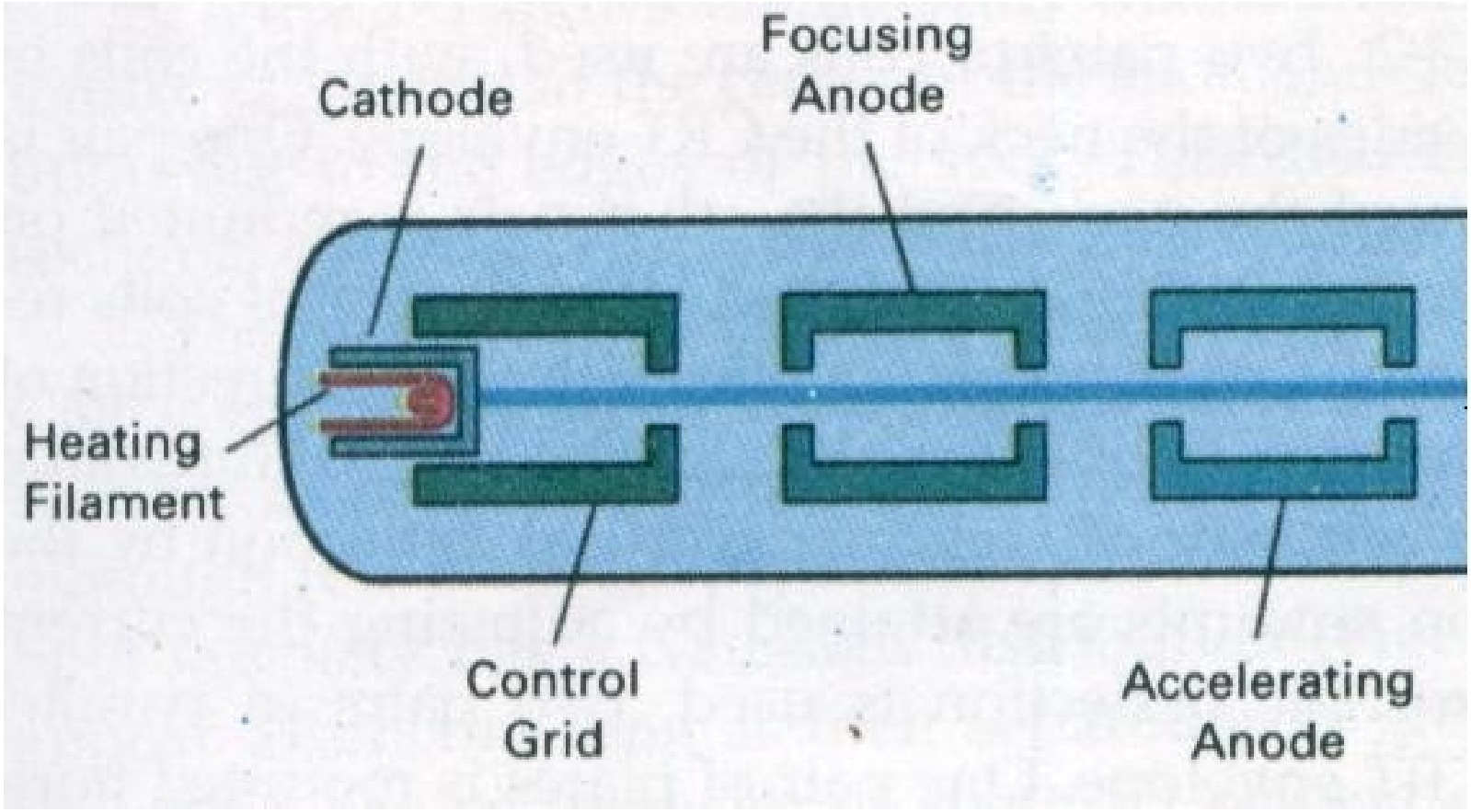
High positive voltage

A positively charged metal coating on the inside of the CRT envelope near the phosphor screen.



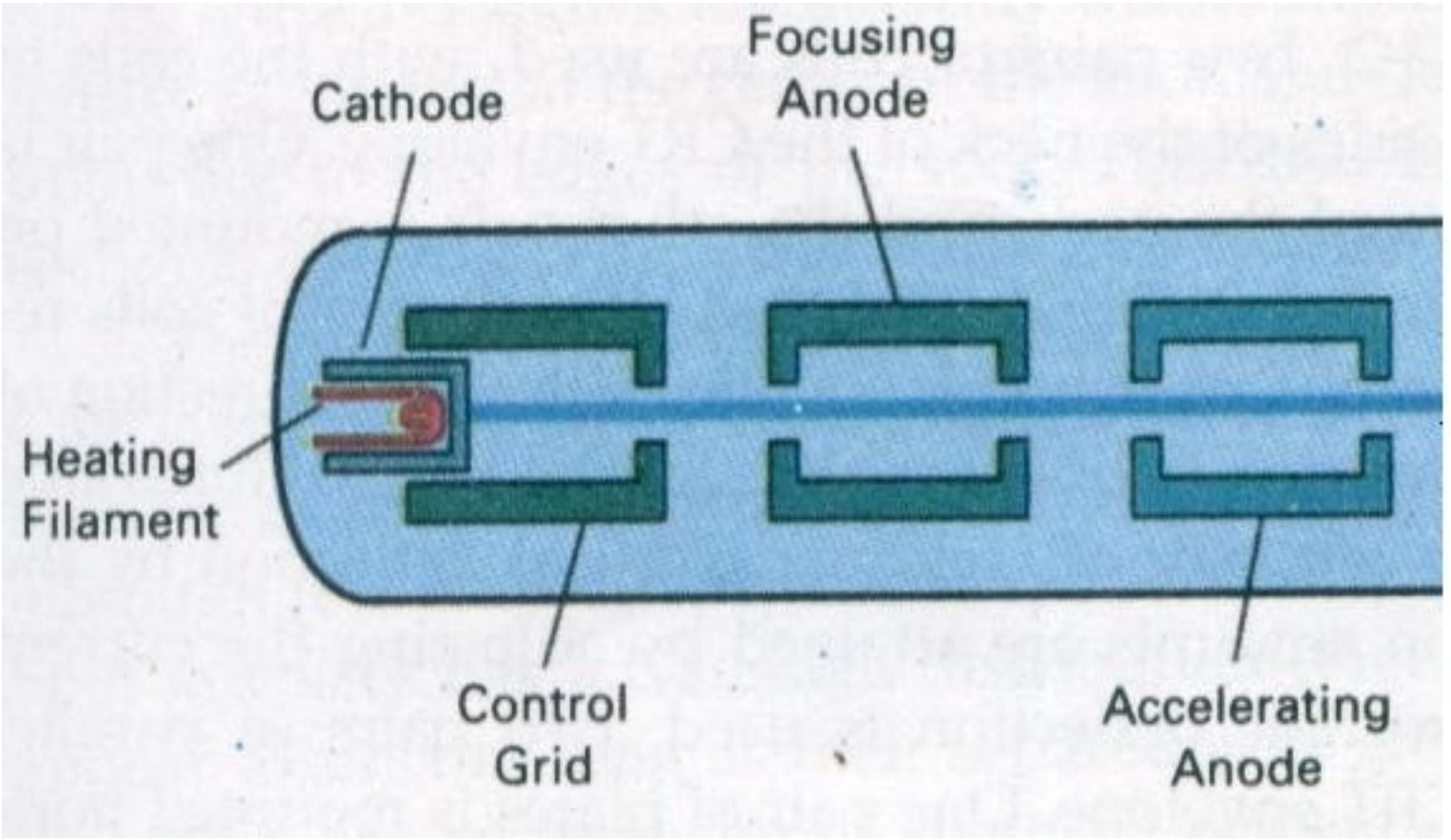
High positive voltage

Accelerating anode



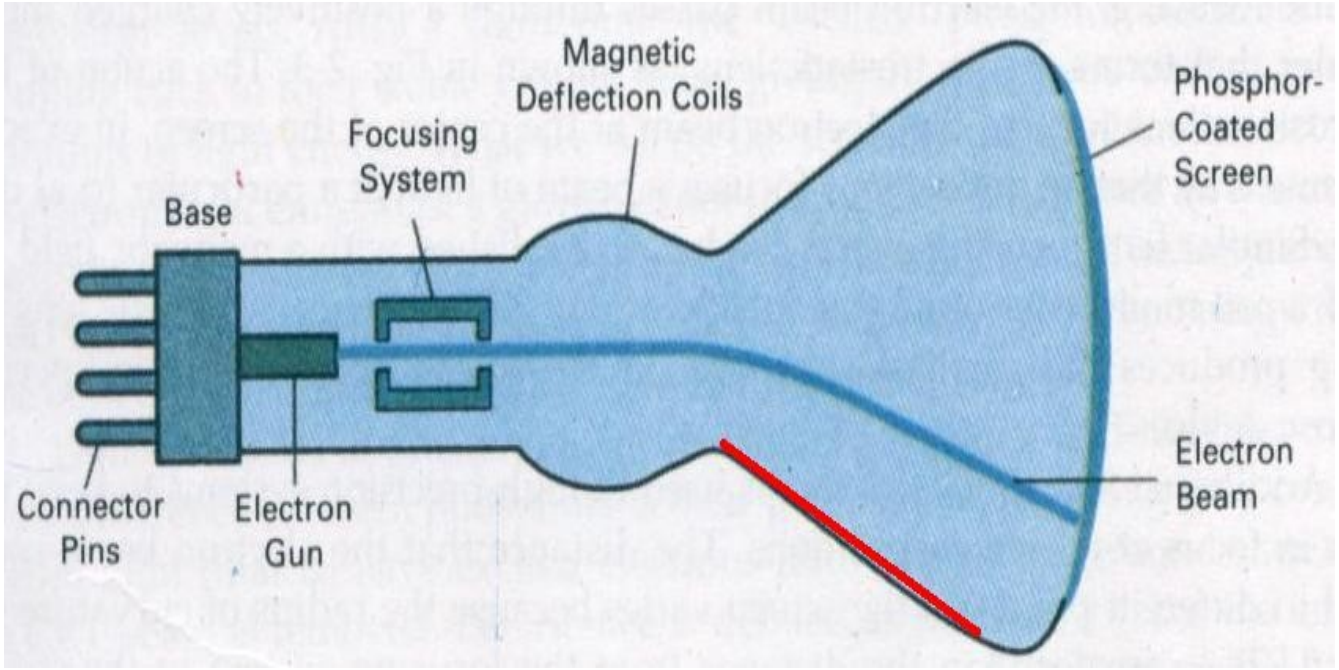
Control grid

Intensity of the electron beam is controlled by setting voltage level on the control grid.



Focusing system

The **focusing system** is needed to force the electron beam to converge into a small spot as it strikes the phosphor.



1) Electrostatic focusing is commonly used in computer graphics monitor.

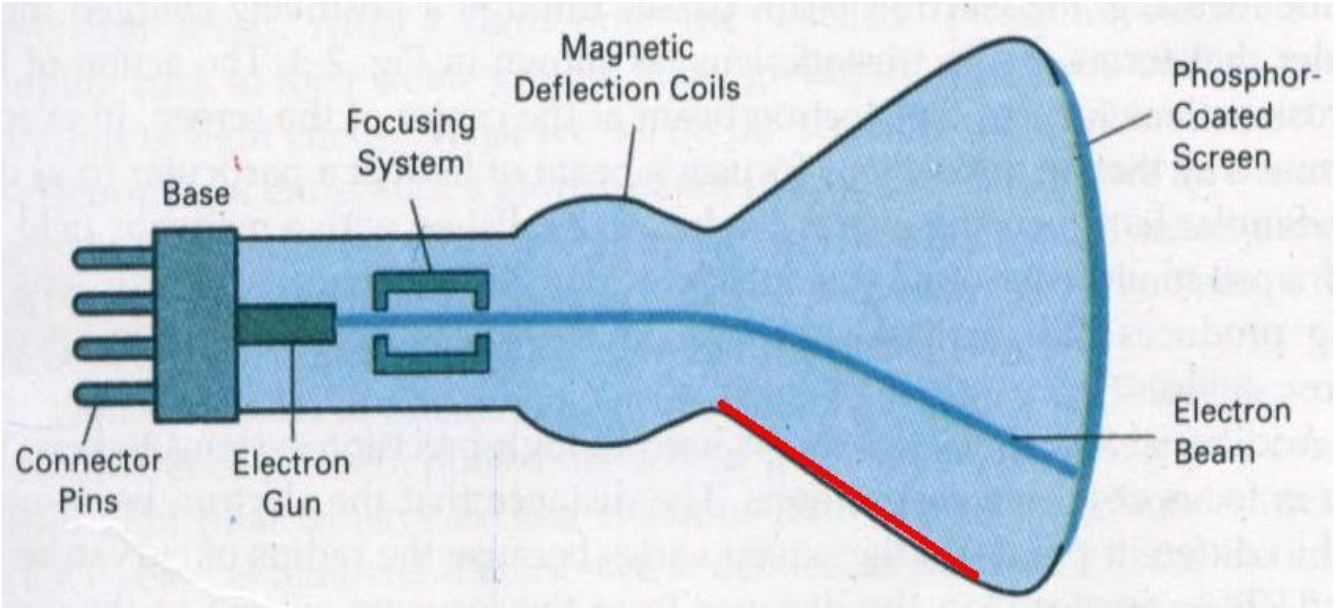
With electrostatic focusing, the electron beam passes through a positively charged metal cylinder that forms an **electrostatic lens**.

2) Similar lens focusing effects can be accomplished with a **magnetic field** set up by a coil mounted around the outside of the CRT envelope.

Deflection Systems

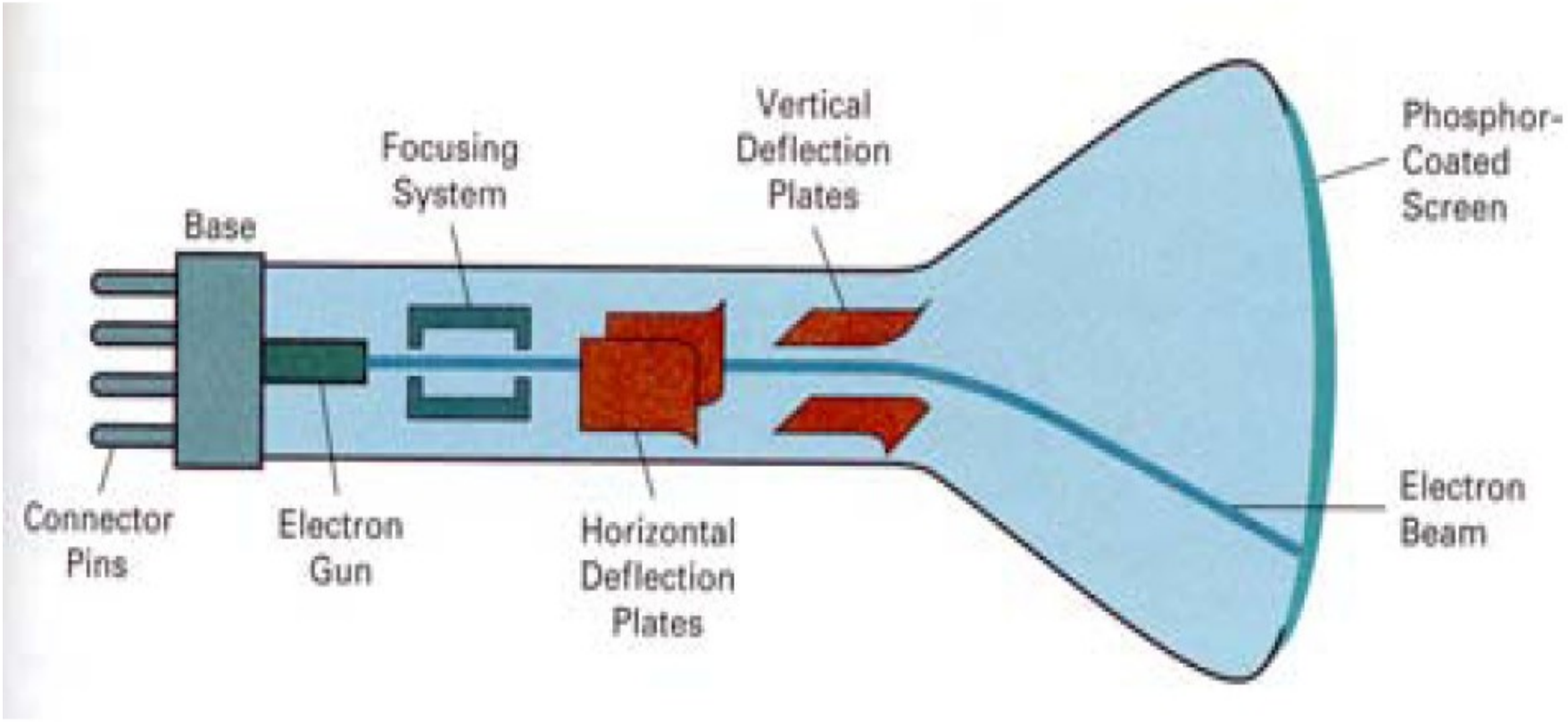
Deflection of the electron beam can be controlled either with **electric fields** or with **magnetic fields**.

1) The magnetic deflection coils mounted on the outside of the CRT envelope.



2) Electrostatic deflection: Two pairs of parallel plates are mounted inside the CRT envelope.

- One pair of plates is mounted horizontally to control the **vertical deflection**, and the other pair is mounted vertically to control **horizontal deflection**.
- The beam is deflected horizontally by applying an electric field between a pair of plates to its left and right, and vertically by applying an electric field to plates above and below.



Phoshor Coating

- Various phosphors are available depending upon the needs of the display application.
- The brightness, color, and persistence of the illumination depends upon the type of phosphor used on the CRT screen.
- Phosphors are available with persistences ranging from less than one microsecond to several seconds

3 types of displays:

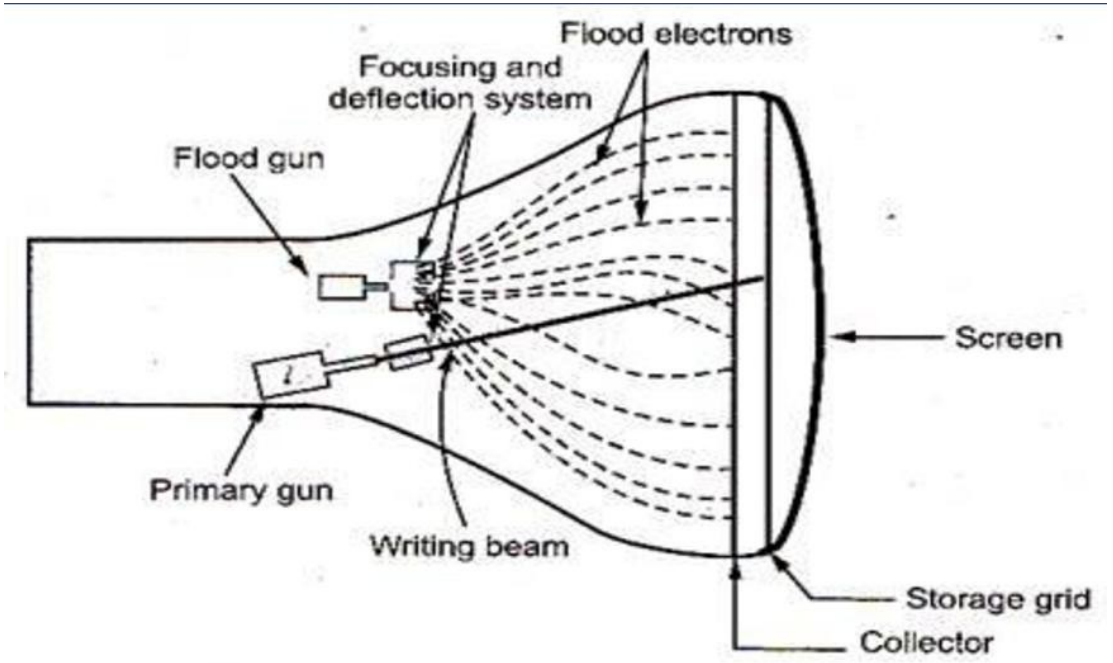
1. Direct View Storage Tube

2. Random Scan Displays

3. Raster Scan Displays

1. Direct View Storage Tube

- DVST is a CRT with **highly persistent phosphor.**
- A direct-view storage tube (DVST) stores the picture information as a charge distribution just behind the phosphor-coated screen.
- Two electron guns are used in a DVST. One, the primary gun, is used to store the picture pattern; the second, the flood gun, maintains the picture display as shown in the figure below.
- The term "storage grid" refers to the ability of the screen to retain the image which has been projected against it, thus avoiding the need to rewrite the image constantly.



Advantages

- No refreshing is needed.
- Very complex pictures can be displayed at very high resolution without flicker.

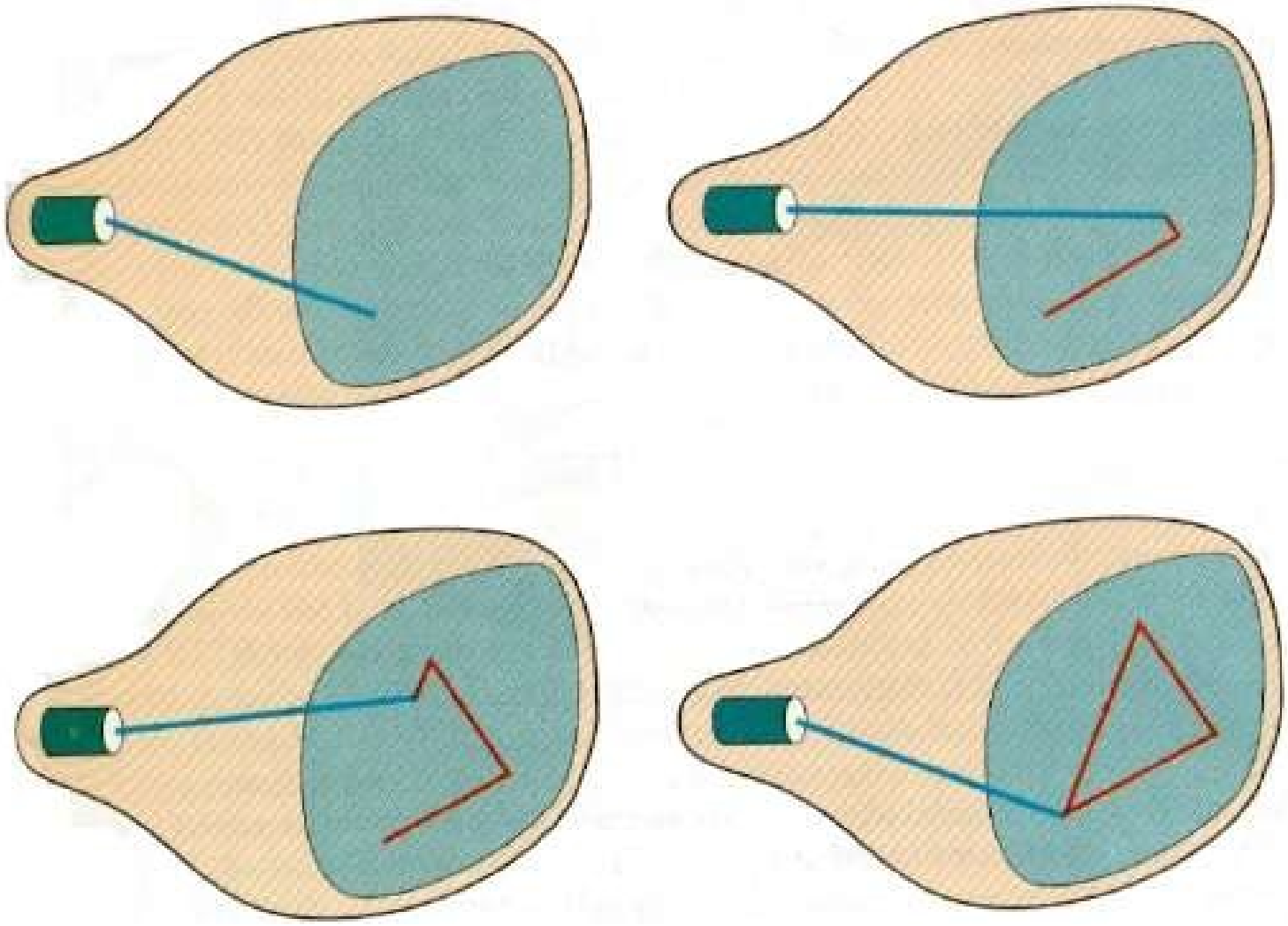
Disadvantages

- They ordinarily do not display color.
- Selected part of the picture can not be erased. Modifying any part of image requires redrawing of entire image.
- No animation in DVST. The erasing and redrawing process can take several seconds for complex pictures.

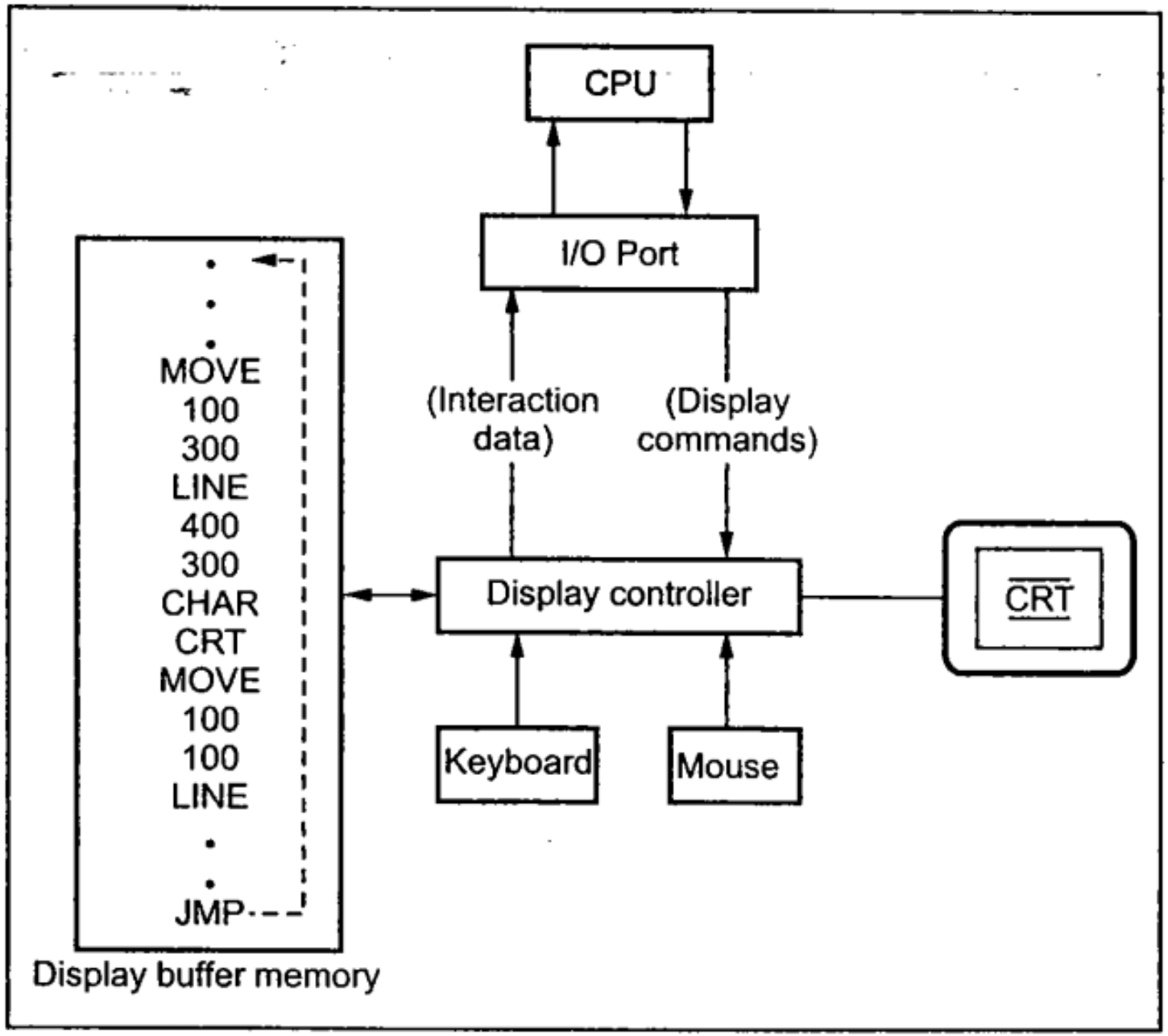
2. Random Scan Displays

- In a random scan display, a CRT has the electron beam directed only to the parts of the screen where a picture is to be drawn.

- Random scan monitors draw a picture one line at a time.
- Random scan display is also called as **Vector display, Stroke – writing or calligraphic displays.**



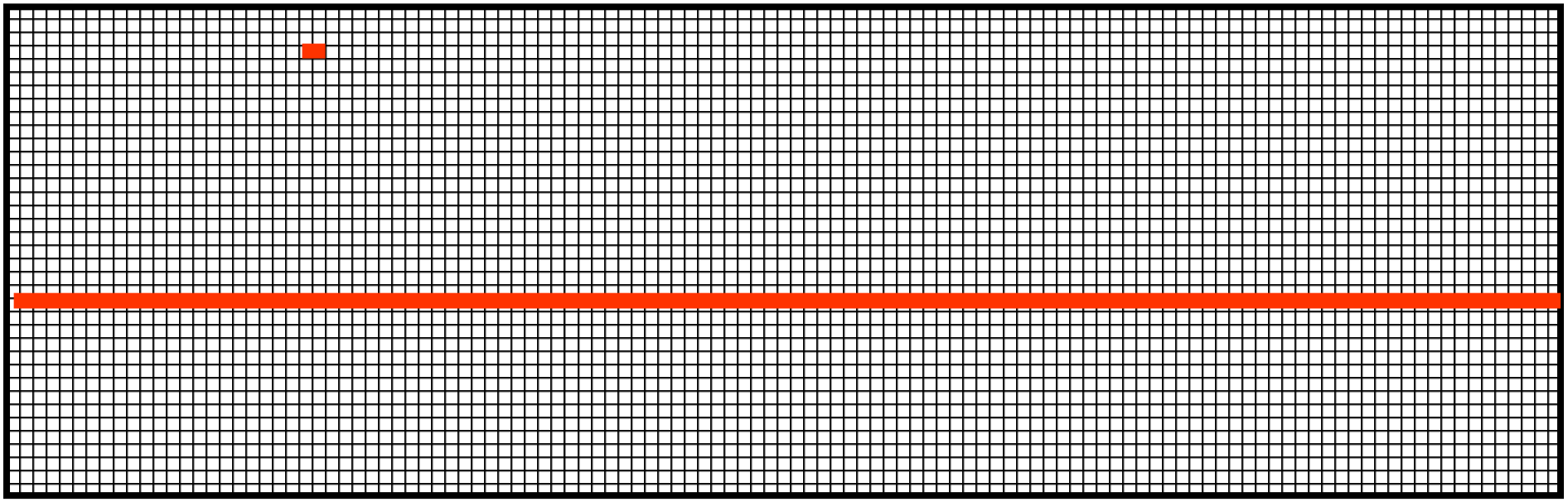
- Picture definition is stored as a set of line-drawing commands in an area of memory referred to as the **Display file (refresh display file / display buffer memory)**
- To display a specified picture, the system cycles through the set of commands in the display buffer memory, drawing each component line in turn.
- After all the line-drawing commands are processed, the system cycles back to the first line command in the list.
- Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second.


CPU
I/O Port
(Interaction data)
(Display commands)
MOVE
100
300
LINE
400
300
CHAR
CRT
MOVE
100
100
LINE
JMP
Display buffer memory
Display controller
CRT
Keyboard
Mouse

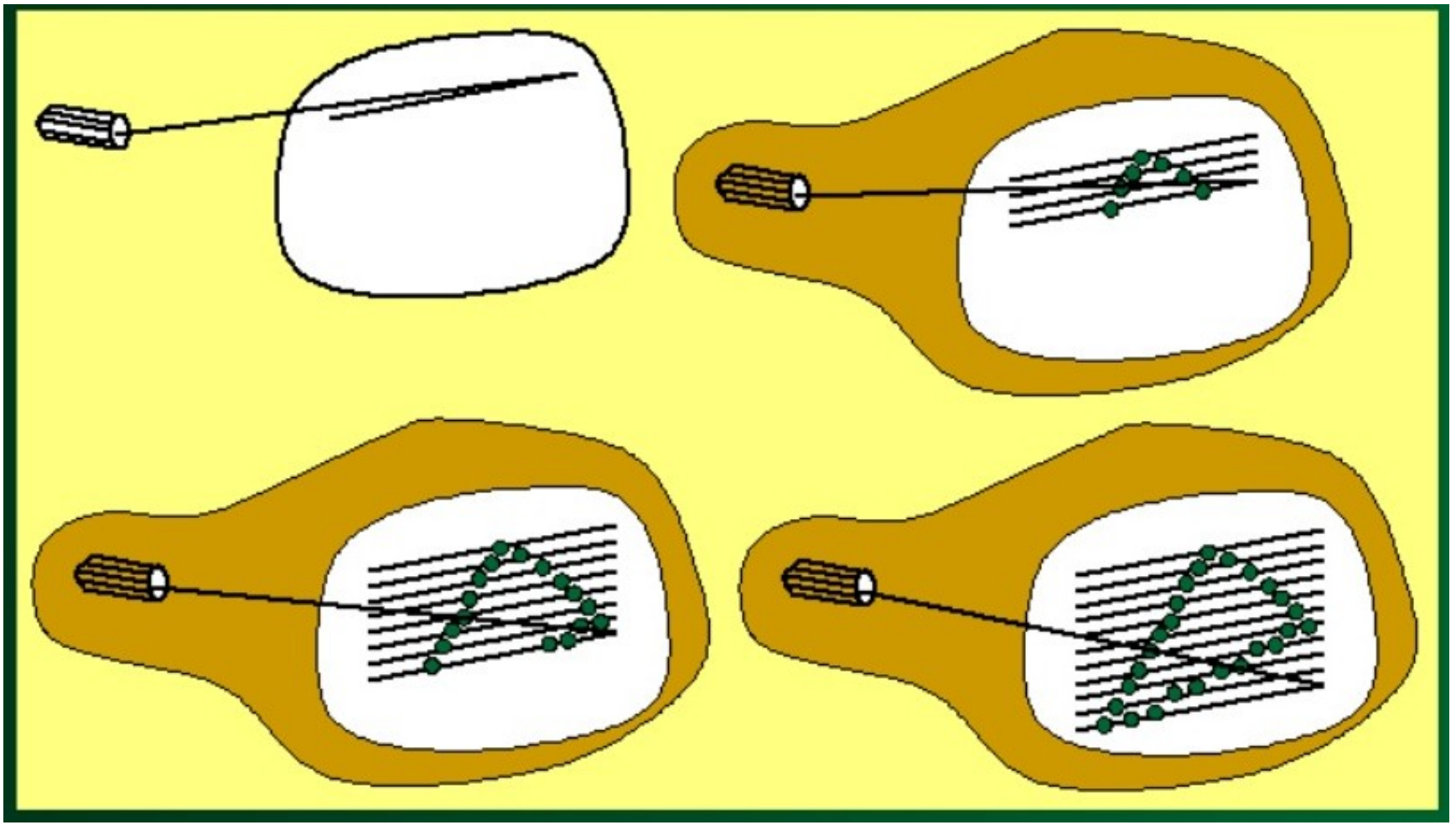
- Random scan displays have higher resolution than raster systems.
- Random displays produce smooth line drawing while a raster system produces jagged lines.
- Random scan displays are designed for line-drawing applications and can not display realistic shaded scenes.

3. Raster Scan displays

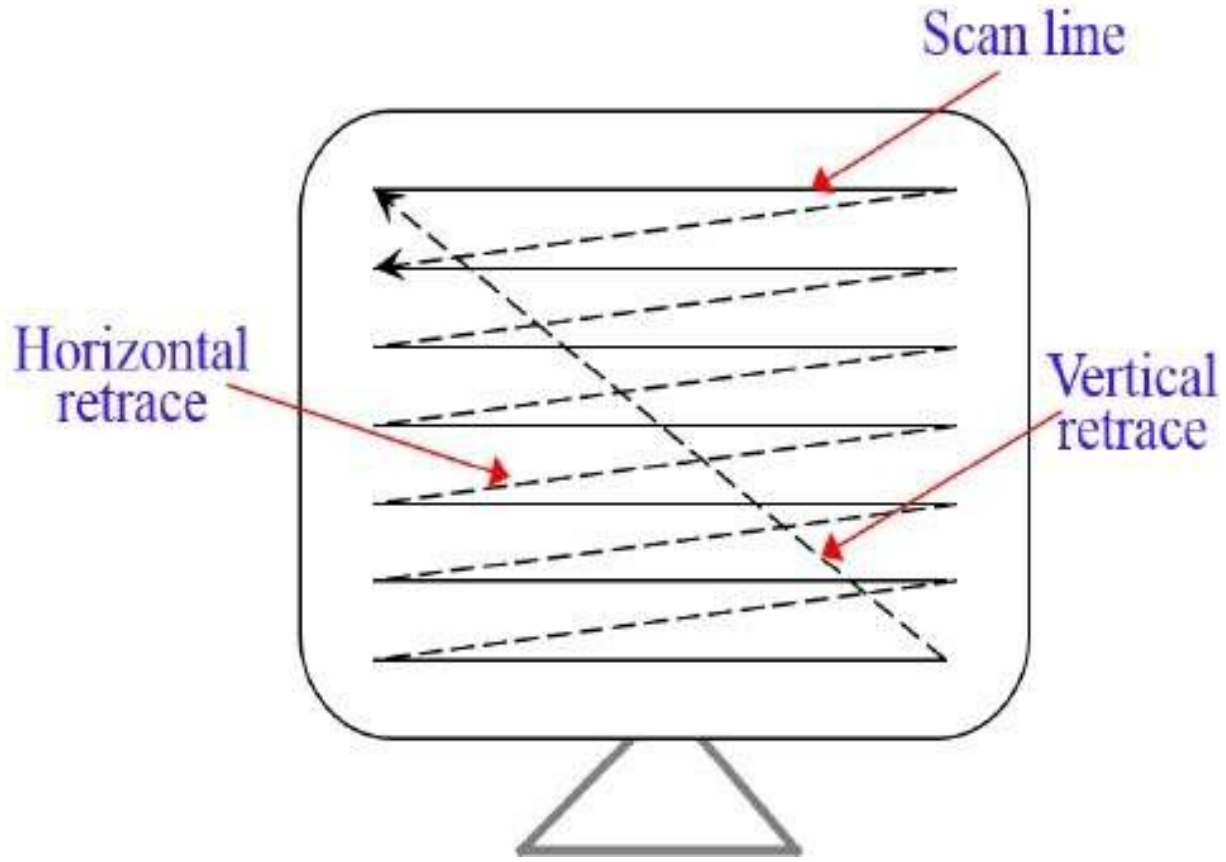
- **Raster:** A rectangular array of points or dots
- **Pixel:** One dot or picture element of the raster
- **Scan Line:** A row of pixels
- Raster Scan is the representation of images as a collection of **pixels**.



- In a raster scan system, the electron beam is swept across the screen, one row at a time from top to bottom.
- As the electron beam moves across each row, the beam intensity is turned on and off to create a pattern of illuminated spots.
- Picture definition is stored in memory area called the **Refresh Buffer or Frame Buffer.** This memory area holds the set of intensity values for all the screen points.
- Stored intensity values are then retrieved from the refresh buffer and “painted” on the screen one row (scan line) at a time.



- Each screen point is referred to as a **pixel (picture element) or pel.**
- At the end of each scan line, the electron beam returns to the left side of the screen to begin displaying the next scan line.



- A black-and-white system: each screen point is either on or off, so only one bit per pixel is needed to control the intensity of screen positions.

  On a black-and-white system with one bit per pixel, the frame buffer is called **bitmap.**

- For system with multiple bits per pixel, the frame buffer is called **pixmap.**

- A raster system produces jagged lines.
- A raster system requires larger file size as compared to random display system.

Application of Computer Graphics

- **Entertainment and Media:**
  - Video Games
  - Movies and Animation
  - Virtual Reality (VR) and Augmented Reality (AR)

- **Design and Visualization:**
  - Graphic Design: Computer graphics are used in designing logos, posters, brochures, and other marketing materials.
  - Architectural Visualization: Architects use computer graphics to create detailed 3D models and visualizations of buildings and interiors.
  - Product Design: Graphics aid in creating and visualizing product prototypes and designs.

- **Engineering and Manufacturing:**

- CAD (Computer-Aided Design): Engineers use computer graphics for designing and modeling complex machinery and structures.
- Simulation: Graphics play a role in simulating real-world scenarios and testing designs before physical implementation.

- **Medical Imaging and Healthcare:**

- Medical Visualization: Graphics are used to visualize medical data from imaging technologies like MRI, CT scans, and X-rays, assisting in diagnosis and treatment planning.
- Surgical Simulation: Graphics-based simulations help train surgeons and practice complex procedures in a virtual environment.

- **Education and Training:**

- Educational Software: Computer graphics aid learning by providing interactive simulations, virtual labs, and visual explanations.
- Training Simulations: Graphics-based simulations are used in fields like aviation, military, and healthcare for training purposes.

- **Scientific Visualization:**

- Data Visualization: Graphics help researchers and scientists visualize complex data sets and patterns, aiding in analysis and discovery.
- Astrophysics and Molecular Modeling: Graphics are used to visualize large-scale astrophysical phenomena and molecular structures.

- **Advertising and Marketing:**

  Digital Advertising: Graphics enhance online advertising with visually engaging content.

- **Digital Art:**

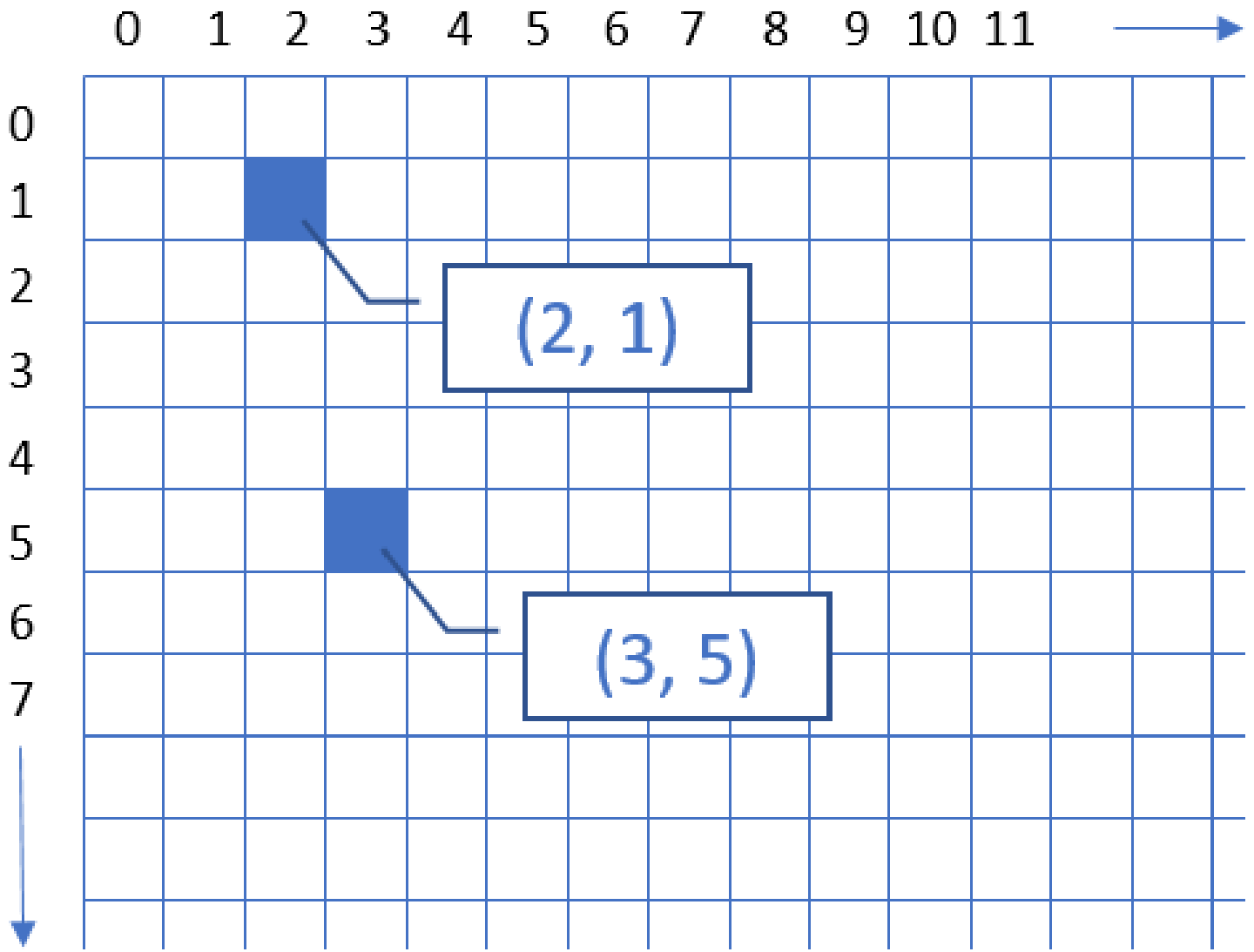
  Artists create digital paintings, illustrations, and multimedia art using computer graphics tools.

- **Geographical Information Systems (GIS):**

  Mapping and GIS: Computer graphics are used to create digital maps, visualize geographic data, and analyze spatial relationships.

Scan Conversion

- The process of representing continuous graphics object as a collection of discrete pixels is called **Scan Conversion.**
- In simple words, scan conversion is to figure out which pixels to fill in order to generate picture on a screen. While **shading** determines a color for each filled pixel.
- It is the responsibility of graphics system or the application program to convert each primitive from its geometric definition into a set of pixels. This conversion task is generally referred to as a **scan conversion** or **rasterization.**


0 1 2 3 4 5 6 7 8 9 10 11
0
1
2
3
4
5
6
7
(2, 1)
(3, 5)

DDA Line Drawing algorithm

1. Take end points of the line (x1,y1) & (x2,y2)

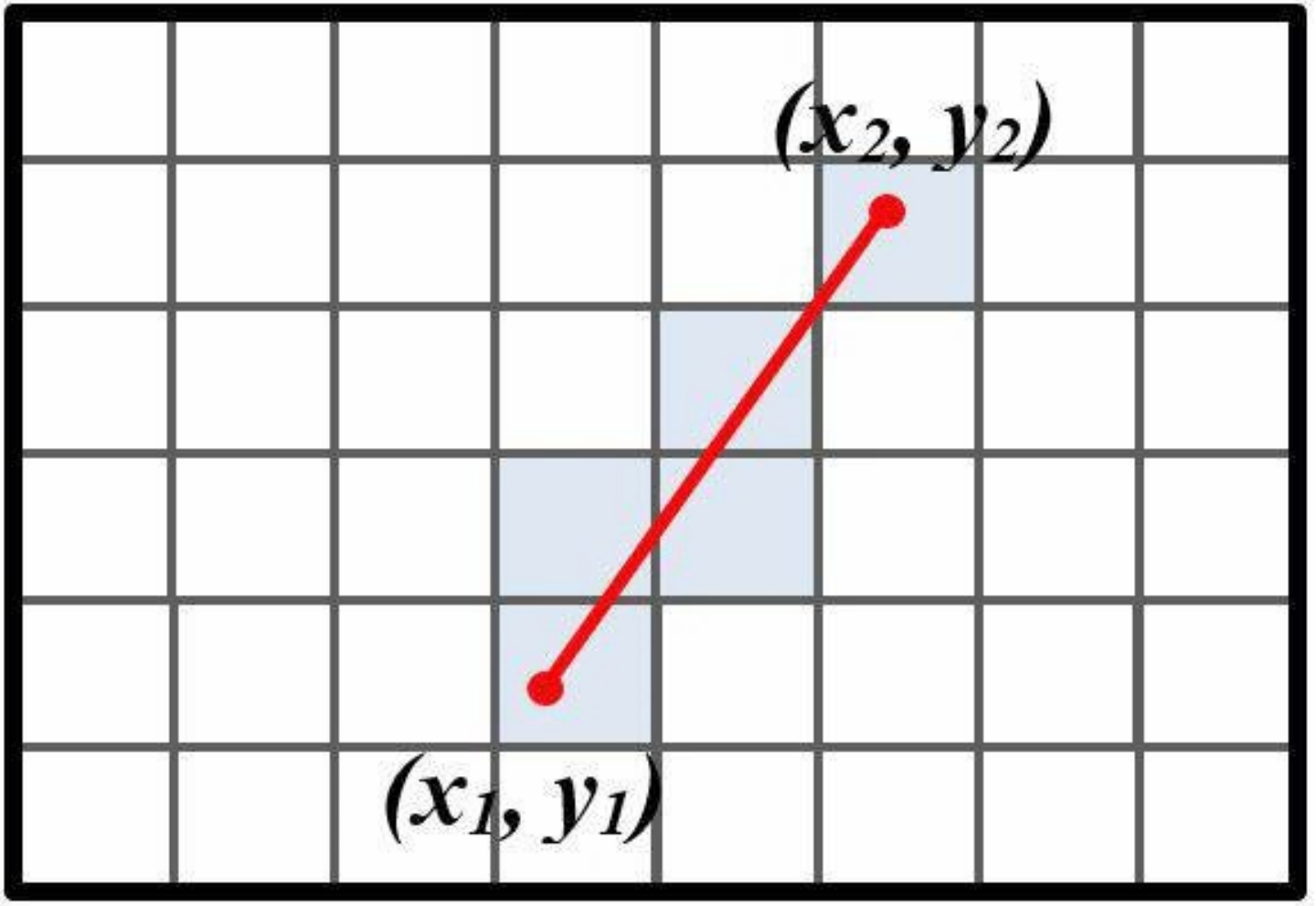
2. Compute: dx = x2-x1;
dy = y2-y1;

```
if (abs(dx) >= abs(dy))
        len=abs(dx);
else
        len=abs(dy);
```

3. xin = dx / len;
yin = dy / len;

4. x = x1;
y = y1;

```
int i=0;
while (i<=len)
        {
          putpixel (Round(x),Round(y),RED);
                x=x+xin;
                y=y+yin;
           i++;
        }
```


(x_2, y_2)
(x_1, y_1)

Bresenham Line Drawing algorithm

- Take end points from user: **(x1,y1)** and **(x2,y2)**
- Initialize variables: x=x1; y=y1;
- dx=abs(x2-x1); dy=abs(y2-y1);
- s1=Sign(x2-x1); s2=Sign(y2-y1);
- Interchange dx and dy depending on slope of the line

  If dy>dx then

  temp=dx

  dx=dy

  dy=temp

  Interchange=1

  else

  Interchange=0
- **e = 2 *dy – dx** //initialise error term

```
main loop
for i=0 to dx
        setpixel(x,y);
        while (e>0)
                if Interchange=1 then
                    x=x+s1;
                else
                    y=y+s2;
                end if
                e = e-2*dx
        end while
        if Interchange=1 then
                y=y+s2;
        else
                x=x+s1;
        end if
        e = e+2*dy
    next i;
end for
```

Bresenham Circle Drawing algorithm

1) Read the x and y coordinates of center: **(centx, centy)**

2) Read the radius of circle: **(r)**

3) Initialize,

x = 0;

y = r;

4) Initialize decision parameter: **p = 3 – (2*r)**

5) do {

setpixel(x,y);

If (p<0)

{

p = p+(4*x)+6;

}

else {

p=p+[4*(x-y)]+10;

y=y-1;

}

x=x+1;

} while(x<y)

```
putpixel(centx+x, centy-y, 4);
putpixel(centx-x, centy-y, 4);
putpixel(centx+x, centy+y, 4);
putpixel(centx-x, centy+y, 4);
putpixel(centx+y, centy+x, 4);
putpixel(centx+y, centy-x, 4);
putpixel(centx-y, centy+x, 4);
putpixel(centx-y, centy-x, 4);
```

Midpoint Circle Drawing algorithm

1) Read the x and y coordinates of center: **(centx, centy)**

2) Read the radius of circle: **(r)**

3) Initialize,

```
x = 0;
y = r;
```

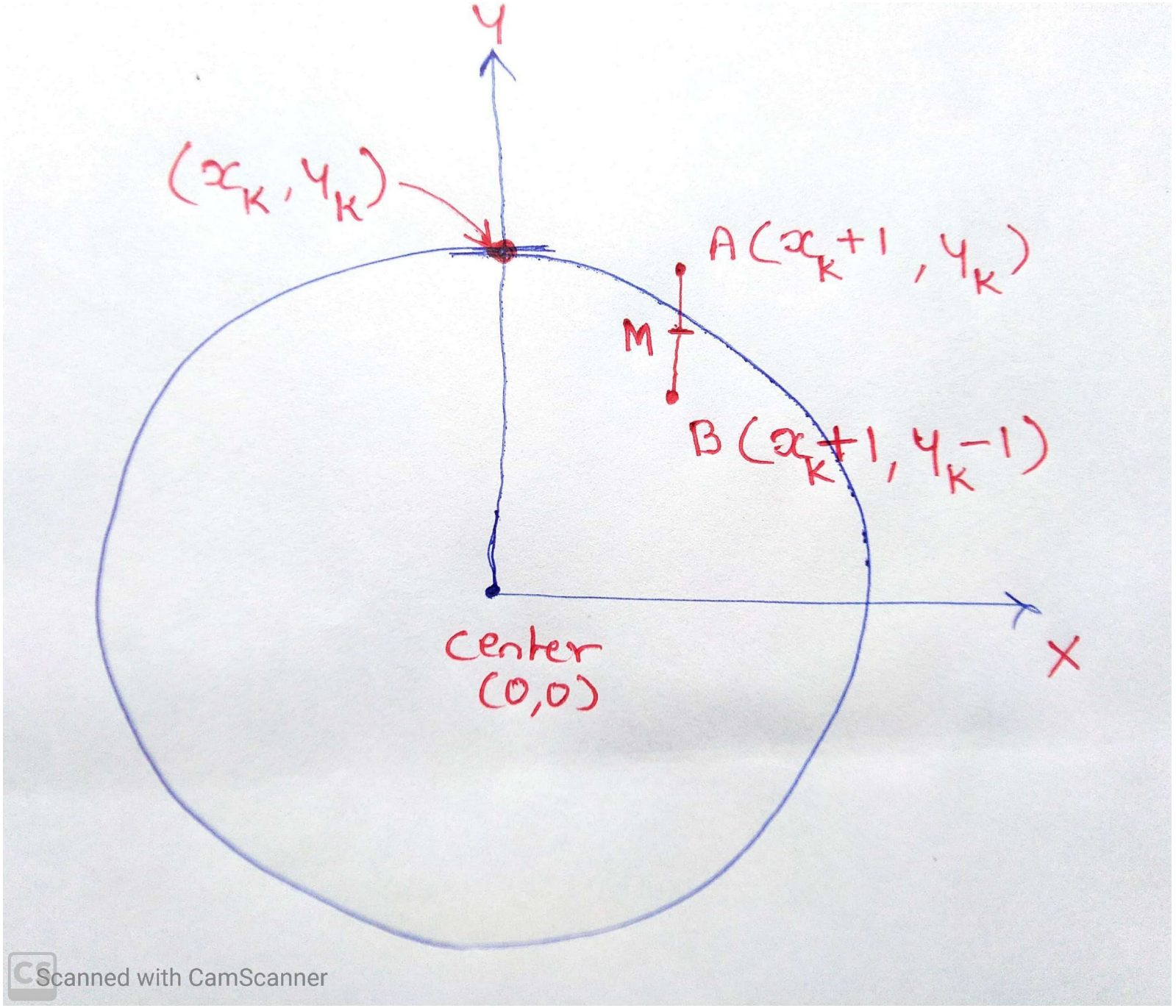
4) Initialize decision parameter: **p = 1 – r;**

5) do {

```
setpixel(x,y);
    If (p<0)
    {
        p = p + 2*x + 3;
    }
    else {
        p=p + 2*x – 2*y + 5;
        y=y-1;
    }
x=x+1;
```

} while(x<y)

```
putpixel(centx+x, centy-y, 4);
putpixel(centx-x, centy-y, 4);
putpixel(centx+x, centy+y, 4);
putpixel(centx-x, centy+y, 4);
putpixel(centx+y, centy+x, 4);
putpixel(centx+y, centy-x, 4);
putpixel(centx-y, centy+x, 4);
putpixel(centx-y, centy-x, 4);
```



- Consider point (x_k, y_k) is on circle
- In octant 1, we have to decide next pixel to be printed i.e. either pixel A or Pixel B.
- Consider M is the midpoint of A & B. Coordinates of M will be,

$$M\left(\frac{(x_k+1)+(x_k+1)}{2}, \frac{(y_k)+(y_k-1)}{2}\right)$$

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

x_k+1: x coordinate; $y_k-\frac{1}{2}$: y coordinate

We have eqn of circle, at origin,

$$x^2 + y^2 = r^2$$

$$d_k = x^2 + y^2 - r^2 \quad \text{(decision parameter)}$$

if (1) $d_k = 0$: point lies on circle

(2) $d_k < 0$: point lies inside circle

(3) $d_k > 0$: point lies outside circle

- If we put x & y coordinates of midpoint in above eqn, we can check whether midpoint is on/inside/outside the circle.
- If midpoint is inside the circle, ~~point~~ means point A is more close to actual circle than point B. Hence we select point A as a next point after (x_k, y_k)
- If midpoint is outside the circle, means point B is more close to actual circle than A. Hence we select point B after (x_k, y_k) as next point

Now, put coordinates of of midpoint in eqn (I)

$$d_k = (x_k+1)^2 + (y_k-\tfrac{1}{2})^2 - r^2 \quad \text{---(III)}$$

Now, next decision parameter d_{k+1} is calculated by replacing k by k+1 ;

$$d_{k+1} = (x_{k+1}+1)^2 + (y_{k+1}-\tfrac{1}{2})^2 - r^2 \quad \text{---(IV)}$$

Now, calculate $(d_{k+1} - d_k)$

$$d_{k+1} - d_k = \left[(x_{k+1}+1)^2 + (y_{k+1}-\tfrac{1}{2})^2 - r^2\right] - \left[(x_k+1)^2 + (y_k-\tfrac{1}{2})^2 - r^2\right]$$

$$d_{k+1} - d_k = (x_{k+1}+1)^2 + (y_{k+1}-\tfrac{1}{2})^2 - (x_k+1)^2 - (y_k-\tfrac{1}{2})^2 \quad \text{---(V)}$$

PAGE :
DATE : / /

(1) **Case - 1** $d_k < 0$

So now in eqn (V) consider case - 1

i.e. $d_k < 0$ $\Rightarrow x_{k+1} = x_k + 1$

& $y_{k+1} = y_k$

Point A is selected

} Put these values in eqn (V)

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

$$= x_k^2 + 4x_k + 4 - x_k^2 - 2x_k - 1$$

$$d_{k+1} - d_k = 2x_k + 3$$

$$\therefore \boxed{d_{k+1} = d_k + 2x_k + 3}$$

(2) **Case-2 :→ $d_k > 0$**

for $d_k > 0$ ⇒ Point B is selected

$$x_{k+1} = x_k + 1$$

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

} Put these values in eqn (V)

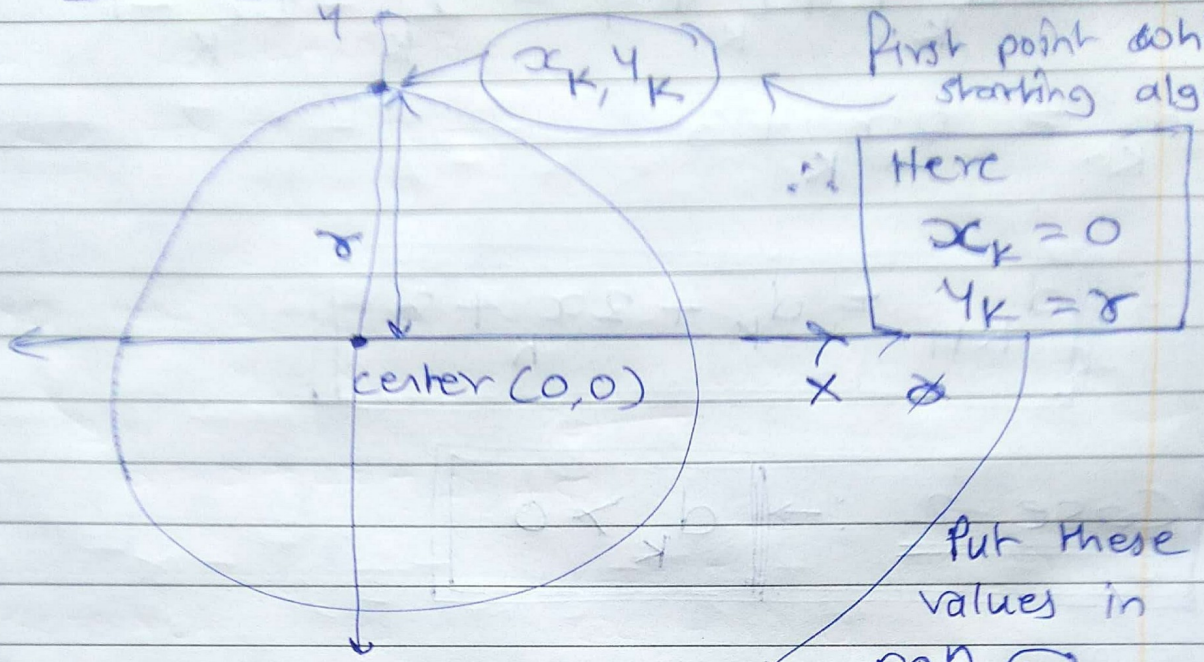
$$d_{k+1} - d_k = \left(x_k + 2\right)^2 + \left(y_k - \frac{3}{2}\right)^2 - \left(x_k + 1\right)^2 - \left(y_k - \frac{1}{2}\right)^2$$

$$= x_k^2 + 4x_k + 4 + y_k^2 - 3y_k + \frac{9}{4} - x_k^2 - 2x_k - 1 - y_k^2 + y_k - \frac{1}{4}$$

$$d_{k+1} - d_k = 2x_k - 2y_k + 5$$

$$\therefore \boxed{d_{k+1} = d_k + 2x_k - 2y_k + 5}$$

Now initial decision parameter (d_0) :



$$d_0 = (0+1)^2 + \left(r - \frac{1}{2}\right)^2 - r^2$$

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

approximate "1"

$$\boxed{d_0 = 1 - r}$$

DDA Circle Drawing algorithm

1) Read the x and y coordinates of center: **(centx, centy)**

2) Read the radius of circle: **(r)**

3) Initialize,

x = 0;

y = r;

We have, $2^{(n-1)} <= r <= 2^{n}$

$\varepsilon = 2^{-n}$;

4) do {

setpixel(x,y);

x = x + ε*y;

y = y – ε*x;

} while(x<y)

```
putpixel(centx+x, centy-y, 4);
putpixel(centx-x, centy-y, 4);
putpixel(centx+x, centy+y, 4);
putpixel(centx-x, centy+y, 4);
putpixel(centx+y, centy+x, 4);
putpixel(centx+y, centy-x, 4);
putpixel(centx-y, centy+x, 4);
putpixel(centx-y, centy-x, 4);
```

Line styles: Solid, dotted, dashed and thick lines

DDA Line drawing (Solid line)

```
dx=x2-x1;     dy=y2-y1;
    if(abs(dx)>=abs(dy))
                len=abs(dx);
   else
                len=abs(dy);
    xin=dx/len;     yin=dy/len;
   x=x1;    y=y1;
   int gd=DETECT,gm;
   initgraph(&gd,&gm,NULL);
   x=x+0.5;        y=y+0.5:
   putpixel(x,y,9);
   int i=1;
     while(i<=len)
     {
       putpixel(x,y,9);
       x=x+xin;
       y=y+yin;
       i++;
      }
```

Dotted line

```
while(i<=len)
        {
          if(i%5==0)
                {
                   putpixel(x,y,9);
                }
           x=x+xin;
           y=y+yin;
           i++;
         }
```

Dashed line

```
while(i<=len)
        {
          if(i%8<4)
                {
                   putpixel(x,y,9);
                }
          x=x+xin;
          y=y+yin;
          i++;
        }
```

Thick line

```
while(i<=len)
        {
            for(j=0;j<t;j++)      // where t is the thickness of line
            {
                putpixel(x,y+j,9);
            }
          x=x+xin;
          y=y+yin;
         i++
        }
```

Dashed-Dotted line

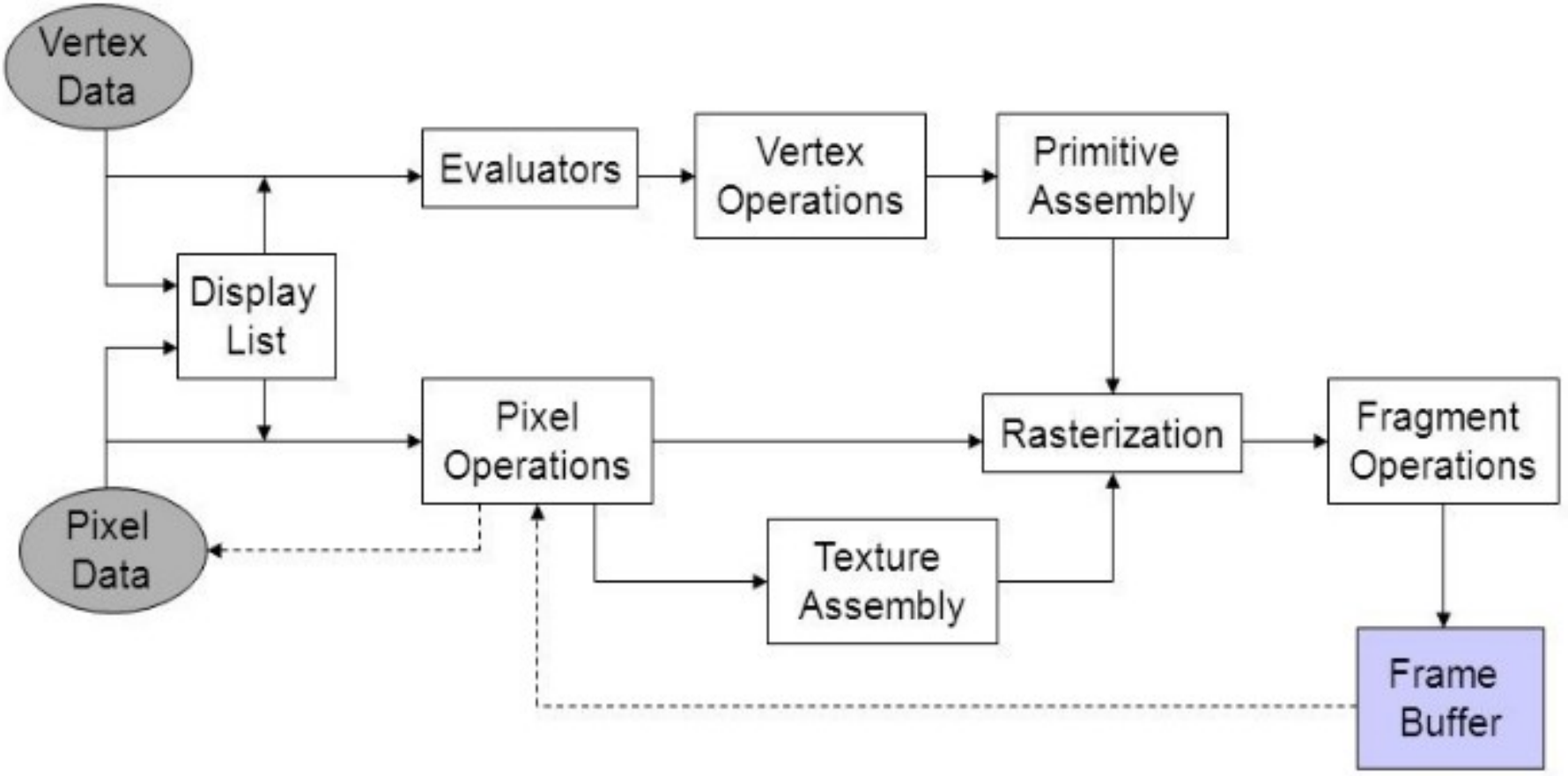
```
while(i<=len)
        {
          if(i%9<4 || i%9==6)
                {
                   putpixel(x,y,9);
                }
          x=x+xin;
          y=y+yin;
          i++;
        }
```

OpenGL

- **Open Graphics Library (OpenGL)** is a cross-language (language independent), cross-platform (platform-independent) API for rendering 2D and 3D Vector Graphics(use of polygons to represent image).

- It provides a set of functions and procedures for interacting with a computer's GPU (Graphics Processing Unit) to generate real-time graphics for applications such as video games, simulations, CAD software, and more.

OpenGL Architecture (Rendering Pipeline)



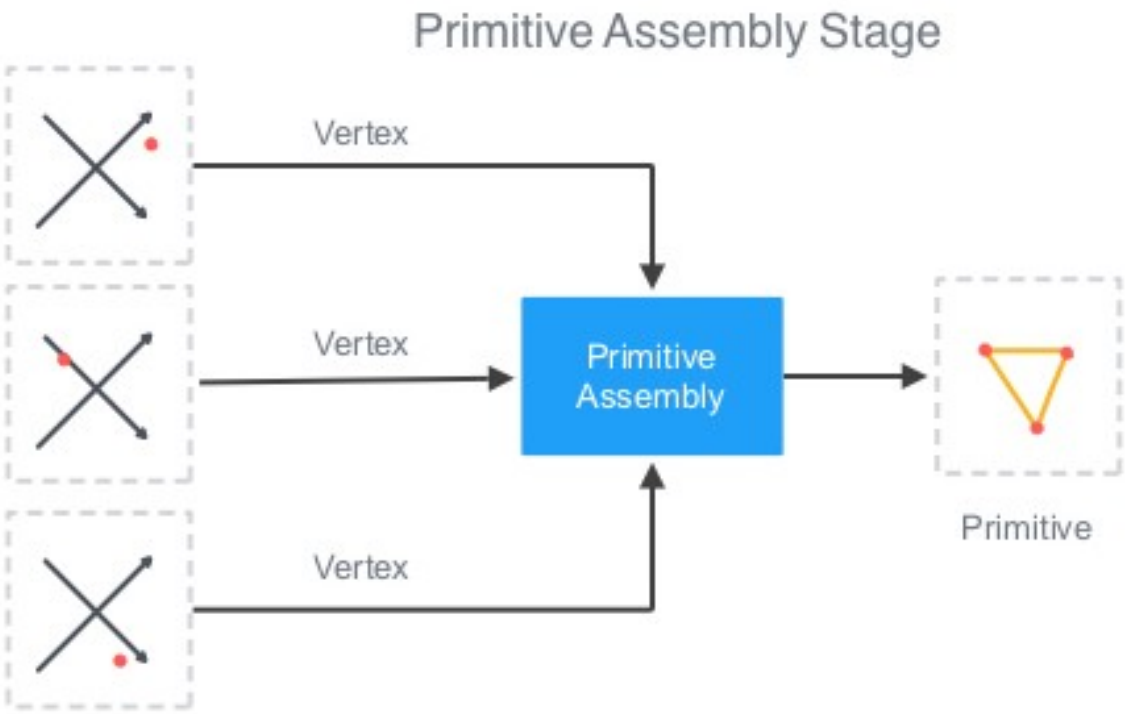
- **Geometric data** (vertices, lines, and polygons) follow the path that includes evaluators and per-vertex operations, while **pixel data** (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

- **Display Lists:** All data, whether it describes geometry or pixels, can be saved in a display list for current or later use.

- **Evaluators:** It convert curves and surfaces that are described by parametric equations into vertex data. Parametric curves and surfaces may be initially described by control points. Evaluators provide a method to derive the vertices used to represent the surface from the control points.

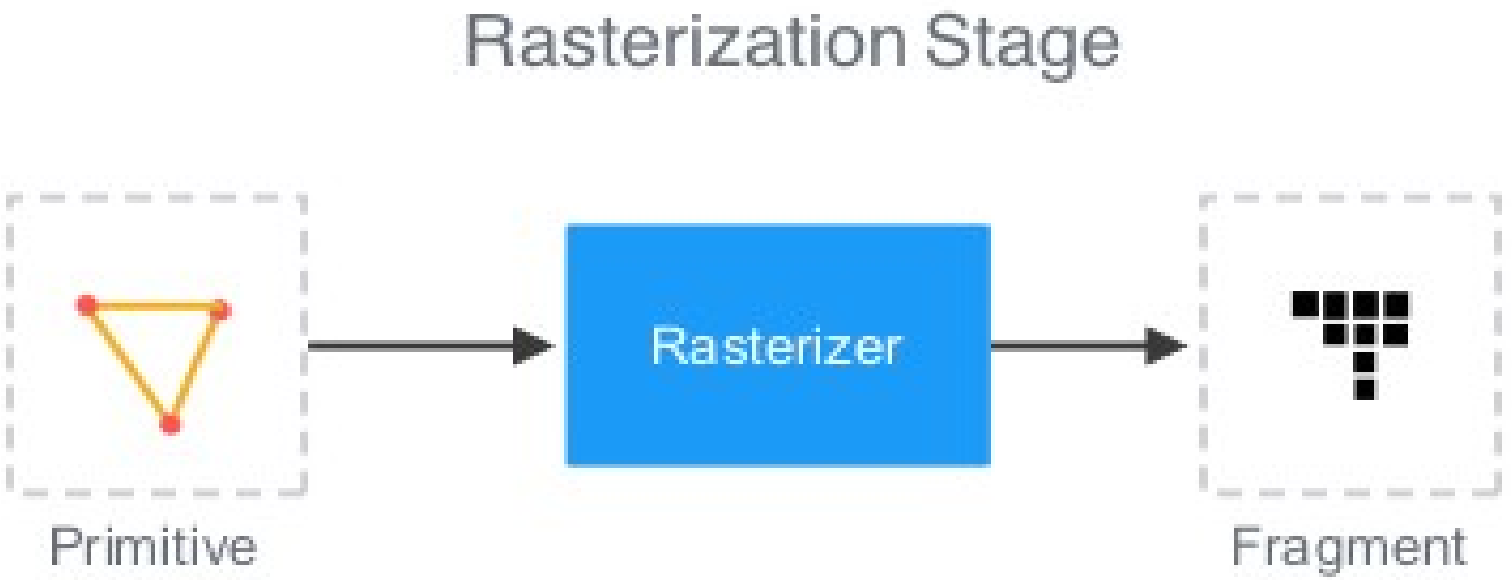
- **Per-Vertex Operation:** Spatial coordinates are projected from a position in the 3D world to a position on your screen by *Vertex Shader*. Just like a photographic camera transforms a 3D scenery into a 2D photograph.

- **Primitive Assembly:** After three vertices have been processed by the vertex shader, they are taken to the *Primitive Assembly* stage. This is where a primitive is constructed by connecting the vertices in a specified order.



- Clipping and Back-face culling are major parts of primitive assembly. Clipping is the elimination of portions of geometry which fall outside view volume (i.e. screen). Back-face culling avoids rendering the primitives facing away from the viewer.

- **Rasterization:** What you ultimately see on a screen are pixels approximating the shape of a primitive. This approximation occurs in the Rasterization stage. In this stage, pixels are tested to see if they are inside the primitive's perimeter. If they are not, they are discarded. If they are within the primitive, they are taken to the next stage. The set of pixels that passed the test is called a fragment.



- **Pixel Operations:** While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, mapped, and processed. The results are then either written into texture memory or sent to the rasterization step
- If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

- **Texture Assembly:** An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic.
- **Fragment Operations:** A Fragment is a set of pixels approximating the shape of a primitive. When a fragment leaves the rasterization stage, it is taken to **Fragment Shader**. The job of the fragment shader is to determine the final color for each fragment.

  Before the pixels in the fragment are sent to the framebuffer, fragments are submitted to several tests like:

  - Pixel Ownership test: It determine which pixels on the screen are "owned" by the object. This test is based on the object's shape and position on the screen.
  - Scissor test: It tests whether the fragment's pixel lies outside of a specified rectangle of the screen.)

- Alpha test: Alpha testing is a technique used to determine whether a fragment (pixel) should be drawn based on its alpha value. The alpha value typically represents the pixel's transparency or opacity. Each pixel in an image or texture can have an associated alpha value, which determines its transparency. An alpha value of 1.0 usually means fully opaque, while 0.0 means fully transparent, and values in between represent varying levels of transparency.

- Depth test: It used to determine which fragments (pixels) should be drawn and which ones should be discarded based on their depth values. A depth buffer (also known as a Z-buffer) is used to store depth value corresponds to the pixels on the screen. A depth value, typically representing the distance from the camera to the object at that pixel.

At the end of the pipeline, the pixels are saved in a Framebuffer.

OpenGL primitives and attributes

1) Points (GL_POINTS):

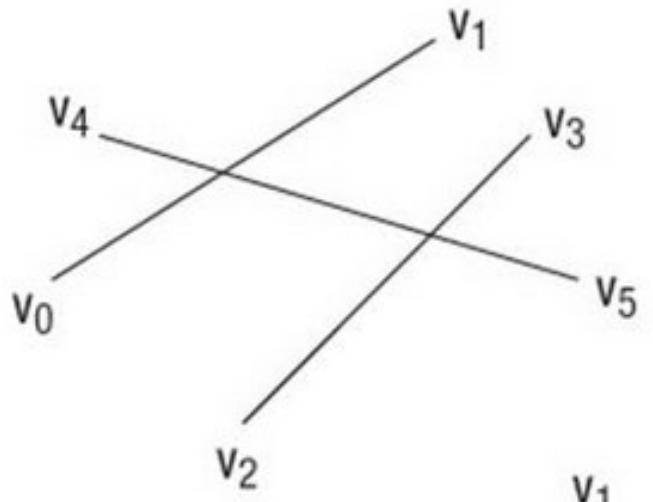
- Points are the simplest primitive in OpenGL. They represent single pixels or vertices in space.
- You can use them to render individual points or to represent vertices of more complex objects.

Attributes: Points are typically represented by a single vertex with attributes like position, color, and texture coordinates.

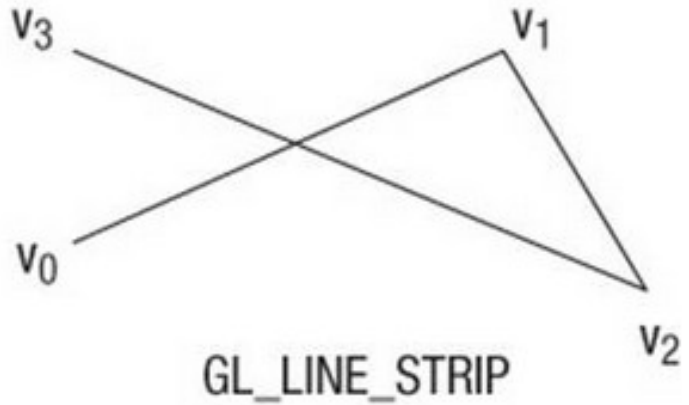
2) Lines (GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP):

- Lines are used to render straight-line segments.
- GL_LINES draws individual line segments between pairs of vertices.
- GL_LINE_STRIP is used for rendering a series of connected line segments. It connects vertices in the order they are specified, creating a continuous line.
- GL_LINE_LOOP is similar to GL_LINE_STRIP but also connects the last vertex to the first, forming a closed loop.

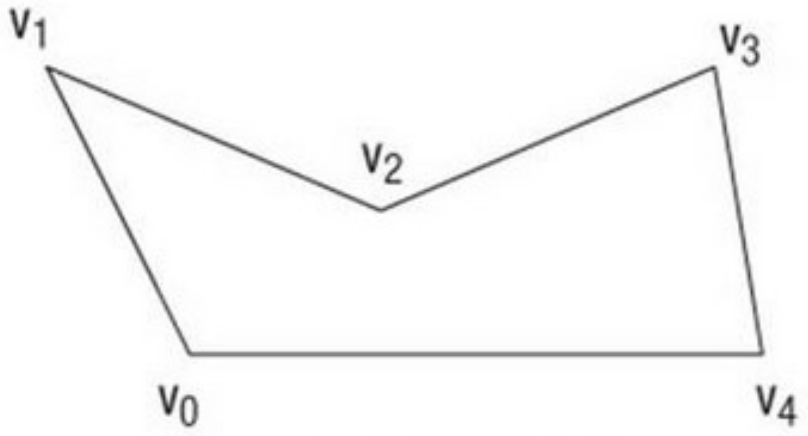
Attributes: Lines have at least two vertices with attributes such as position, color, and texture coordinates.



GL_LINES



GL_LINE_STRIP

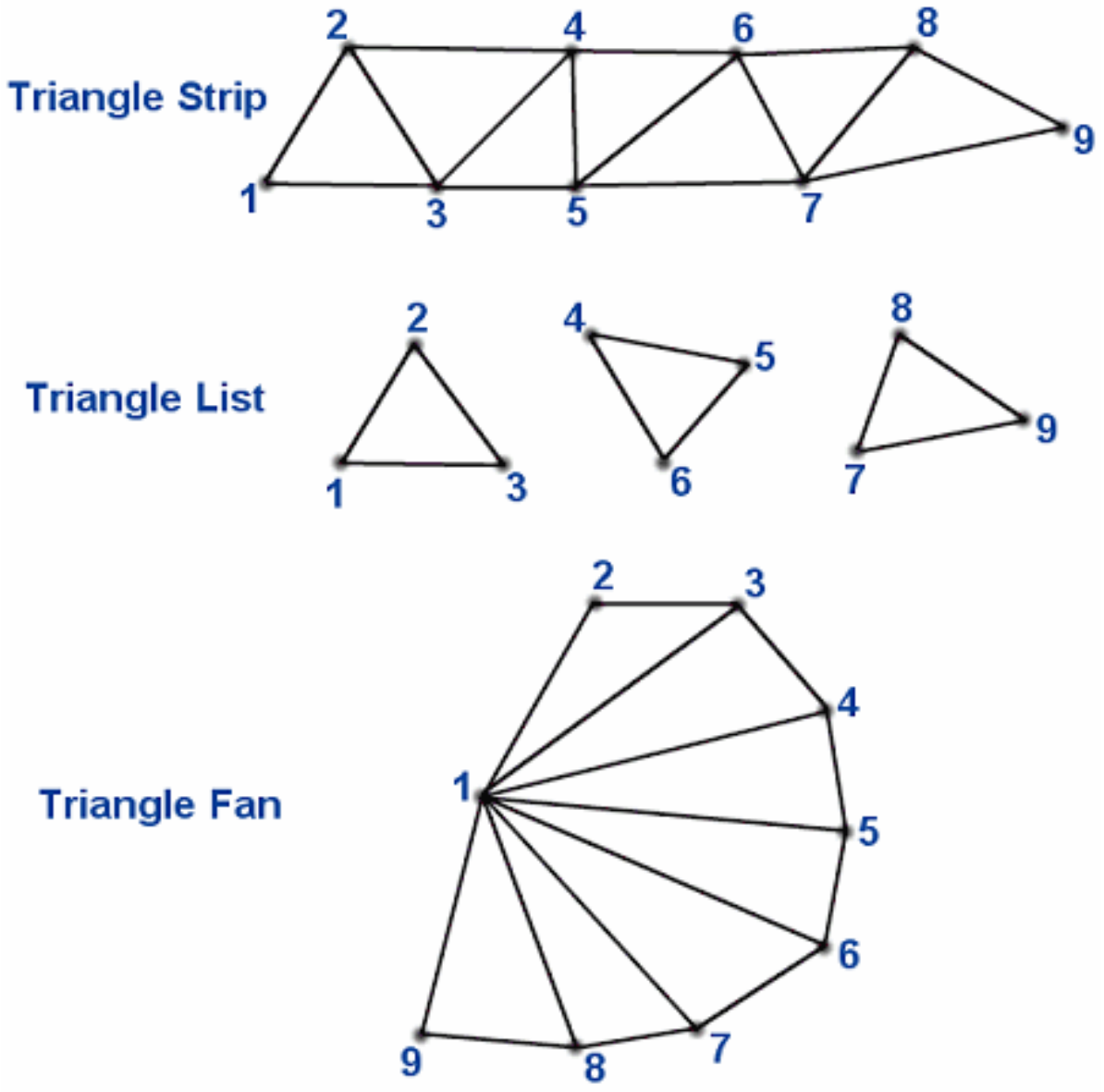


GL_LINE_LOOP

3) Triangles (GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN):

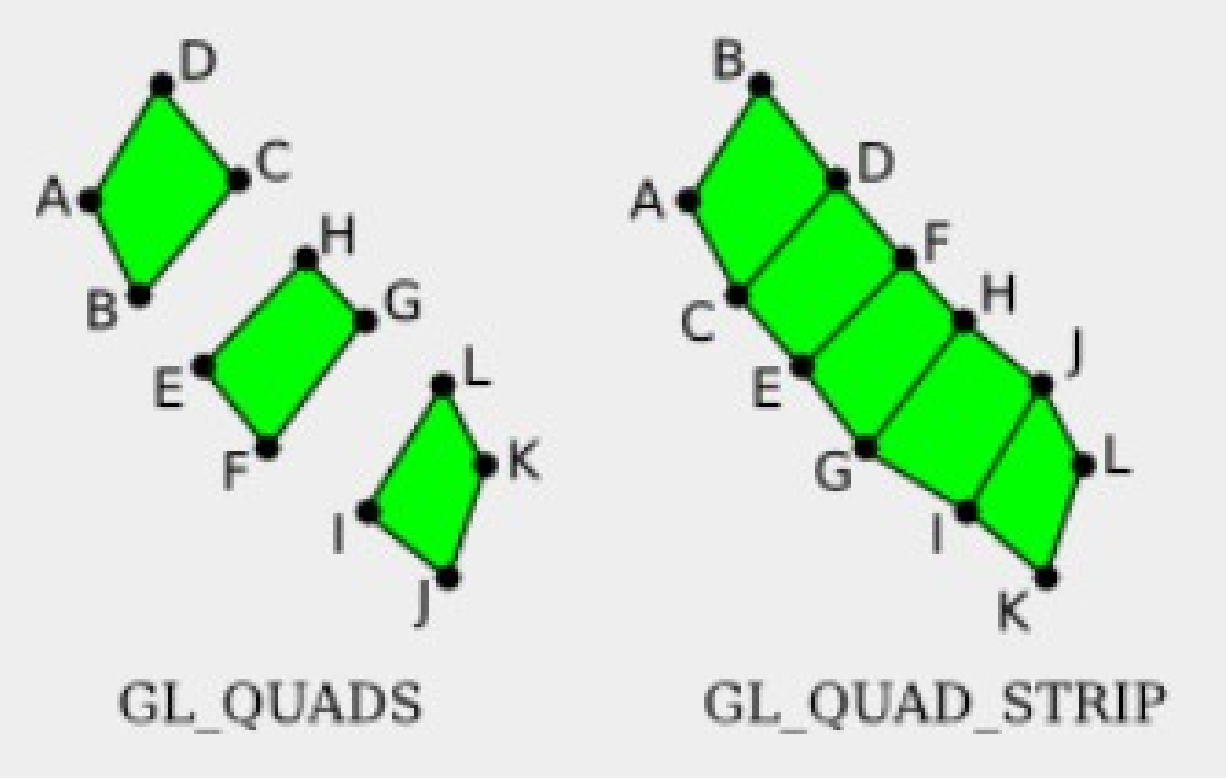
- Triangles are the most commonly used primitive for rendering 3D objects.
- GL_TRIANGLES renders individual triangles between sets of three vertices.
- GL_TRIANGLE_STRIP connects vertices in a strip, forming a series of triangles.
- GL_TRIANGLE_FAN connects vertices in a fan-like manner, also creating triangles.

Attributes: Triangles have three vertices with attributes like position, color, and texture coordinates.


Triangle Strip
1
2
3
4
5
6
7
8
9
Triangle List
1
2
3
4
5
6
7
8
9
Triangle Fan
1
2
3
4
5
6
7
8
9

4) Quads (GL_QUADS, GL_QUAD_STRIP)

- Quads are used to render four-sided polygons.
- GL_QUADS renders individual quads
- GL_QUAD_STRIP connects vertices in a strip fashion, creating a series of connected quads.



5) Polygons (GL_POLYGON):

- Polygons allow you to render arbitrary convex or concave shapes with more than three sides.

6) Patches (GL_PATCHES):

- Patches are used to define complex surfaces by subdividing them into smaller patches. They are essential for achieving curved surfaces and detailed geometry.

OpenGL Primitive Attributes

Here are some common attributes associated with OpenGL primitives:

- Position (Vertex Position): Defines the 3D coordinates of a vertex in space.
- Color: Specifies the color of a vertex or a fragment.
- Texture Coordinates: Used for mapping textures onto primitives. They define how a texture is applied to the surface.
- Normal Vectors: Used for shading calculations and lighting.

Modeling and rendering 2-D objects

Modeling and rendering two- dimensional geometric objects in OpenGL involves several key steps.

1) Define Vertex Data:

- Choose a representation for your 2D geometric object, such as a triangle, rectangle, or custom shape.
- Define the vertices of the object by specifying their 2D coordinates (x, y).
- Optionally, define additional attributes per vertex, like color or texture coordinates.

2) Vertex Array or Buffer:

- Store the vertex data in a data structure, such as an array or list.

3) Transformation:

- Apply transformations to position and orient the 2D object in space.
- Use transformation matrices to control translation, rotation, and scaling.

4) Shader program:

- A shader program typically consists of at least two shaders: a vertex shader and a fragment shader.
- The vertex shader processes each vertex's attributes (e.g., position, color, texture coordinates)
- The fragment shader is responsible for determining the final color of each pixel and can apply various effects, such as texture mapping or lighting calculations.

5) Rendering:

- Render the object by binding the vertex array or buffer, setting shader uniforms, and calling OpenGL's rendering functions
- Use appropriate OpenGL primitives (e.g., GL_TRIANGLES, GL_LINES, GL_POINTS, or GL_POLYGON) to draw the object.

6) Handle User Input:

- Implement input controls to interact with and manipulate the 2D object, if necessary.

Modeling and rendering 3-D objects

Modeling and rendering three- dimensional geometric objects in OpenGL involves several key steps.

1) Define Vertex Data:

- Choose a representation for your 3D geometric object, such as a cube, sphere, or custom shape.
- Define the vertices of the object by specifying their 3D coordinates (x, y, z).
- Specify additional attributes per vertex, such as color, normal vectors, and texture coordinates.

2) Vertex Array or Buffer:

- Store the vertex data in a data structure, such as an array or list.

3) Transformation:

- Apply transformations to position and orient the 3D object in space.
- Use transformation matrices to control translation, rotation, and scaling.

4) Shading and Materials:

- Set up shaders to control the object's appearance, including lighting, materials, and textures.
- Implement lighting models, such as Phong or Lambertian, to simulate how light interacts with the object's surface.

5) Camera Setup:

- Create a camera system with view and projection matrices to define the viewpoint and perspective for the 3D scene.

6) Rendering:

- Render the 3D object by binding the vertex array or buffer, setting shader uniforms, and calling OpenGL's rendering functions.
- Implement depth testing and culling for correct rendering order and performance optimization.

7) Handle User Input:

- Implement input controls to interact with and manipulate the 2D object, if necessary.

GLUT

- GLUT, which stands for the "OpenGL Utility Toolkit," is a library for creating and managing windows, as well as handling user input events in OpenGL applications.
- It provides a simple and platform-independent way to set up graphical user interfaces (GUIs) for OpenGL programs.
- GLUT is not part of the OpenGL specification but is often used in conjunction with OpenGL for developing interactive graphics applications.

Features of GLUT

- **Window Management:** GLUT provides functions for creating and managing windows, including window creation, resizing, and window title management.
- **Input Handling:** It allows you to handle keyboard and mouse input events easily, making it suitable for developing interactive graphics applications.
- **Timer Functions:** GLUT provides timers for scheduling function calls at specific intervals, which can be used for animation or other time-dependent tasks.
- **Menu Creation:** You can create pop-up menus and associate them with specific mouse buttons or keypresses.
- **Simple Main Loop:** It provides a simple main loop, abstracting platform-specific code, and making it easier to get started with OpenGL programming.

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

void display() {
    // OpenGL rendering code goes here
    glClear(GL_COLOR_BUFFER_BIT);
    // Draw your graphics here
    glFlush();
}

int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutCreateWindow("OpenGL with GLUT");

    glutDisplayFunc(display);

    glutMainLoop();
    return 0;
}
```

- **glutInit:** initializes GLUT and parses command-line arguments.
- **glutInitDisplayMode:** sets the display mode (single buffer, RGB color).
- **glutInitWindowSize:** specifies the window size.
- **glutCreateWindow:** creates the window with the given title.
- **glutDisplayFunc:** registers the display function that is called to render the graphics.
- **glutMainLoop:** It is the main loop of your OpenGL application and serves several important purposes like event handing, animation, window management etc.

Interaction with the Mouse and Keyboard

- Interacting with the mouse and keyboard in an OpenGL application using GLUT involves registering callback functions for various input events and responding to those events accordingly.

1) Mouse Callback Functions:

- You can register callback functions for mouse events like mouse click and mouse movement using GLUT functions.
- Common mouse callback functions include ***glutMouseFunc*** for mouse click events and ***glutMotionFunc*** for mouse movement events.

```
// Mouse click callback function
void mouseClick(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        // Handle left mouse button click at (x, y)
    }
}

// Mouse movement callback function
void mouseMove(int x, int y) {
    // Handle mouse movement at (x, y)
}

int main(int argc, char** argv) {
    // Initialize GLUT and create a window

    // Register mouse callback functions
    glutMouseFunc(mouseClick);
    glutMotionFunc(mouseMove);

    // Initialize OpenGL and enter the main loop
    glutMainLoop();
    return 0;
}
```

In the example above:

- ***mouseClick*** is a callback function called when a mouse button is clicked.
- ***mouseMove*** is a callback function called when the mouse is moved.
- You register these callback functions using **glutMouseFunc** and **glutMotionFunc**

2) Keyboard Callback Functions:

A. Keyboard Key Press Event:

- Register a callback function using **glutKeyboardFunc** to handle key press events.
- In following example, ***keyPressed*** is callback function when keyboard key is pressed.

```
void keyPressed(unsigned char key, int x, int y) {
    switch (key) {
        case 'q':
        case 'Q':
        case 27: // Escape key
            exit(0); // Exit the application
            break;
        // Handle other key presses here
    }
}

// Register the keyboard key press callback function
glutKeyboardFunc(keyPressed);
```

B. Keyboard Special Key Press Event (e.g., Arrow Keys):

- For special keys like arrow keys, use **glutSpecialFunc** to register a callback function.
- In following example, ***specialKeyPressed*** is callback function when special keys like arrow key is pressed.

```
void specialKeyPressed(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_LEFT:
            // Handle left arrow key press
            break;
        case GLUT_KEY_RIGHT:
            // Handle right arrow key press
            break;
        // Handle other special key presses here
    }
}

// Register the special key press callback function
glutSpecialFunc(specialKeyPressed);
```

C. Keyboard Key Release Event:

- To detect when a key is released, use **glutKeyboardUpFunc**.
- In following example, **k*eyReleased*** is callback function when key is released.

```
void keyReleased(unsigned char key, int x, int y) {
    // Handle key release here
}

// Register the keyboard key release callback function
glutKeyboardUpFunc(keyReleased);
```

- You can customize the callback functions to respond to specific events and key presses based on your application's requirements.