# GUDLAVALLERU ENGINEERING COLLEGE

**(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)**

**Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.**

## DEPARTMENT

## OF

## COMPUTER SCIENCE AND ENGINEERING

## HANDOUT

**on**

## COMPUTER GRAPHICS

## Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society

## Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behaviour & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

## Program Educational Objectives

**PEO1 :**Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

**PEO2 :** Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations.

**PEO3 :**Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

# HANDOUT ON COMPUTER GRAPHICS

II B.Tech – II Semester Year: 2018-19

OPEN ELECTIVE Credits: **3**

==========

**Brief History and Scope Of The Subject**

- The precursor sciences to the development of modern computer graphics were the advances in electrical engineering, electronics, and television that took place during the first half of the twentieth century. Screens could display art since the Lumiere brothers' use of mattes to create special effects for the earliest films dating from 1895, but such displays were limited and not interactive. The first cathode ray tube, the Braun tube, was invented in 1897 - it in turn would permit the oscilloscope and the military control panel - the more direct precursors of the field, as they provided the first two-dimensional electronic displays that responded to programmatic or user input.

- **Pre-Requisites**
  - Basics of C and graphic elements.
  - Equations of geometric elements.
- **Course Objectives:**
  - To introduce computer graphics applications and functionalities of various graphic systems.
  - To familiarize with 2D and 3D geometrical transformations.
  - To disseminate knowledge on the visible surface detection and animation.
- **Course Outcomes:**

  Upon successful completion of the course, the students will be able to

  **CO1:** design a conceptual model for the mathematical model to determine the set of pixels to turn on for displaying an object.

  **CO2**: analyze the functionalities of various display devices and visible

surface detection methods

**CO3:** analyze the performance of different algorithms to draw different shapes.

**CO4**: choose different transformations and viewing functions on objects.

**CO5:** apply raster animations for Engine oil advertisements.

- **Program Outcomes:**

Graduates of the Computer Science and Engineering Program will have Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

**8.Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

**9.Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

**10.Communication:** Communicate effectively on complex engineering activities with the engineering community and wit society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

**11.Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

**12.Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

- **Mapping of Course Outcomes with Program Outcomes:**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CO1 | | 3 | | | 2 | | | | | 1 | | |
| CO2 | 2 | | 1 | 3 | | | | 2 | | | | |
| CO3 | | 3 | | 2 | | | | | | | 1 | |
| CO4 | 2 | | 3 | | 1 | | 2 | | | | | |
| CO5 | 2 | | 3 | | 1 | | | | | | | 2 |

**3- High 2- Medium 1-Low**

- **Prescribed Text Books**

1. Donald Hearn, M.Pauline Baker, Computer Graphics C version, Pearson.

2. Francis S. Hill, Stephen M. Kelley, "Computer Graphics using OpenGL", 3rd edition, Pearson Education

- **Reference Text Books**

1. Foley, VanDam, Feiner, Hughes, "Computer Graphics Principles and Practice".2nd edition, Pearson Education.

2. Rajesh K Maurya, "Computer Graphics with Virtual Reality Systems", Wiley.

- **URLs and Other E-Learning Resources**
  - IEEE -transactions on Computer Graphics
  - http://www.inf.ed.ac.uk/teaching/courses/cg/Web/intro_graphics.pdf
  - http://www.crazyengineers.com/threads/computer-graphics-project-ideas-topics-for-cs-it-students.58544/
  - https://www.dgp.toronto.edu/~hertzman/418notes.pdf
  - http://freevideolectures.com/Course/2275/Computer-Graphics/20
  - http://cosmolearning.org/courses/introduction-to-computer-graphics-521/video-lectures/
- **Digital Learning Materials:**
  1. https://www.youtube.com/watch?v=m5YbqpL7BIY&index=1&list=PLLOxZwkBK52DkMLAYhRLA_VtePq5wW_N4
  2. https://www.youtube.com/watch?v=D-tV-vZv4Co
- **Lecture Schedule / Lesson Plan**

| TOPIC | No. of Periods | |
|---|---|---|
| | Theory | Tutorial |
| **UNIT-I: Introduction** | | |
| Applications of Computer Graphics | 1 | |
| Raster Scan Systems, Raster scan display processors | 1 | |
| Random scan systems | 1 | |
| Points and Lines | 1 | |
| Line Drawing Algorithms-DDA | 2 | |
| Bresenham,s Line Drawing Algorithm | 1 | |
| Filled Area Primitives: Inside and outside tests | 1 | |
| Boundary Fill Algorithm, Flood Fill Algorithm | 1 | |
| Scan line polygon fill algorithm | 1 | |
| **UNIT-II: 2-D Geometrical Transforms** | | |

| | | |
|---|---|---|
| Translation, Scaling | 1 | |
| Rotation, Reflection | 2 | |
| Shear Transformations | 1 | |
| Matrix Representations | 1 | |
| Homogenous Coordinates | 1 | |
| Composite Transformations | 1 | |
| **UNIT-III: 2-D viewing** | | |
| The viewing pipeline | 1 | |
| Window to viewport coordinate transformation | 2 | |
| Viewing Functions | 1 | |
| Cohen Sutherland line clipping algorithm | 2 | |
| Sutherland Hogeman polygon clipping algorithm | 2 | |
| **UNIT-IV: 3D Geometric Transformations** | | |
| Translation, Scaling | 1 | |
| Rotation, Reflection | 2 | |
| Shear Transformations | 1 | |
| Composite Transformations | 1 | |
| 3D viewing pipeline | 1 | |
| Parallel Projections | 2 | |
| Perspective projections | 2 | |
| **UNIT-V: Visible surface Detection Methods** | | |
| Classification | 1 | |
| Back-face Detection | 1 | |
| Depth Buffer Method | 1 | |
| BSP tree method | 1 | |
| Area sub division method | 2 | |
| **UNIT-VI: Computer Animation** | | |

| | | |
|---|---|---|
| Design of animation sequence | 1 | |
| Raster Animations | 1 | |
| Key frame systems | 1 | |
| Graphics programming using OpenGL | 1 | |
| drawing three dimensional objects | 1 | |
| drawing three dimensional scenes | 2 | |
| **Total No. of Periods:** | **48** | **0** |

- **Seminar Topics**
  - 3D Translation
  - 3D viewing pipeline
  - Key frame systems

# UNIT – I

**Objective:**

To familiarize with the functionalities of various graphics systems.

**Syllabus:**

**Introduction:**

- Applications of Computer Graphics
- Raster scan systems
- Random scan systems
- Raster scan display processors

**Output primitives:**

- Points and lines
- Line drawing algorithms

**Learning Outcomes:**

Students will be able to

- Understand the functionalities of raster and random scan systems.
- Attain a conceptual model understanding of the underlying mathematical model for determining the set of pixels to turn on for displaying an object.

## Learning Material

## ***INTRODUCTION***

Computer graphics is an art of drawing pictures on computer screens with the help of programming. It involves computations, creation, and manipulation of data. In other words, we can say that computer graphics is a rendering tool for the generation and manipulation of images.

### ***Applications of Computer Graphics***

**(1) Computer Aided Design:**

- A major use of computer graphics is in **design processes**, particularly for engineering and architectural systems, but almost all products are now computer designed.
- Computer-Aided Design methods are used in the design of buildings, automobiles, aircraft, watercraft, spacecraft, computers, textiles, and many other products.
- For some design applications, objects are first displayed in a wireframe outline form that shows the overall sham and internal features of objects.
- Wireframe displays also allow designers to quickly see the effects of interactive adjustments to design shapes.

**(2) Presentation Graphics:**

- Presentation graphics, used to produce illustrations for reports using projectors.
- Presentation graphics is commonly used to summarize financial, statistical, mathematical, scientific, and economic data for research reports, managerial reports, consumer information bulletins, and other types of reports.
- Typical examples of presentation graphics are bar charts, line graphs, surface graphs, pie charts, and other displays shows relationships between multiple parameters.

**(3) Computer Art:**

- Computer graphics methods are widely used in both **fine art** and **commercial art** applications.
- Fine artists use a variety of computer technologies to produce images.
- Commercial art is used for logos and other designs, page layouts combining text and graphics, TV advertising spots, and other areas.

**(4) Entertainment:**

- Computer graphics methods am now commonly used in making motion pictures, music videos, and television shows.
- Sometimes the graphics scenes are displayed by themselves, and sometimes graphics objects are combined with the actors and live scenes.

**(5) Education & Training:**

- Computer-generated models of physical, financial, and economic systems are often used as educational aids..
- Graphics enhances the way of teaching.
- For some training applications, special systems are designed. Examples of such specialized systems are the simulators for practice sessions or training of ship captains, aircraft pilots, heavy-equipment operators, and air traffic control personnel.

**(6) Visualization:**

- Scientists, engineers, medical personnel, business analysts, and others often need

  to analyze large amounts of information or to study the behaviour of certain processes.
- If the data are converted to a visual form, the trends and patterns are often immediately apparent. if the data are converted to a visual form, the trends and patterns are often immediately apparent.

**(7) Image Processing:**

- Although methods used in computer graphics and Image processing overlap, the two areas concerned with fundamentally different operations.
- In computer graphics, a computer is used to create a picture.

- Image processing, on the other hand. applies techniques to modify or interpret existing pictures.
- Image processing and computer graphics are typically combined in many applications. Medicine, for example, uses these techniques to model and study physical functions, to design artificial limbs, and to plan and practice surgery. This application is generally referred to as computer-aided surgery.

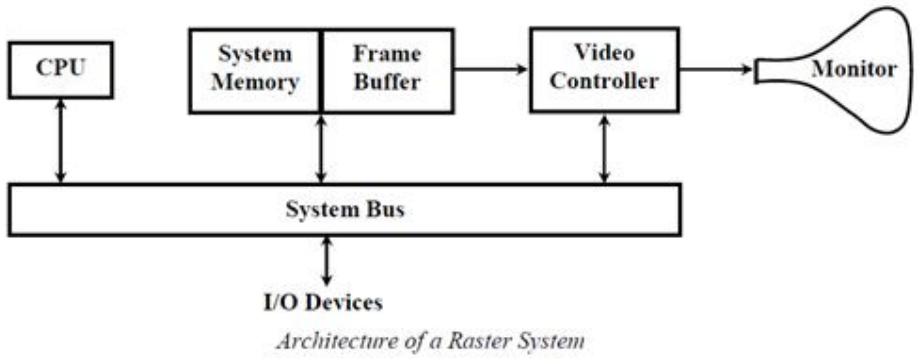
**(8) GUI(Graphical User Interface):**

- A graphic, mouse-oriented paradigm which allows the user to interact with a computer.
- The advantages of icons are that they take up less screen space than corresponding textual descriptions and they can be understood more quickly.

***Raster scan systems***

- Interactive Raster graphics systems typically employ several processing units.
- In addition to the CPU, a special-purpose processor called the ***Video Controller*** or ***Display Controller*** is used to control the operation of the display device.

**Video Controller:**

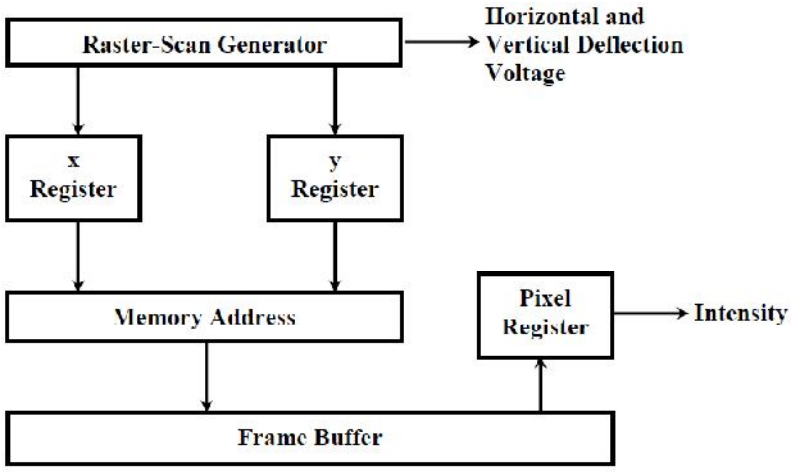



*Architecture of a Raster System*

- A fixed area of the system memory is reserved for the frame buffer, and the Video Controller is given direct access to the Frame-Buffer Memory.

- Frame-Buffer locations and the corresponding screen positions are referred in Cartesian Coordinates.
- For many graphics monitors, the coordinate origin is defined at the lower left screen corner.

**The basic refresh operations of the Video Controller:**




*Basic Video-Controller Refresh Operations*

- Two registers are used to store the coordinates of the screen pixels. Initially, *x* register is set to *0* and *y* is set to *ymax*.
- The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam.
- Then the *x* register is incremented by 1 and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the scan line.
- After the last pixel on the top scan line has been processed, the *x* register is reset to 0 and *y* register is decremented by 1.
- After cycling through all pixels along the bottom scan line (*y = 0*), the Video Controller resets the registers to the first pixel position on the top scan line and the refresh process starts again.
- Since the screen must be refreshed at the rate of 60 frames per second, the cycle time is too slow; this can't be accommodated by typical RAM chips.

- To speed up pixel processing, Video Controllers can retrieve multiple pixel values from the refresh buffer on each pass.
- When that group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.
- In high-quality systems, 2 frame buffers are often provided so that one buffer can be used for refreshing while the other is being filled with intensity values.

***Raster-Scan Display Processor***

- ***Display Processor*** is also referred as a ***Graphics Controller*** or a ***Display Coprocessor***.
- The purpose of the display processor is to free the CPU from the graphics chores.
- In addition to the system memory, a separate display processor memory area can also be provided.
- A major task of the display processor is digitizing a picture definition given in an application
- program into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called ***Scan Conversion***.
- Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete intensity points.
- Similar methods are used for scan converting curved lines and polygon outlines.
- For Example, Characters can be defined with rectangular grids or they can be defined with curved outlines.

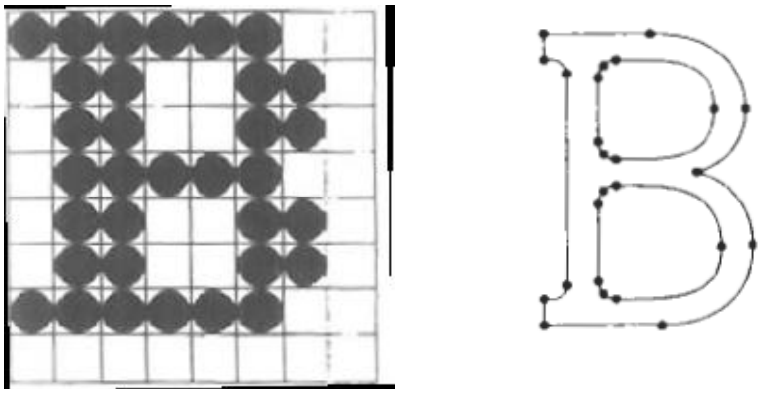
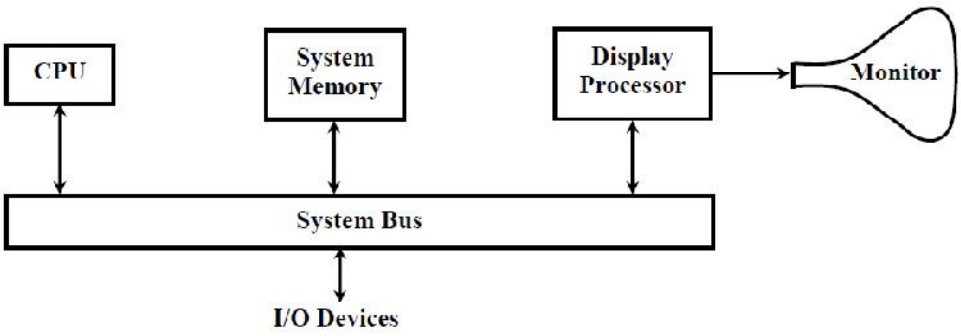


Fig : A character defined as a rectangular Grid of Patterns & Curved Outlines

- The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher-quality displays.
- Display Processors are also designed to perform a number of additional operations.
- These functions include generating various line styles (dashed, dotted or solid), displaying color areas and performing certain transformations and manipulations on displayed objects.

***Random scan systems***




- An application program is input and stored in the system memory along with a graphics package.
- Graphics commands in the application program are translated by the graphics package into a display file stored in the system memory.
- This display file is then accessed by the display processor to refresh the screen.

- The display processor cycles through each command in the display file program once during every refresh cycle.
- Sometimes the display processor in a Random-Scan system is referred to as a ***Display Processing Unit*** or a ***Graphics Controller***.
- Graphics patterns are drawn on a random-scan system by directing the electron beam along the component lines of the picture.
- Lines are defined by the values for their coordinate endpoints, and these input coordinate values are converted to *x* and *y* deflection voltages.

## ***OUTPUT PRIMITIVES***

- Graphics programming packages provide functions to describe a scene in terms of these basic geometric structures, referred to as ***Output Primitives.***

### ***Points and Lines:***

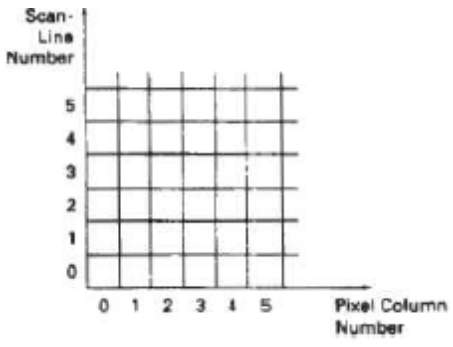
- Point plotting is accomplished by converting a single coordinate position furnished by an application program into appropriate operations for the output device.
- A Random-Scan (Vector) System stores point-plotting instructions in the display list, and coordinate values in these instructions are converted to deflection voltages that position the electron beam at the screen locations to be plotted during each refresh cycle.
- For a black-and-white raster system, a point is plotted by setting the bit value corresponding to a specified screen position within the frame buffer to 1. Then, as the electron beam sweeps across each horizontal scan line, it emits a burst of electrons (plots a point) whenever a value of 1 is encountered in the frame buffer.
- With an RGB system, the frame buffer is loaded with the color codes for the intensities that are to be displayed at the screen pixel positions.

***Line drawing algorithms:***

- Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints.
- For a raster video display, the line color (intensity) is then loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller then "plots" the screen pixels. Screen locations are referenced with integer values, so plotted positions may only approximate actual line positions between two specified endpoints.
- *For example,* a computed line position is (10.48, 20.51), it is rounded to (10, 21). This rounding of coordinate values to integers causes lines to be displayed with a **stair step appearance ("the jaggies"),** as represented below.

- This stair step shape is noticeable in low resolution systems.
- For the raster-graphics device-level algorithms, object positions are specified directly in integer device coordinates.
- To load a specified color into the frame buffer at a position corresponding to ***column x along scan line y,*** we will assume we have available a low-level procedure of the form

  **setPixel (x, y)**




- Sometimes we want to retrieve the current frame-buffer intensity setting for a specified location. We

accomplish this with the low-level function. We use, **getPixel (x, y)**

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

$$y = m \cdot x + b \qquad (3\text{-}1)$$

with m representing the slope of the line and b as they intercept.

- Given that the **two** endpoints of a line segment are specified at positions ***(x1, y2)*** and ***(x2, y2)*** as shown in Fig. **3-3,** we can determine values for the slope m and y intercept b with the following calculations:

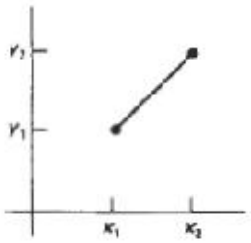


*Figure 3-3*
Line path between endpoint positions $(x_1, y_1)$ and $(x_2, y_2)$.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \qquad (3\text{-}2)$$

$$b = y_1 - m \cdot x_1 \qquad (3\text{-}3)$$

- Algorithms for displaying straight lines are based on the line equation 3-1 and the calculations given in Eq: **3-2** and **3-3.**
- For any given ***x*** interval D**x** along a line, we can compute the corresponding y interval from Eq: 3-2 as

$$\Delta y = m\,\Delta x \qquad (3\text{-}4)$$

- Similarly, we can obtain the ***x*** interval D***x*** corresponding to a specified D**y as**

$$\Delta x = \frac{\Delta y}{m} \qquad (3\text{-}5)$$

- These equations form the basis for determining deflection voltages in analog devices.
- For lines with slope magnitudes **| *m* | < 1,** D**x** can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to D**y** as calculated from Equation 3.4.

- For lines whose slopes have magnitudes **| *m* | > 1**, D**y** can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to D**x**, calculated from Equation 3.5.
- For lines with ***m* = 1,** D**x** = D**y** and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.

***DDA algorithm***

- The **Digital Differential Analyzer (DDA)** is a Scan-Conversion line algorithm based calculating either Dy or Dx using equations (4) and (5).
- Consider first a line with **positive slope, less than or equal to 1**, we sample at unit intervals (Dx=1) and compute each successive y value as

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

- subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final endpoints is reached. Since m can be any real number between 0 & 1, the calculated y values must be rounded to the nearest integer.
- For lines with a **positive slope greater than 1**, we reserve the roles of x & y. That is, we sample at unit y intervals (Dy=1) and calculate each succeeding x value as

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

- 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 that, then either we have Dx= -1 and or (when the slope is greater than 1) we have y = -1 with Equations (6), (7), (8) and (9) can also be used to calculate pixel positions along a line with negative slope.

- If the absolute value of the slope is less than 1 and the start endpoint is at the left, we set Dx=1 and calculate y values with equation (6).

$$y_{k+1} = y_k - m \longrightarrow (8)$$

$$x_{k+1} = x_k - \frac{1}{m} \longrightarrow (9)$$

- When the start endpoint is at the right (for the same slope), we set Dx= -1 and obtain y positions from equation (8). Similarly, when the absolute value of a negative slope is greater than 1, we use Dy= -1 and equation (9) or we use Dy=1 and equation (7).

```
# define ROUND (a) ((int) (a+0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
     int dx = xb - xa, dy = yb - ya, steps, k;
     float xIncrement, yIncrement, x = xa, y = ya;
     if (abs (dx) > abs (dy))
          steps = abs (dx) ;
     else
          steps = abs (dy);
     xIncrement = dx / (float) steps;
     yIncrement = dy / (float) steps;
     setpixel (ROUND(x), ROUND(y)) :
     for (k=0; k<steps; k++)
     {
          x += xIncrment;
          y += yIncrement;
          setpixel (ROUND(x), ROUND(y));
     }
}
```

Draw a line from (0,0) to (-8,-4) using DDA algorithm

| i | plot | x | y |
|---|---|---|---|
| 1 | (0,0) | -1.0 | -0.5 |
| 2 | (-1,-1) | -2.0 | -1.0 |
| 3 | (-2,-1) | -3.0 | -1.5 |
| 4 | (-3,-2) | -4.0 | -2.0 |
| 5 | (-4,-2) | -5.0 | -2.5 |
| 6 | (-5,-3) | -6.0 | -3.0 |
| 7 | (-6,-3) | -7.0 | -3.5 |
| 8 | (-7,-4) | -8.0 | -4.0 |

**Advantages :**

- The DDA algorithm is a **faster method** for calculating pixel position than the direct use of slope intercept form equation.

**Disadvantages:**

- The accumulation of round off error in successive additions of the floating-point increment, however, can cause the calculated pixel positions **to drift away from the true line path** for long line segments.
- Furthermore, the rounding operations and floating-point arithmetic in procedure line DDA are still **time-consuming**.
- The algorithm is **orientation dependent**. Hence end point accuracy is poor.

**Improvements can be done in this algorithm are :**

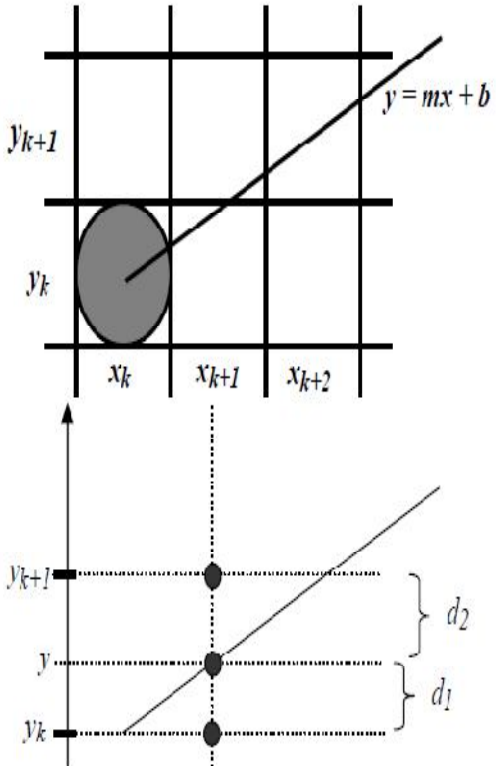
- We can improve the performance of the DDA algorithm by separating the increments m and 1/m into integer and fractional parts so that all calculations are reduced to integer operations.

***Bresenham's Line Algorithm***

- An accurate and efficient raster line-generating algorithm, developed by Bresenham, will scan converts lines using only incrementa1

integer calculations that can be adapted to display circles and other curves.

- We first consider the scan-conversion process for lines with positive slope less than 1. Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x0, y0) of a line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.




- The beside 1st figure demonstrates the kth step in this process.
- Assuming we have determined that the pixel at (xk, yk) is to be displayed, we next need to decide which pixel to plot in column xk+1.
- Our choices are the pixels at positions (xk+1, yk) and (xk+1, yk+1).
- At sampling position xk+1, we label vertical pixel separations from the mathematical line path as d1 and d2 shown in the besides 2nd figure .
- The y coordinate on the mathematical line at pixel column position xk+1 is calculated as

**y = m (xk + 1) + b ----------(1)**

Then,

**d1 = y - yk**

**= m (xk + 1) + b - yk**

and

**d2 = (yk+1) - y**

**= yk + 1 – m (xk + 1) - b**

- The difference between these two separations is

  **d1 - d2 = 2m (xk + 1) - 2yk + 2b – 1 ---------- (2)**

- A ***decision parameter pk*** for the kth step in the line algorithm can be obtained by rearranging equation (2) so that it involves only integer calculations.
- We accomplish this by substituting m = Dy / Dx, where Dy and Dx are the vertical and horizontal separations of the endpoint positions, and defining:

  **pk = Dx(d1-d2)**

  **= 2Dy.xk - 2Dx . yk + c ----------(3)**

- The sign of pk is the same as sign of d1-d2, since Dx > 0 for our examples.
- Parameter c is constant and has the value 2Dy + Dx (2b-1), which is independent of pixel position and will be eliminated in the recursive calculations for pk.
- If the pixel at yk is closer to the line path than the pixel at yk+1(i.e., d1<d2), then decision parameter pk is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.
- Coordinate changes along the line occur in unit steps in either the x or y directions.
- Therefore, we can obtain the values of successive decision parameters using incremental integer calculations.
- At step k+1, the decision parameter is evaluated from equation (3) as

  **pk+1 = 2Dy . xk+1 – 2Dx . yk+1 + c**

  Substituting equation (3) from the preceding equation, we have

  pk+1 – pk = 2Dy (xk+1 - xk) - 2Dx (yk+1 - yk)

  But xk+1 = xk + 1, so that

  **pk+1 = pk + 2Dy - 2Dx(yk+1-yk) ---------- (4)**

  where the term yk+1 – yk is either 0 or 1, depending on the sign

parameter pk.

- This recursive calculation of decision parameter is performed at each integer x position, starting at the left coordinate endpoint of the line.
- The first parameter, p0, is evaluated from 3 at the starting pixel position (x0,y0) and with m evaluated as Dy / Dx:

**p0 = 2Dy - Dx ----------(5)**

- We can summarize Bresenham line drawing for a line with a **positive slope less than 1** in the following listed steps:

*1. Input 2 endpoints, store left endpoint in (x0, y0).*

*2. Load (x0, y0) into frame buffer, i.e. plot the first point.*

*3. Calculate constants □x, □y, 2□y, 2□y – 2□x, and initial value of decision parameter:*

$$p0 = 2\square y - \square x$$

*4. At each xk along the line, start at k=0, test:*

*if pk < 0, plot (xk+1, yk) and*

$$pk+1 = pk + 2\square y$$

*else plot (xk+1, yk+1) and*

$$pk+1 = pk + 2\square y - 2\square x$$

*5. Repeat step (4) □x times.*

Example: Draw a line from (20,10) and (30, 18) using Bresenham's Line Drawing Algorithm

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

The initial decision parameter has the value

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

and the increments for calculating successive decision parameters are

$$2\Delta y = 16, \qquad 2\Delta y - 2\Delta x = -4$$

We plot the initial point $(x_0, y_0) = (20, 10)$, and determine successive pixel positions along the line path from the decision parameter as

| $k$ | $p_k$ | $(x_{k+1}, y_{k+1})$ | $k$ | $p_k$ | $(x_{k+1}, y_{k+1})$ |
|---|---|---|---|---|---|
| 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) |

## UNIT-I
## Assignment-Cum-Tutorial Questions
## SECTION-A

### Objective Questions

1. The number of pixels stored in the frame buffer of a graphics system is known a [ ]
   a. Resolution b. Depth c. Resolution d. a & b

2. The application area of computer graphics are [ ]
   a. Training b. Education
   c. CAD and entertainment d. All of these

3. The purpose of display processor is __from the graphics routine task?
   a. to free the CPU b. To free the secondary memory
   c. to free the main memory d. Both a & c [ ]

4. CAD means [ ]
   a. Car aided design b. Computer art design
   c. Computer aided design d. None of these

5. What are the components of Interactive computer graphics [ ]
   a. A digital memory or frame buffer b. A television monitor
   c. An interface or display controller d. All of these

6. A display controller serves to pass the contents of [ ]
   a. Frame buffer to monitor b. Monitor to frame buffer
   c. Both a & b d. None of these

7. On a black and white system with one bit per pixel, the frame buffer is commonly called as [ ]
   a. Pix map b. Multi map c. Bitmap d. All of the mentioned

8. To store black and white images ,black pixels are represented by_____in the frame buffer and white pixels by_____ [ ]
   a. Zero and one b. One and Zero c. Both a & b d. None of these

9. Examples of Presentation Graphics is [ ]
   a. Bar charts b. CAD c. Line Graphs d. A and C

10. The basic attributes of a straight line segment are [ ]
    a. Type b. Width c. Color d. All of these

11. The Cartesian slope-intercept equation for a straight line is [ ]
    a. y = m.x + b b. y = b.x + m c. y = x.x + m d. y = b + m.m

12. For lines with slope magnitude |m|<1, x can be_____ [ ]

a. A set corresponding vertical deflection
b. A set proportional to a small horizontal deflection voltage
c. Only a d. All of these

13. On raster system, lines are plotted with [ ]

a. Lines b. Dots c. Pixels d. None

14. Aspect ratio means [ ]

a. Number of pixels
b. Ratio of vertical points to horizontal points
c. Ratio of horizontal points to vertical points
d. Both b and c

15. Which algorithm is a faster method for calculating pixel positions?

a. Bresenham's line algorithm b. Parallel line algorithm [ ]
c. Mid-point algorithm d. DDA line algorithm

16. In Bresenham's line algorithm, if the distances d1 < d2 then decision parameter $P_k$ is_____ [ ]

a. Positive b. Equal c. Negative d. Option a(or)c

17. A line connecting the points (1, 1) and (5, 3) is to be drawn, using DDA algorithm. Find the value of x and y increments [ ]

a. x-increments = 1; y-increments =1
b. x-increments = 0.5; y-increments =1
c. x-increments = 1; y-increments =0.5
d. None of above

18. Raster is a synonym for the term ? [ ]

a. Array b. Matrix c. Model d. All of above

19.Digitizing a picture definition into a set of intensity values is known as ..... [ ]

a. Digitization b. Scan conversion c. Refreshing d. Scanning

20.....will free the CPU from graphics chores. [ ]

a. Display processor b. Monitor c. ALU d. Video controller

21.The ..... simply reads each successive byte of data from the frame buffer. [ ]

a. Digital Controller b. Data Controller c. Display Controller d. All of above

22. An accurate and efficient raster line-generating algorithm is [ ]

a.DDA algorithm
b. Mid-point algorithm
c. Parallel line algorithm
d. Bresenham's line algorithm

## SECTION-B

## SUBJECTIVE QUESTIONS

1. Define Computer Graphics. List and explain the applications of computer Graphics?
2. Explain about Raster Scan Systems?
3. What is the purpose of having a separate Display Processor?
4. How refreshing operations are done using the video controller?
5. Explain the architecture of Random Scan Systems?
6. Explain in detail about the DDA scan conversion algorithm?
7. Explain Bresenham's line drawing algorithm?
8. Explain scan line polygon fill algorithm.
9. Explain the method for determining whether the point is inside or outside the region

## SECTION-C

## QUESTIONS AT THE LEVEL OF GATE

1. How Many k bytes does a frame buffer needs in a 600 x 400 pixel?
2. Consider two raster systems with the resolutions of 640 x 480 and 1280 x 1024.How many pixels could be accessed per second in each of these systems by a display controller that refreshes the screen at a rate of 60 frames per second?

3. Consider three different raster systems with resolutions of 640 x 480, 1280 x 1024, and 2560 x 2048.

    a. What size is frame buffer (in bytes) for each of these systems to store 12 bits per pixel?

    b. How much storage (in bytes) is required for each system if 24 bits per pixel are to be stored?

4. How much time is spent scanning across each row of pixels during screen refresh on a raster system with resolution of 1280 X 1024 and a refresh rate of 60 frames per second?

5. Plot the intermediate pixels for a line with endpoints (20,10) and (30,18) using Bresenham's line drawing algorithm.

6. Digitize the line with endpoints (0,0) and (-8,-4) using DDA line drawing algorithm.

# UNIT-II

**Objective:**

To perform different transformation on objects.

**Syllabus:**

**2-D geometrical transforms**: Translation, Scaling, Rotations, reflection, shear transformations, matrix representations and homogenous coordinates, composite transformations.

**Learning Outcomes:**

Student will be able to:

- Explain various geometrical transformations.
- Understand the representations of transformations

## Learning Material

### 2.1 BASIC TRANSFORMATIONS

- Changes in orientation, size, and shape are accomplished with geometric transformations that alter the coordinate descriptions of objects.
- The basic geometric transformations are **translation, rotation, and scaling.**
- Other transformations that are often applied to objects include **reflection and shear.**

### 2.1.1 Translation

- A translation is applied to an object by repositioning it along a straight-line path from one coordinate location to another.
- We translate a two-dimensional point by adding translation distances, $t_x$ and $t_y$, to the original coordinate position (x, y) to move the point to a new position (x', y').

$x' = x + t_x$ ,

$y' = y + t_y$

- The translation distance pair ($t_x$ , $t_y$ ) is called a **translation vector or shift vector**.
- We can express the translation equations as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

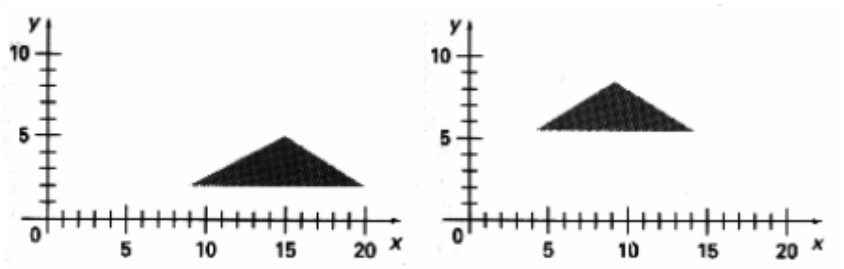
$$P = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad P' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}, \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- This allows us to write the two-dimensional translation equations in the matrix form:

$$\mathbf{P' = P + T}$$

- Sometimes matrix-transformation equations are expressed in terms of coordinate row vectors instead of column vectors. In this case, we would write the matrix representations as :

$$\mathbf{P = [x\ y]\ and\ T = [t_x\ t_y]}$$




- Translation is a **rigid-body transformation** that moves objects without deformation, i.e., every point on the object is translated by the same amount.
- **Polygons** are translated by adding the translation vector to the coordinate position of each vertex and regenerating the polygon using the new set of vertex coordinates and the current attribute settings. Similar methods are used to translate **curved objects**.

- To change the position of a **circle or ellipse**, we translate the center coordinates and redraw the figure in the new location.
- We translate other curves **(splines)** by displacing the coordinate positions defining the objects, and then we reconstruct the curve paths using the translated coordinate points.

### 2.1.2 Scaling

- A scaling transformation alters the size of an object. This operation can be carried out for polygons by multiplying the coordinate values (x, y) of each vertex by scaling factors $s_x$ and $s_y$ to produce the transformed coordinates (x', y'):

  **$x'=x.\ s_x \quad y'=y.s_y$**

- Scaling factor $s_x$, scales objects in the x direction, while $s_y$ scales in the y direction.
- The transformation equations can be written in the matrix form as,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$P' = S \cdot P$$

where S is 2 by 2 scaling matrix.

- Specifying a value of 1 for both sx and sy leaves the size of objects unchanged.
- When $s_x$ and $s_y$ are assigned the same value, a uniform scaling is produced that maintains relative object proportions.
- Unequal values for $s_x$ and $s_y$ result in a differential scaling that are often used in design applications, when pictures are constructed from a few basic shapes that can be **adjusted** by scaling and positioning transformations.

### 2.1.3 Scaling About a fixed point:

- We can control the location of a scaled object by choosing a position, called the **fixed point that is to remain unchanged** after the scaling transformation.
- Coordinates for the fixed point ($x_f$ , $y_f$) can be chosen as one of the vertices, the object centroid, or any other position. A polygon is then scaled relative to the fixed point by scaling the distance from each vertex to the fixed point.
- For a vertex with coordinates (x, y), the scaled coordinates (x', y') are calculated as:

$$x' = x_f + (x - x_f)\, s_x \quad , \quad y' = y_f + (y - y_f)\, s_y$$

- We can rewrite these scaling transformations to separate the multiplicative and additive terms:

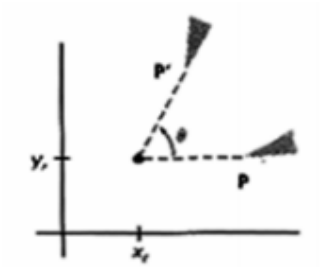
$$x' = x \cdot s_x + x_f(1 - s_x)$$
$$y' = y \cdot s_y + y_f(1 - s_y)$$

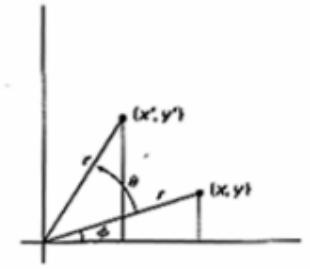
where the additive terms $\mathbf{x_f (1-s_x)}$, $\mathbf{y_f (1-s_y)}$ are constant for all points in the object.

### 2.1.4 Rotation

- A two-dimensional rotation is applied to an object by repositioning it along a circular path in the xy plane. To generate a rotation, we specify a rotation angle □ and the position (x, y) of the rotation point (or pivot point) about which the object is to be rotated.
- Positive values for the rotation angle define **counterclockwise rotations**. Negative values rotate objects in the **clockwise direction**.

- We first determine the transformation equations for rotation of a point position P when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point




positions are shown in the diagram.

- In this figure, **r is the constant distance** of the point from the origin, **angle φ** is the original angular position of the point from the horizontal, and **θ is the rotation angle**. Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ and φ as:

$$x' = r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi+\theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

- The original coordinates of the point in polar coordinates are,

**x = r cos φ y = r sin φ**

- Substituting expressions 2nd into 1st, we obtain the transformation equations for rotating a point at position (x, y) through an angle θ about the origin:

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

- We can write the rotation equations in the matrix form:

**P′ = R . P**

where the rotation matrix is

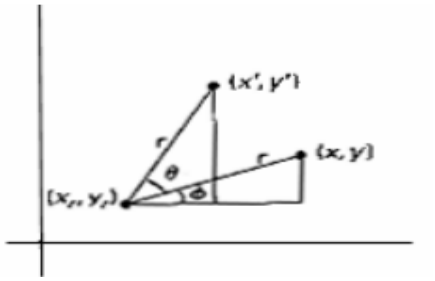
$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

- When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation is transposed so that the transformed row coordinate vector [x' y'] calculated as:

$$P'^T = (R \cdot P)^T$$
$$= P^T \cdot R^T$$

where P'T = [x y], and the transpose R T of matrix R is obtained by interchanging rows and columns. For a rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

### 2.1.5 Rotation of a point about an arbitrary pivot position




- We can generalize to obtain the transformation equations for rotation of a point about any specified rotation position ($x_r$, $y_r$):

$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta + (y - y_r)\cos\theta$$

- Rotations are **rigid-body transformations** that move objects without deformation. Every point on an object is rotated through the same angle.
- **Polygons** are rotated by displacing each vertex through the specified rotation angle and regenerating the polygon using the new vertices.

- **Curved lines** are rotated by repositioning the defining points and redrawing the curves.
- **A circle or an ellipse**, can be rotated about a non central axis by moving the center position through the arc that subtends the specified rotation angle. An ellipse can be rotated about its center coordinates by rotating the major and minor axes.

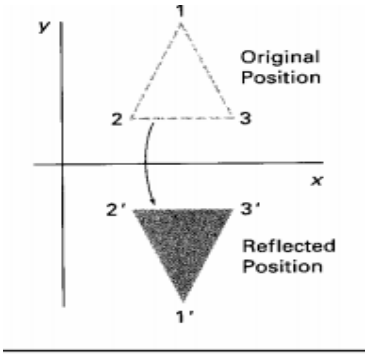
### 2.1.6 Reflection

- A reflection is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an axis of reflection by rotating the object 180" about the reflection axis. Some common reflections are as follows:

**x-Reflection:**

- Reflection about the line y = 0, the x axis, is accomplished with the transformation Matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

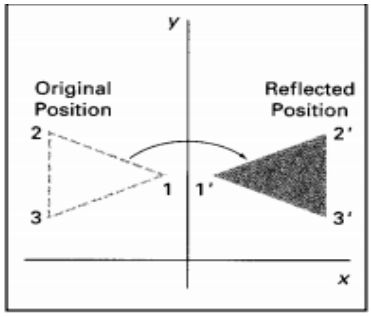



- This transformation keeps x values the same, but "flips" the y values of coordinate positions.

**y-Reflection:**

- A reflection about the y axis flips x coordinates while keeping y coordinates the same. The matrix for this transformation is:

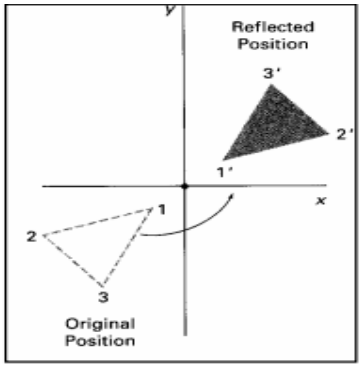
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$




**Origin-Reflection:**

- We flip both the x and y coordinates of a point by reflecting relative to an axis that is perpendicular to the xy plane and that passes through the coordinate origin. This transformation, referred to as a **reflection relative to the coordinate origin**, has the matrix representation:

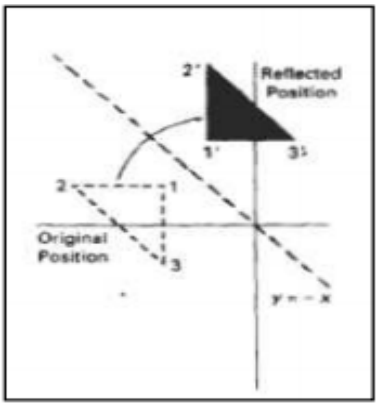
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$




**Reflection along diagonal y=x:**

- The reflection matrix is:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$




**Reflection along diagonal y=x:**

- To obtain a transformation matrix for reflection about the diagonal y = - x, we could concatenate matrices for the transformation sequence:

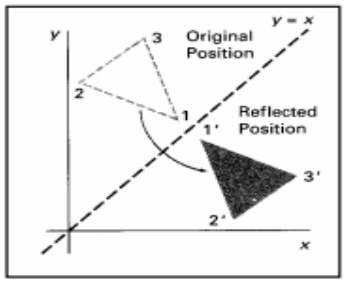
(1) clockwise rotation by 45°,

(2) reflection about the y axis, and

(3) counterclockwise rotation by 45°.

- The resulting transformation matrix is:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$




- Reflections about any line y = mx + h in the xy plane can be accomplished with a combination of translate-rotate-reflect transformations.

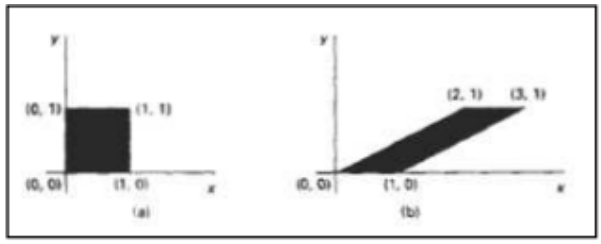
### 2.1.7 Shear

- A transformation that distorts (deform or alter) the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a shear.
- Two common shearing transformations are those that shift coordinate x values and those that shift y values.

**x-Shearing:**

- An x-direction shear relative to the x axis is produced with the transformation matrix:

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



- This transforms coordinate positions as:

  **x' = x + shx . y  y' = y**

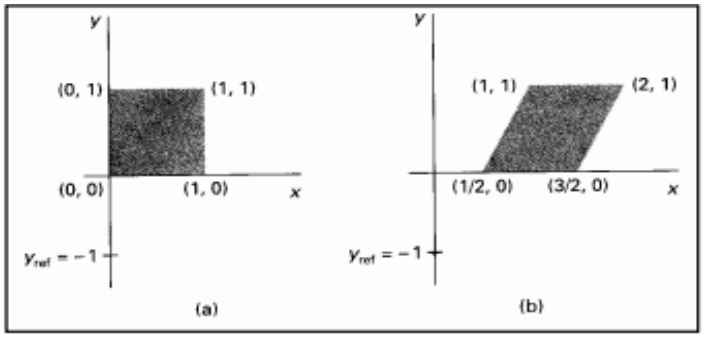
- In the following diagram, shx =2, changes the square into a parallelogram.
- Negative values for $sh_x$ shift coordinate positions to the left.
- We can generate x-direction shears relative to other reference lines

$$\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ with

with coordinate positions transformed as

**$x' = x + sh_x ( y - y_{ref} ) , y' = y$**

- An example of this shearing transformation is given in the following diagram for a shear parameter of ½ relative to the line $y_{ref} = -1$.




**y-Shearing:**

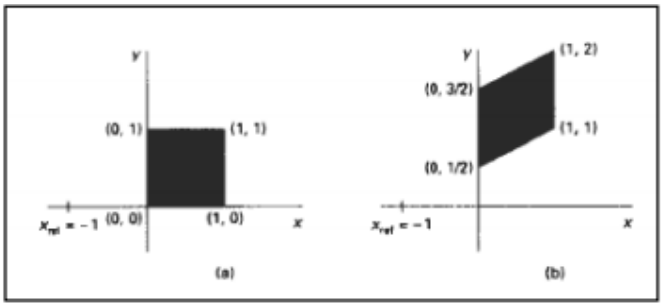
- A y-direction shear relative to the line x = xref is generated with the transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

which generates transformed coordinate positions

**$x' = x$  $y' = sh_y ( x - x_{ref} ) + y$**

- This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{ref}$.
- The diagram shows the conversion of a square into a parallelogram with $sh_y = ½$ and $x_{ref} = -1$.




## 2.2 Matrix Representation and Homogeneous Coordinates

- Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an object to be translated and rotated at each increment of the motion.
- In design and picture construction applications, we perform translations, rotations, and scalings to tit the picture components into their proper positions.
- Each of the basic transformations can be expressed in the general matrix form

$$P' = M1 \cdot P + M2$$

with coordinate positions P and P' represented as column vectors.

- Matrix M1 is a 2 by 2 array containing multiplicative factors, and M2 is a two-element column matrix containing translational terms.
- **For translation**, M1 is the identity matrix.
- **For rotation**, M2 contains the translational terms associated with the pivot point.
- **For scaling**, M2 contains the translational terms associated with the fixed point.
- To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinate one step at a time.

- To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position (x, y) with the homogeneous coordinate triple **$(x_h, y_h, h)$** where

$$x = \frac{x_h}{h} \ , \ y = \frac{y_h}{h}$$

- Thus, a general homogeneous coordinate representation can be written as

**(h.x, h.y, h).**

- For two dimensional geometric transformations, we can choose the homogeneous parameter h to be any nonzero value. A convenient choice is simply to set h = 1. Each two-dimensional position is then represented with homogeneous coordinates (x, y, 1).
- When a Cartesian point (x, y) is converted to a homogeneous representation (xh, yh, h) equations containing x and y such as f (x, y) = 0, become homogeneous equations in the three parameters $x_h$, $y_h$ and h. Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications. Coordinates are represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices.
- **For Translation**,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**$P' = T(t_x, t_y) \cdot P$**

with **$T(t_x, t_y)$** as the 3 by 3 translation matrix.

The inverse of the translation matrix is obtained by replacing the translation parameters **$t_x$ and $t_y$** with their negatives **$-t_x$ and $-t_y$ .**

- **For Rotation,** equations about the coordinate origin are written as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{P' = R(\theta) \cdot P}$$

The rotation transformation operator R (θ) is the 3 by 3 matrix with rotation parameter θ. We get the inverse rotation matrix when θ is replaced with -θ.

- **For Scaling,** relative to the coordinate origin:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{P' = S(s_x, s_y) \cdot P}$$

where S ($s_x$, $s_y$) is the 3 by 3 matrix with parameters $s_x$ and $s_y$.

Replacing these parameters with their multiplicative inverses ($1/s_x$ and $1/s_y$) yields the inverse scaling matrix.

## 2.3 Composite Transformations

- With the matrix representations, we can set up a matrix for any sequence of transformations as a composite transformation matrix by calculating the matrix product of the individual transformations.
- Forming products of transformation matrices is often referred to as a **concatenation, or composition, of matrices.**
- For column-matrix representation of coordinate positions, we form composite transformations by **multiplying matrices in order from right to left**, i.e., each successive transformation matrix pre multiplies the product of the preceding transformation matrices.

### 6.3.1 Translations:

- If two successive translation vectors **($t_{x1}$, $t_{y1}$)** and **($t_{x2}$, $t_{y2}$)** are applied to a coordinate position P, the final transformed location P' is calculated as

$$P' = T(t_{x2}, t_{y2}) \cdot \{T(t_{x1}, t_{y1}) \cdot P\}$$
$$= \{T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1})\} \cdot P\}$$

  where P and P' are represented as homogeneous-coordinate column vectors.

- The composite transformation matrix for this sequence of translations is:

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1}+t_{x2} \\ 0 & 1 & t_{y1}+t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1}) = \mathbf{T}(t_{x1}+t_{x2}, t_{y1}+t_{y2})$$

**Two successive translations are additive.**

### 6.2.2 Rotations:

- Two successive rotations applied to point P produce the transformed position

$$\mathbf{P' = R(\theta 2) \cdot \{R(\theta 1) \cdot P\}}$$
$$\mathbf{= \{R(\theta 2) \cdot R(\theta 1)\} \cdot P}$$

- By multiplying the two rotation matrices, we can verify that **two successive rotations are additive:**

$$\mathbf{R(\theta 2) \cdot R(\theta 1) = R(\theta 1 + \theta 2)}$$

  so that the final rotated coordinates can be calculated with the composite rotation matrix as

$$\mathbf{P' = R(\theta 1 + \theta 2) \cdot P}$$

### 6.2.3 Scaling:

- Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix:

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s) = S(s_{x1} \cdot s_{x2}, s_{y1} \cdot s_{y2})$$

**Successive scaling operations are multiplicative.**

### 6.2.4 General Pivot-Point Rotation:

- With a graphics package that only provides a rotate function for revolving objects about the coordinate origin, we can generate rotations about any selected pivot point **$(x_r, y_r)$** by performing the following sequence of **translate-rotate-translate** operations:

  1. Translate the object so that the pivot-point position is moved to the coordinate origin.
  2. Rotate the object about the coordinate origin.
  3. Translate the object so that the pivot point is returned to its original position.

- The composite transformation matrix for this sequence is obtained with the concatenation

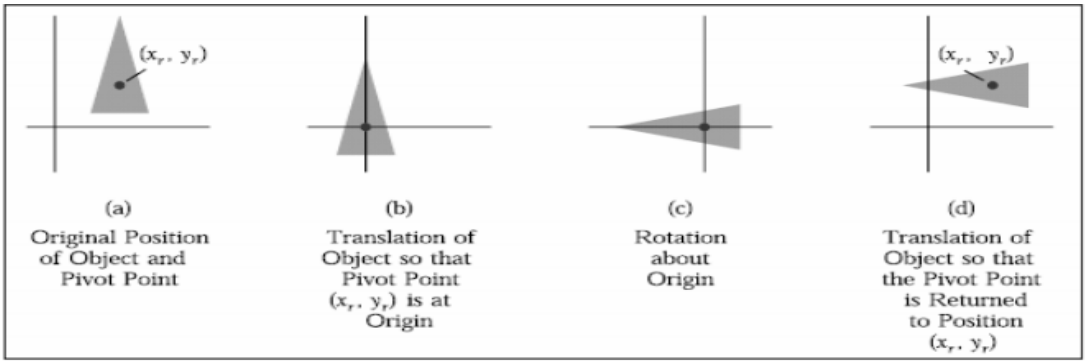
$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

This can be expressed as:

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

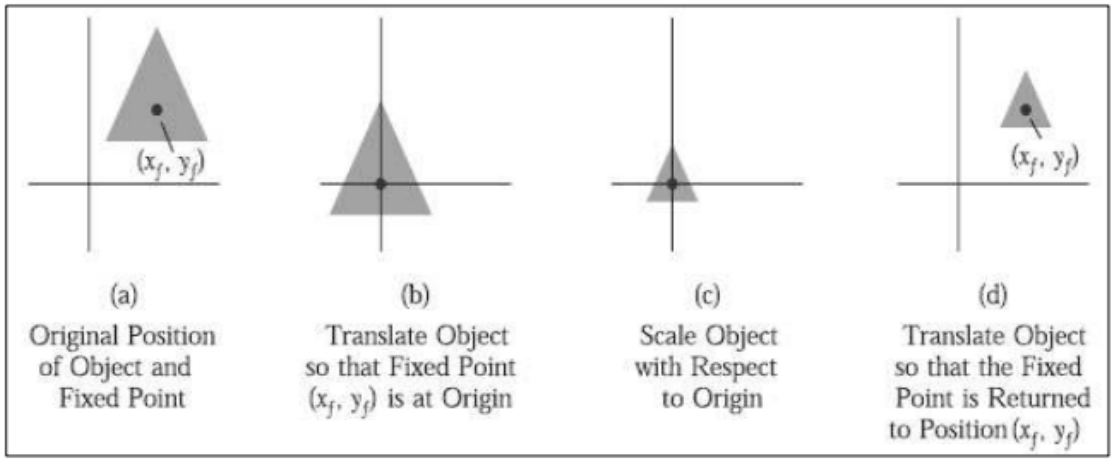
where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$




### 6.2.5 General Fixed-Point Scaling:

- The following diagram illustrates a transformation sequence to produce scaling with respect to a selected fixed position $(x_f, y_f)$ using a scaling function that can only scale relative to the coordinate origin.

  1. Translate object so that the fixed point coincides with the coordinate origin.
  2. Scale the object with respect to the coordinate origin.
  3. Use the inverse translation of step 1 to return the object to its original position



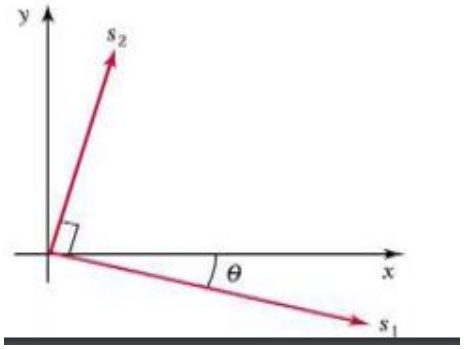
- Concatenating the matrices for these three operations produces the required scaling matrix.

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y)$$

### 6.2.6 General Scaling Directions:

- Parameters $s_x$ and $s_y$ scale objects along the x and y directions. We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.
- If we want to apply scaling factors with values specified by parameters sl and s2 in the directions shown in the diagram, to accomplish the scaling without changing the orientation of the object, we first perform a rotation so that the directions for s1 and s2 coincide with the x and y axes, respectively. Then the scaling transformation is applied, followed by an opposite rotation to return points to their original orientations. The composite matrix resulting from the product of these three transformations is,



$$\mathbf{R}^{-1}(\theta) \cdot \mathbf{S}(s_1, s_2) \cdot \mathbf{R}(\theta) = \begin{bmatrix} s_1 \cos^2\theta + s_2 \sin^2\theta & (s_2 - s_1)\cos\theta\sin\theta & 0 \\ (s_2 - s_1)\cos\theta\sin\theta & s_1 \sin^2\theta + s_2 \cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

### 6.8.7 Concatenation Properties:

- **Matrix multiplication is associative.**

  For any three matrices, A, B and C, the matrix product A . B . C can be performed by first multiplying A and B or by first multiplying B and C:

  **A . B . C = (A . B) . C**

  **= A . (B . C)**

  Therefore, we can evaluate matrix products using either a left-to-right or a right-to-left associative grouping.

- **Transformation products may not be commutative:**

  The matrix product A . B is not equal to B . A. This commutative property holds also for two successive translations or two successive scalings.

  Another commutative pair of operations is rotation and uniform scaling.

  ($s_x = s_y$)

## UNIT-II
## Assignment-Cum-Tutorial Questions
## SECTION-A

### Objective Questions

1. The most basic transformation that are applied in three-dimensional planes are [ ]
   a. Translation b. Scaling c. Rotation d. **All of these**
2. The transformation in which an object can be shifted to any coordinate position in three dimensional plane are called [ ]
   a. **Translation** b. Scaling c. Rotation d. All of these
3. The transformation in which an object can be rotated about origin as well as any arbitrary pivot point are called [ ]
   a. Translation b. Scaling c. **Rotation** d. All of these
4. The transformation in which the size of an object can be modified in x-direction ,y-direction and z-direction [ ]
   a. Translation b. **Scaling** c. Rotation d. All of these
5. Apart from the basic transformation ,_____are also used [ ]
   a.Shearing b.Reflection c. **Both a & b** d. None of these
6. In which transformation ,the shape of an object can be modified in any of direction depending upon the value assigned to them [ ]
   a. Reflection b. **Shearing** c. Scaling d. None of these
7. In which transformation ,the mirror image of an object can be seen with respect to x-axis, y-axis ,z-axis as well as with respect to an arbitrary line [ ]
   a. **Reflection** b. Shearing c. Translation d. None of these
8. A translation is applied to an object by [ ]
   a) Repositioning it along with straight line path
   b) Repositioning it along with circular path
   c) Only b
   d) All of the mentioned
9. We translate a two-dimensional point by adding [ ]
   a) Translation distances b) Translation difference c) X and Y d) None

10. The translation distances (dx, dy) is called as [ ]

a) Translation vector b) Shift vector c) Both a and b d) Neither a nor b

11. In 2D-translation, a point (x, y) can move to the new position (x', y') by using the equation [ ]

a) x'=x+dx and y'=y+dx b) x'=x+dx and y'=y+dy

c) X'=x+dy and Y'=y+dx d) X'=x-dx and y'=y-dy

12. To generate a rotation , we must specify [ ]

a) Rotation angle □ b) Distances dx and dy

c) Rotation distance d) All of the mentioned

13. Positive values for the rotation angle □ defines [ ]

a) Counterclockwise rotations about the end points

b) Counterclockwise translation about the pivot point

c) Counterclockwise rotations about the pivot point

d) Negative direction

14. The original coordinates of the point in polor coordinates are [ ]

a) X'=r cos (Φ +□) and Y'=r cos (Φ +□)

b) X'=r cos (Φ +□) and Y'=r sin (Φ +□)

c) X'=r cos (Φ -□) and Y'=r cos (Φ -□)

d) X'=r cos (Φ +□) and Y'=r sin (Φ -□)

15. The transformation that is used to alter the size of an object is

a) Scaling b) Rotation c) Translation d) Reflection[ ]

16. The two-dimensional scaling equation in the matrix form is

a) P'=P+T b) P'=S*P c) P'=P*R d) P'=R+S [ ]

17. Scaling of a polygon is done by computing

a) The product of (x, y) of each vertex b) (x, y) of end points

c) Center coordinates d) None

## SECTION-B

## SUBJECTIVE QUESTIONS

1. Explain the 2D basic transformations with suitable diagrams.
2. Explain the necessity of homogenous coordinates?
3. Explain reflection and shear?

4. Explain about composite transformations?
5. Perform the following transformations:
   - Scale the image two times in x-direction 5 times in y-direction.
   - Scale the image five times in length 1/5 times in height.
   - Rotate the image $35^0$ in clockwise direction about the horizon.
   - Translate the image 2 units in x-direction and 3 units in y-direction.
   - Translate the image 5 units to the right direction and 3 units up words direction.
   - Translate the image 5 units to the left down words direction and 3 units down words direction.
   - Rotate in anticlock wise direction about 450
6. Give a 3 X 3 homogeneous transformation matrix for the following
   a. Scale the image 5 units in x-direction and 3 units in y –direction.
   b. Scale the image 1/3 units in x-direction and 5 units in y –direction.
   c. Scale the image 3 units in x –direction and no change in y.
   d. Scale the length by 2 units and height by 1/5 unit.
   e. Scale the height by 7 units.
   f. Rotate the image in clockwise direction by 30 degrees.
   g. Rotate the image by 45 degrees in anti-clock wise direction. Prove that two scaling transformations commute that is S 1 * S2 = S2* S1.
7. Prove that two 2 D rotations about origin commute that is R1 * R2 = R2 * R1
8. Find the matrix that represents rotation of an object by 30 degrees about origin and what are the new coordinates of the point P(2,-4) after the rotation.
9. Write the general form of a scaling matrix with respect to a fixed point P (h, k). And using this magnify the triangle with vertices A (0,0), B (1,1) and C(5,2) to twice its size while keeping C(5,2) fixed.
10. Show that transformation matrix for a reflection about a line Y=X is equivalent to reflection to X-axis followed by counter-clock wise rotation of 90 degrees.

11. Perform a 45 degrees rotation of triangle A (0,0) B (1,1) C (5,2) a) about the origin and about P (-1,-1).
12. Give a 3 x 3 transformation matrix for the following:

    a. Translate the image 3 units in X-direction,5 units in Y-direction.
    b. Translate the image 3 units up 7 units left.
    c. Translate the image 5 units right and 4 units downwards.
    d. Translate the image upward direction by 4 units.
    e. Translate image right side by 2 units.
13. Give a single 3 x 3 homogeneous co-ordinate transformation matrix, which will have the same effect as each of the following transformation sequences.
    a. Scale the image to be twice as large and then translate it 1 unit to the left.
    b. Scale the X-direction to be one half as large and then rotate counter clockwise by $\pi/2$ about origin.
    c. Rotate counter clock about the origin by 90 degrees and then scale the X- direction to be one-half as large.
    d. Translate down 1⁄2 unit and then rotate counter clockwise by 45 degrees.
    e. Scale the Y co ordinate to make the image twice as tall, shift down 1 unit and then rotate counter clock wise by 30 degrees.

## UNIT-3

**Objective:**

To familiarize with viewing and clipping

**Syllabus: 2D viewing**

The viewing pipeline, window to viewport coordinate transformation, viewing functions, cohen-sutherland line clipping algorithm, Sutherland-Hodgeman polygon clipping algorithm.

**Outcomes:**

Students will be able to:

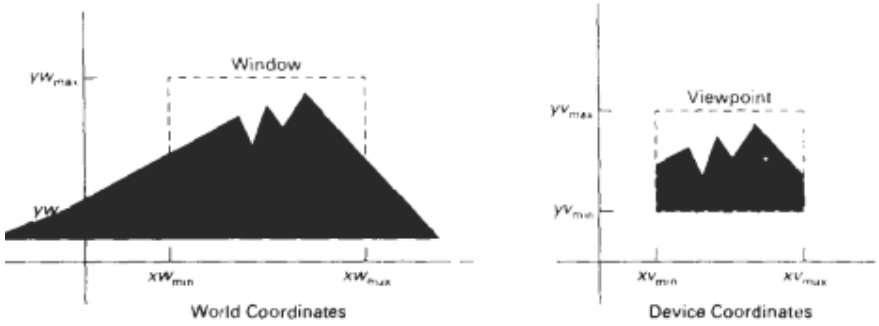
- Understand window to viewport transformation.
- Understand an algorithm for clipping a line.
- Understand the process of clipping a polygon.

**Learning Material**

## The viewing pipeline

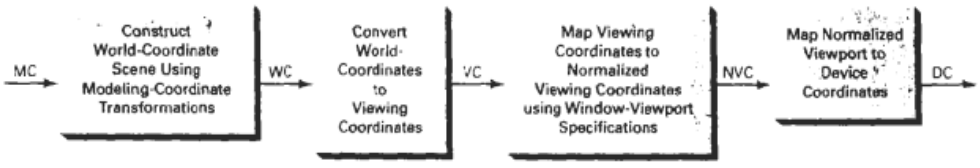
- A world-coordinate area selected for display is called a **window.**
- An area on a display device to which a window is mapped is called a **viewport.**
- The window defines **what is to be viewed**; the viewport defines **where it is to be displayed.**
- Windows and Viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
- In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a **viewing transformation**. It is also called as the **window-to-viewport transformation** or the **windowing transformation**.

- The term window is referred to an **area of a picture that is selected**



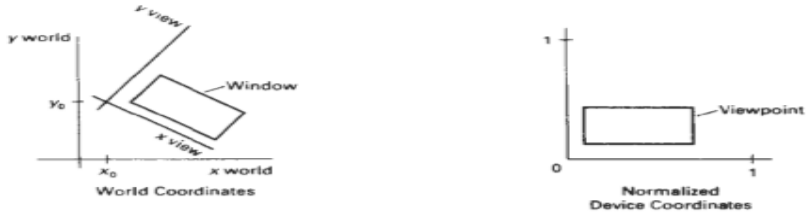

**for viewing**.

- First, we construct the scene in world coordinates using the output primitives and attributes.
- Next to obtain a particular orientation for the window, we set up a two-dimensional **viewing-coordinate system** in the world-coordinate plane, and define a window in the viewing-coordinate system.
- The viewing-coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows.
- Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates.
- We then define a viewport in normalized coordinates (in the range from O to I) and map the viewing-coordinate description of the scene to normalized co-ordinates.
- At the final step, all parts of the picture that are outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates.




- By changing the position of the viewport, we can view objects at different positions on the display area of an output device.

- By varying the size of viewports, we can change the size and proportions of displayed objects.
- We achieve **zooming effects** by successively mapping different-sized windows on a fixed-size viewport. As the windows are made smaller, we zoom in on some part of a scene to view details that are not shown with larger windows. Similarly, more overview is obtained by zooming out from a section of a scene with successively larger windows. **Panning effects** are produced by moving a fixed-size window across the various objects in a scene.




- Viewports are typically defined within the unit square (normalized coordinates). This provides a means for separating the viewing and other transformations from specific output-device requirements, so that the graphics package is largely device independent.
- Once the scene has been transferred to normalized coordinates, the unit square is simply mapped to the display area for the particular output device in use at that time. Different output devices can be used by providing the appropriate device drivers.
- When all coordinate transformations are completed, viewport clipping can be performed in normalized coordinates or in device coordinates. This allows us to reduce computations by concatenating the various transformation matrices.

## Window to viewport transformation

- Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates.
- A point at position ($x_w$, $y_w$) in the window is mapped into position ($x_v$, $y_v$) in the associated viewport. To maintain the same relative placement in the viewport as in the window, we quire that:


$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$
$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

- Solving these expressions for the viewport position ($x_v$, $y_v$), we have:

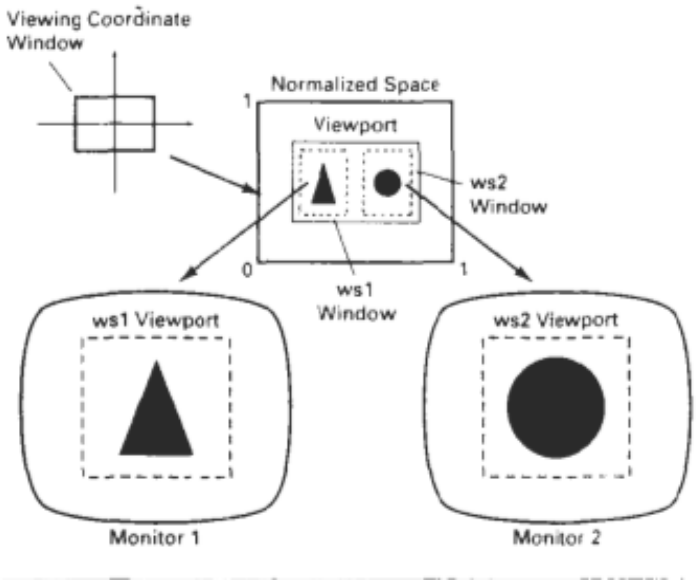
$$xv = xv_{min} + (xw - xw_{min})sx$$
$$yv = yv_{min} + (yw - yw_{min})sy$$

Where the scaling factors are

$$sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$
$$sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

- From normalized coordinates, object descriptions are mapped to the various display devices. Any number of output devices can be open in a particular application, and another

window-to-viewport transformation can be performed for each open output device. This mapping, called the **workstation transformation.**




## Two Dimensional Viweing functions

- We **define a viewing reference system** in a PHIGS application program with the following function:

  **evaluateViewOrientationMatrix ($x_0$, $y_0$, $x_v$, $y_v$,error, viewMatrix)**

  where parameters $x_0$ and $y_0$ are the coordinates of the viewing origin, and parameters $x_v$ and $y_v$ are the world-coordinate positions for the view up vector.

  An integer error code is generated if the input parameters are in error; otherwise, the viewMatrix for the world-to-viewing transformation is calculated. Any number of viewing transformation matrices can be defined in an application.

- To **set up the elements of a window-to-viewport mapping matrix**, we invoke the function:

  **evaluateViewMappingMatrix (xwmin, xwmax,ywmin, ywmax, xvmin, xvmax, yvmin, yvmax, error, viewMappingMatrix)**

Here, the window limits in viewing coordinates are chosen with parameters xwmin, xwmax, ywmin, ywmax. Viewport limits are set with normalized coordinate positions xvmin, xvmax, yvmin, yvmax**.**

- we can store **combinations of viewing and window-viewport mappings** for various workstations in a viewing table with:

  **setViewRepresentation (ws, viewIndex, viewMatrlx, viewMappingMatrix, xclipmin, xclipmax, yclipmin, yclipmax, clipxy)**

  where parameter **ws** designates the output device (workstation), and parameter **viewIndex** sets an integer identifier for this particular window-viewport pair. The matrices **viewMatrix** and **viewMappingMatrix** can be concatenated and referenced by the **viewIndex**. The parameter **clipxy** is assigned either the value **noclip** or the value **clip**. This allows us to turn off clipping if we want to view the parts of the scene outside the viewport. We can also select **noclip** to speed up processing when we know that all of the scene is included within the viewport limits.

- The below function selects a particular set of options from the viewing table. This view-index selection is then applied to subsequently specified output primitives and associated attributes and generates a display on each of the active workstations.

  **setViewIndex(viewIndex)**

- we apply a **workstation transformation** by selecting a workstation window-viewport pair:

  **setWorkstationWindow (ws, xwswindmin, xwswindmax, ywswindrnin, ywswindmax)**

  **setWorksrationViewport (ws, xwsVPortmin, xwsVPortmax, ywsVPortmin, ywsVPortmax)**

where parameter **ws** gives the workstation number. Windowioordinate extents are specified in the range from 0 to 1 (normalized space), and viewport limits are in integer device coordinates.

## Clipping

- Any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is called **"clipping algorithm",** or simply **"clipping".**
- The region against which an object is to clipped is called a **"clip window".**
- Applications of clipping include:
  - extracting part of a defined scene for viewing;
  - identifying visible surfaces in three-dimensiona1 views;
  - ant aliasing line segments or object boundaries;
  - creating objects using solid-modelling procedures;
  - displaying a multi window environment;
  - drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating
- Depending on the application, the clip window can be a general polygon or it can even have curved boundaries.
- For the viewing transformation, we want to display only those picture parts that are within the window area. Everything outside the window is discarded.
- Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates.
- The following clipping procedures:

- Point Clipping
- Line clipping (straight-line segments)
- Area clipping (polygons)
- Curve Clipping
- Text Clipping

## Point Clipping

- Assuming that the clip window is a rectangle in standard position, we save a point P = (x, y) for display if the following inequalities are satisfied:

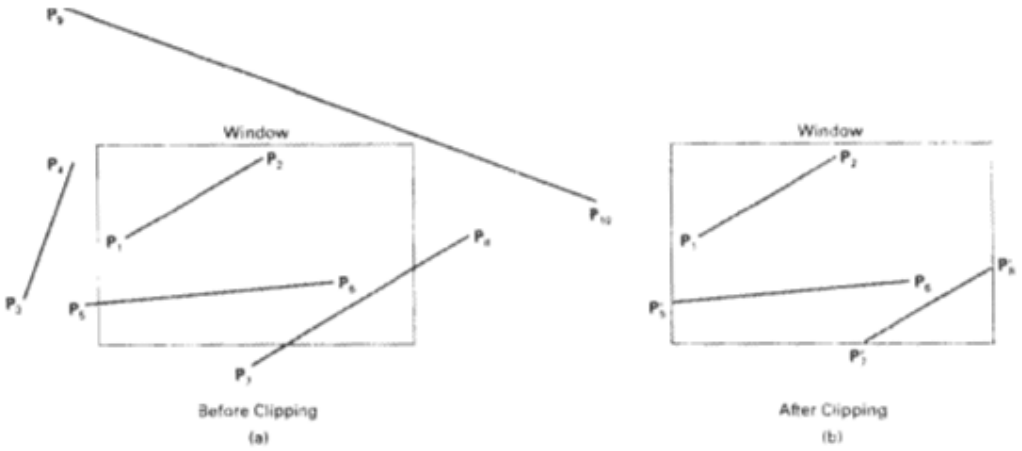
$$xw_{min} \le x \le xw_{max}$$
$$yw_{min} \le y \le yw_{max}$$

where the edges of the clip window ($xw_{min}$, $yw_{min}$, $xw_{max}$, $yw_{max}$) can be either the world-coordinate window boundaries or viewport boundaries.

If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

## Line Clipping

- A line clipping procedure involves several parts:
- First, we can test a given line segment to determine whether it lies completely inside the clipping window. If it does not, we try to determine whether it lies completely outside the window. Finally, if we cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries.
- We process lines through the "inside-outside" tests by checking the **line endpoints**.




- A line with both endpoints inside all clipping boundaries, such as the line from $P_1$ to $P_2$ is saved.
- A line with both endpoints outside any one of the clip boundaries, such as $P_3$ to $P_4$ is outside the window.
- All other lines cross one or more clipping boundaries, and may require calculation of multiple intersection points.
- For a line segment with endpoints $(x_1, y_1)$ and $(x_2 y_2)$ and one or both endpoints outside the clipping rectangle, the parametric representation:

$$x = x_1 + u(x_2 - x_1)$$
$$y = y_1 + u(y_2 - y_1), \quad 0 \le u \le 1$$

- The above representation could be used to determine values of parameter **u** for intersections with the clipping boundary coordinates.
- If the value of u for an intersection with a rectangle boundary edge is **outside the range 0 to 1**, the line does not enter the interior of the window at that boundary.
- If the value of u is **within the range from 0 to 1**, the line segment may cross into the clipping area. This method can be applied to each clipping boundary edge in turn to determine whether any part of the

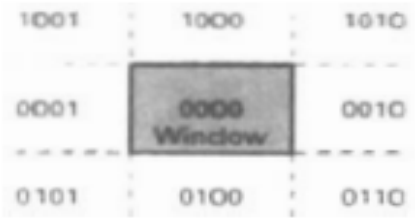
line segment is to be displayed. Line segments that are parallel to window edges can be handled as special cases.

## Cohen Sutherland Line Clipping Algorithm

- This is the oldest and most popular line-clipping procedures.
- The method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must he calculated.

**Region Codes:**

- Every line end point in a picture is assigned a four-digit binary code, called a "**region code"**, that identifies the location of the point relative to the boundaries of the clipping rectangle.
- Regions are set up in reference to the boundaries as shown below:




- Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the clip window: to the left, right, top, or bottom.
- By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions can be correlated with the bit positons as:

**bit 1: left**
**bit 2: right**
**bit 3: below**
**bit 4: above**

- A value of 1 in any bit position indicates that the point is in that relative position; otherwise, the bit position is set to 0. If a point is within the clipping rectangle, the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

- Bit values in the region code are determined by comparing endpoint coordinate values (x, y) to the clip boundaries:

**Bit 1 is set to 1 if $x < xw_{min}$**
**Bit 2 is set to 1 if $x > xw_{max}$**
**Bit 3 is set to 1 if $y < yw_{min}$**
**Bit 4 is set to 1 if $x > yw_{max}$**

- Region-code bit values can be determined with the following two steps:

(1) Calculate differences between endpoint coordinates and clipping boundaries.

(2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

**Bit 1 is the sign bit of $x – xw_{min}$**
**Bit 2 is the sign bit of $xw_{max} - x$;**
**Bit 3 is the sign bit of $y – yw_{min}$**
**Bit 4 is the sign bit of $yw_{max} - y$.**

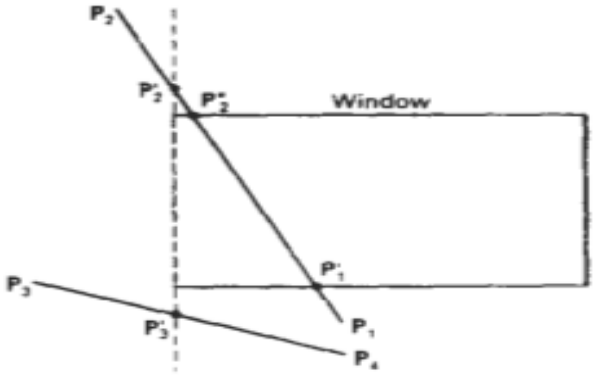
**Processing the Lines using the region codes:**

- Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside.
- Any lines that are completely contained within the window boundaries have **a region code of 0000 for both endpoints, they are completely inside the clipping rectangle and we accept these lines.**
- **Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we reject these lines.** We would discard the line that

has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code.

- A method that can be used to test lines for total clipping is to **perform the logical AND** operation with both region codes.
  - If the result is not 0000, the line is completely outside the clipping region.
  - If the result is 0000, the line is intersecting the clipping boundary.
- We begin the clipping process for a line by comparing an outside endpoint to a clipping boundary to determine how much of the line can be discarded. Then the remaining part of the line is checked against the other boundaries, and we continue until either the line is totally discarded or a section is found inside the window. We set up our algorithm to check line endpoints against clipping boundaries in the order left, right, bottom, top.

**Example:**



- Starting with the bottom endpoint of the line from $P_1$ to $P_2$, we check $P_1$ against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle.
- We then find the intersection point $P_1^1$, with the bottom boundary and discard the line section from $P_1$ to $P_1^1$. The line now has been reduced to the section from $P_1^1$ to $P_2$.
- Since $P_2$ is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point $P_2^1$ is calculated, but this point is above the window. So the final intersection calculation yields $P_2^{11}$ and the line from $P_1^1$ to $P_2^{11}$ is saved.
- This completes processing for this line, so we save this part and go on to the next line.
- Point $P_3$ in the next line is to the left of the clipping rectangle, so we determine the intersection $P_3^1$ and eliminate the line section from $P_3$ to $P_3$' .By checking region codes for the line section from $P_3^1$ to $P_4$, we find that the remainder of the line is below the clip window and can be discarded also.
- Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates $(x_1, y_1)$ and $(x_2, y_2)$, the y coordinate of the intersection point with a vertical boundary can be obtained with the calculation:

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

where the **x value is set either to $xw_{min}$ or to $xw_{max}$** , the slope of the line is calculated as

**$m = (y_2 - y_1)/(x_2 - x_1)$**

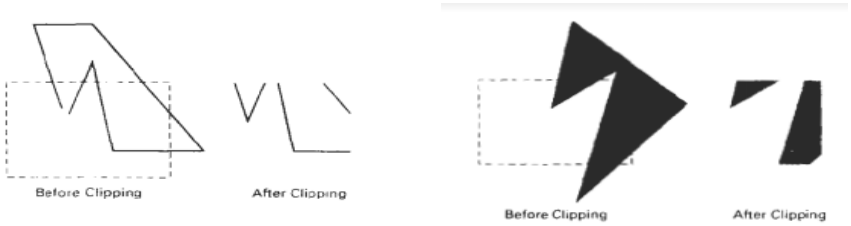
- For the intersection with a horizontal boundary, the x coordinate can be calculated as:

$$x = x_1 + \frac{y - y_1}{m}$$

with **y set either to $yw_{min}$ or to $yw_{max}$**.

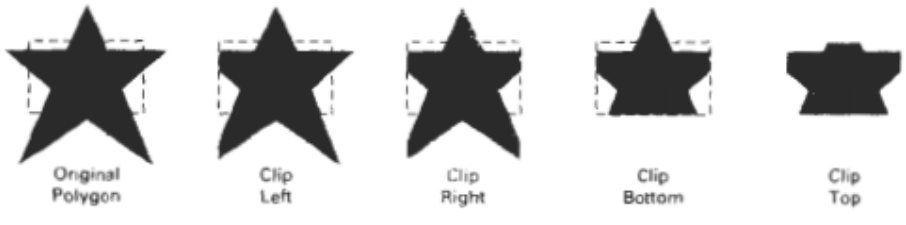
## Polygon Clipping

- For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.




## Sutherland-Hodgeman polygon Clipping

- We can correctly clip a polygon by processing the polygon boundary as a whole against each window edge.
- This could be accomplished by processing all polygon vertices against each clip rectangle boundary in turn.
- Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then be successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper.



- At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.
- There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests:

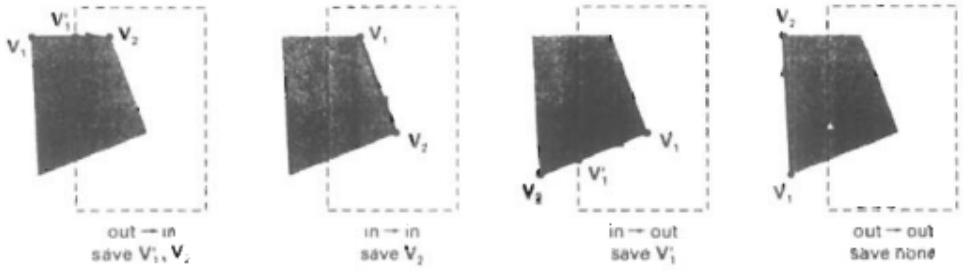
  (1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list.

  (2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.

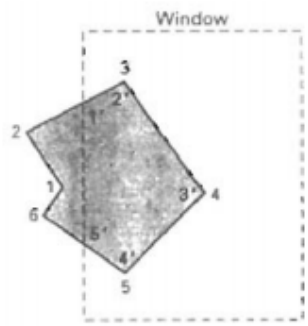
  (3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list.

  (4) If both input vertices are outside the window boundary, nothing is added to the output list.

- Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.




Let us consider the following figure:




- Vertices 1 and 2 are found to be on the outside of the boundary. So none is saved.
- Moving along 2 and 3, intersection point $1^1$ and the vertex 3 i.e., $2^1$ is saved.
- Vertex 4 is saved as $3^1$.
- Vertex 5 is saved as $4^1$.
- Moving from vertex 5 to vertex 6, the intersection point $5^1$ is saved.
- Now using the points $1^1$-$2^1$-$3^1$-$4^1$-$5^1$ , we would repeat the process for the next window boundary.
- The output list of vertices is stored as the polygon is clipped against each window boundary. We can eliminate the intermediate output vertex lists by simply clipping individual vertices at each step and passing the clipped vertices on to the next boundary.
- A point (vertex or intersection) is added to the final output vertex list only after it has been determined to be inside or on a window boundary by all four boundary clippers.

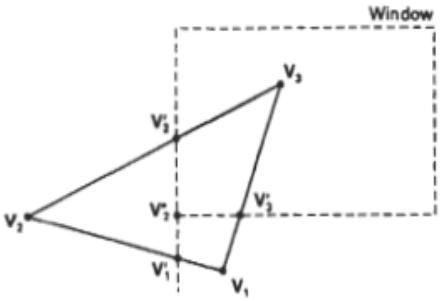
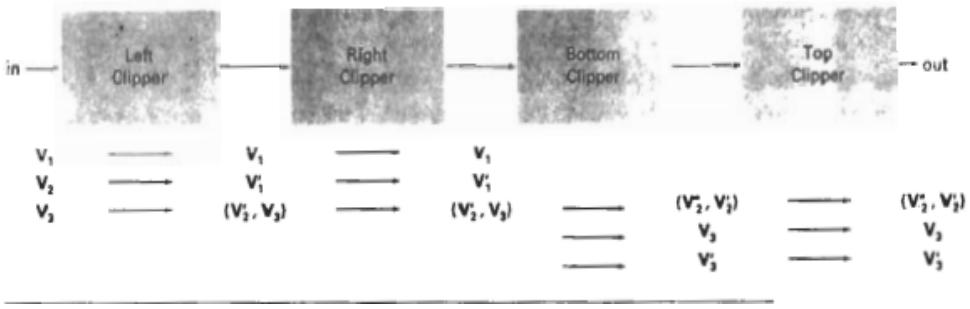



*Figure 6-22*
A polygon overlapping a rectangular clip window.



## UNIT-III
## Assignment-Cum-Tutorial Questions
## SECTION-A

### Objective Questions

1.The rectangle portion of the interface window that defines where the image will actually appear are called [ ]

a) Transformation viewing c) View port

b) Clipping window d) Screen coordinate system

2. The rectangle space in which the world definition of region is displayed are called [ ]

a) Screen coordinate system c) Clipping window or world window

b) World coordinate system d) None of these

3. The object space in which the application model is defined [ ]

a) Screen coordinate system c) Clipping window or world window

b) World coordinate system d) None of these

4. The process of cutting off the line which are outside the window are called [ ]

a)Shear b ) Reflection c) Clipping d) Clipping window

5. The process of mapping a world window in world coordinate system to viewport are called [ ]

a) viewing transformation c) Clipping window

b) Viewport d) Screen d) coordinate system

6. Coordinates of window are known as ..... [ ]

a) Screen coordinates c) World coordinates

b) Device coordinates d) Cartesian coordinates

7. A method used to test lines for total clipping is equivalent to the__?

a) logical XOR operator c) logical OR operator [ ]

b) logical AND operator d) both a and b

8. .....clips convex polygons correctly , but in case of concave polygon , it displays an extraneous line . [ ]

a) sutherland-hodgeman algorithm b) Cohen –Sutherland algorithm

c )none of above d) either ( a ) or (b)

9. The Cohen-Sutherland line clipping algorithm divides the entire region into .....number of sub-regions [ ]

a) 4 b) 8 c) 9 d) 10

10. . ..... number of bits are used for representing each sub-region of Cohen-Sutherland line clipping algorithm : [ ]

a) 1 b) 2 c) 3 d) 4

11. The region against which an object is clipped is called a ..... [ ]

a) Clip window b) Boundary c) Enclosing rectangle d)Clip square

12.A line with endpoints codes as 0000 and 0100 is ? [ ]

a) Partially invisible b) Completely visible

c) Completely invisible d) Trivially invisible

13. According to Cohen-Sutherland algorithm, a line is completely outside the window if ..... [ ]

a) The region codes of line endpoints have a '1' in same bit position.

b)The endpoints region code are nonzero values

c) If L bit and R bit are nonzero.

d) The region codes of line endpoints have a '0' in same bit position.

14. The result of logical AND operation with endpoint region codes is a nonzero value. Which of the following statement is true? [ ]

a) The line is completely inside the window

b) The line is completely outside the window

c) The line is partially inside the window

d) The line is already clipped

15. The left (L bit ) bit of the region code of a point (X,Y) is '1' if .....[ ].

a) X > XWMIN b) X< XWMIN c) X< XWMAX d) X>XWMAX

16.The Most Significant Bit of the region code of a point (X,Y) is '1' if .....

a) Y > YWMIN b) Y< YWMIN c) Y< YWMAX d) Y>YWMAX[ ]

17. In a clipping algorithm of Cohen & Sutherland using region codes, a line is already clipped if the? [ ]

a) codes of the end point are same
b) logical AND of the end point code is not 0000
c) logical OR of the end points code is 0000
d) logical AND of the end point code is 0000
e) A and B

18. In displaying a clipped picture the efficient method is ? [ ]

a. Clipping against the window and then applying the window transformation
b. Applying window transformation and then clipping against the viewport
c. Both A and B have the same efficiency
d. Efficiency depends on whether the window is an aligned rectangle or not

19.In the Cohen-Sutherland line clipping algorithm, if codes of the two points P and Q are 0101 and 0001 then the line segment joining the points P and Q will be the clipping window [ ]

(a) Totally outside (b) Partially outside
(c) Totally inside (d) None of the above

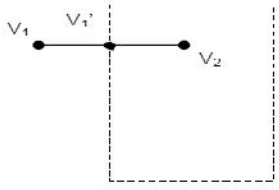
20. 1f XL, .XR, YB, YT represent the four parameters of x-left, x—right, y—bottom and ytop of a clipping window and (x, y) is a point inside the window such that x > XL and x > XL and YB < y < YT, then the code of the point (x, y) in Cohen—Sutherland algorithm

(a) 1100 (b) 1000 (c) 1110 (d) 0000 [ ]

21. For the figure given below what are the new vertices to be saved as output vertices, after clipping with the window boundary [ ]

a) V1', V2

b) V2



c)V1' d) none

22. In Sutherland-Hodgeman algorithm for polygon clipping, assume P (Present point) lies inside the window and S(Previous point) lies outside the window. Then, while processessing through that window boundary, we should [ ]

a) Store the intersection point of the line Ps (S') only

b) store the point P and S'

c) store the point P only

d) store the points S and S'

23.Perform window to viewport transformation for the point (20,15) Assume that (Xwmin, Ywmin) is (0,0), (Xwmax, Ywmax) is (100, 100); (Xvmin, Yvmin) is (5,5); (Xvmax, Yvmax) is (20, 20). The value of x and y the viewport [ ]

a) x=4, y=4 b) x=3, y=3 c) x=8, y=7.25 d) x=3, y=4

## SECTION-B

## SUBJECTIVE QUESTIONS

1) What is window-to-viewport (viewing) transformation. Explain the steps involved in it.
2) What are the stages involved in 2D viewing transformation pipeline. Explain briefly about each stage.
3) Give a brief note on 2d viewing functions. Give an example which uses 2D viewing functions.
4) Explain the Cohen-Sutherland line clipping algorithm . Demonstrate with an example all the three cases of lines.
5) Why the Sutherland-Hodgeman is called as re-entrant algorithm. Describe Sutherland-Hodgeman algorithm for polygon clipping.

6) What are the advantages and disadvantages of Cohen-Sutherland out-code algorithm
7) Distinguish between Cohen-Sutherland and Sutherland Hodgeman algorithms
8) Justify that the Sutherland-Hodgeman algorithm is suitable for clipping concave polygons also.

# UNIT-IV

## 3D Geometric Transformations

**Objective:** To familiarize with the 3D Transformation, viewing, clipping and various Projections **.**

**Syllabus**

Translation, Scaling ,Rotation, Reflection ,Shear Transformations ,Composite Transformations, Parallel Projections ,Perspective projections

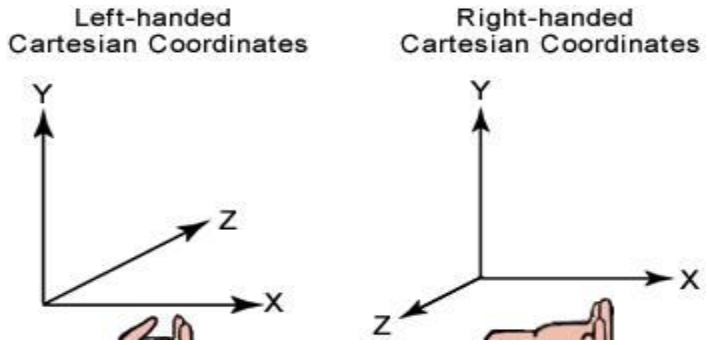
**Learning Outcomes**:

Students will be able to

- Differentiate between 2D and 3D Graphic objects representation
- Perform various 3D-Transformations like translation, scaling, rotation, reflection and shearing
- Know the Importance of 3D –Projections in Engineering Applications

### 4.1.Introduction.

- In the 2D system, we use only two coordinates X and Y but in 3D, an extra coordinate Z – axis is added. 3D graphics techniques and their application are fundamental to the entertainment, games, and computer-aided design industries
- Two Coordinate system are available to use in 3D.
  - Left Handed Coordinate System and
  - Right Handed Coordinate System
- Commonly we will use the Right Coordinate System.



## 4.2 TRANSLATION :

We now translate an object by specifying a three-dimensional translation vector, which determines how much the object is to be moved in each of the three coordinate directions.

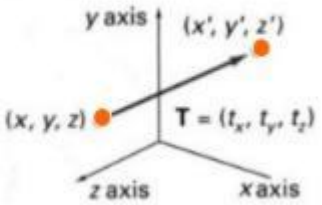
**Translation of a point :**

In a three-dimensional homogeneous coordinate representation, a point is translated (Fig.**)** from position P = ***(x,* y, *z)*** to position P' = ***(x',* y',** z') with the matrix Operation.

- This can be written as:-

Using **P' = T . P**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$




Parameters *tx, ty and tz* specifes translation distances for the coordinate directions ***x,* y,** and z, and they can be assigned any real values. The matrix representation in Eq. 11-2 is equivalent to the below three equations

**X' =x+tx**

**Y'=y+ty**

**Z'=z+tz**

**Translation of object :** An object is translated in three dimensions by transforming **each of the defining points** of the object. For an object represented as a set of polygon surfaces, we translate each vertex of each surface and redraw the polygon facets in the new position.

**Inverse Translation:**

We obtain the inverse of the translation matrix in Eq. 11-1 by ***negating the translation distances*** tx, ty, and tz. This produces a translation in the opposite direction.

**$T^{-1}$. T = Identity Matrix : T**he product of a translation matrix and its inverse produces the identity matrix.

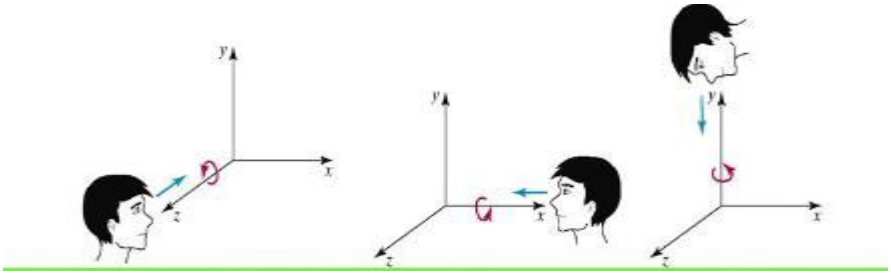
## 4.3 ROTATION

- To generate a rotation transformation for an object, we must designate
  - an **axis of rotation** (about which the object is to be rotated) and
  - the **amount of angular rotation**.

- Unlike two-dimensional applications, where all transformations are carried out in the **xy** plane, a three-dimensional rotation can be specified around **any line** in space.

- The easiest rotation axes to handle are those that are **parallel** to the coordinate axes. Also, we can use combinations of coordinate axis rotations (along with appropriate translations) to specify any general rotation.

- By convention, positive rotation angles produce counter-clockwise rotations about a coordinate axis, if we are looking along the positive half of the axis toward the coordinate origin (Fig. below).



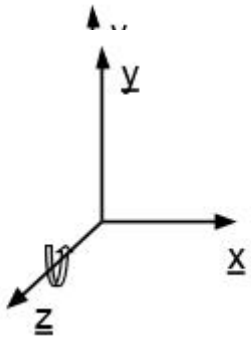
- Therefore, positive rotations in the **xy** plane are counter-clockwise about axes parallel to the z axis.

  Rotations can be performed w.r.t below considerations :

- coordinate axis rotation
- rotation about an axis parallel to one of the coordinate axis
- rotation about an arbitary axis.

### 4.3.1 Coordinate-Axes Rotations

**ROTATION ABOUT Z-AXIS :**




- The two-dimensional z-axis rotation equations are easily extended to three dimensions:

X'=xcosθ-ysinθ [ eq's 11.4]
Y'=xsinθ+ycosθ
Z'=z

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

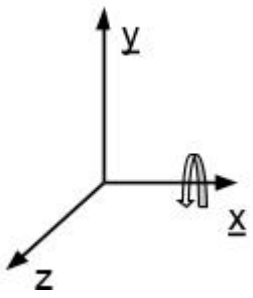
the above matrix can be compactly written as **$P' = R_Z(\theta).P$**

Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters *x,* y, and z in Eqs. 11-4. That is, we use the replacements **x-> y -> z -> x**

Substituting these permutations in our equations we get

**ROTATION ABOUT X-AXIS as :**

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$




Y'=ycosθ-zsinθ [eq. 11.8]

Z'=ysinθ+zcosθ
X'=x
**$P' = R_x(\theta).P$**

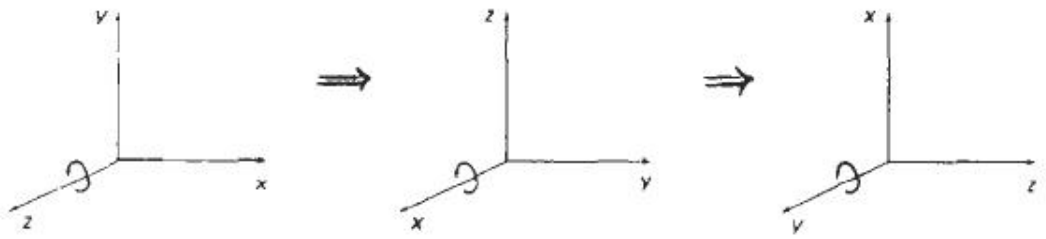


Figure 11-5
Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

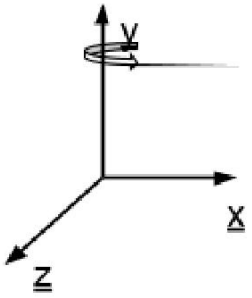
**ROTATION ABOUT Y-AXIS**

Cyclically permutating eq.11.8

$z' = z\cos\theta - x\sin\theta$
$x' = z\sin\theta + x\cos\theta$
$y' = y$

The matrix representation for y-axis rotation is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



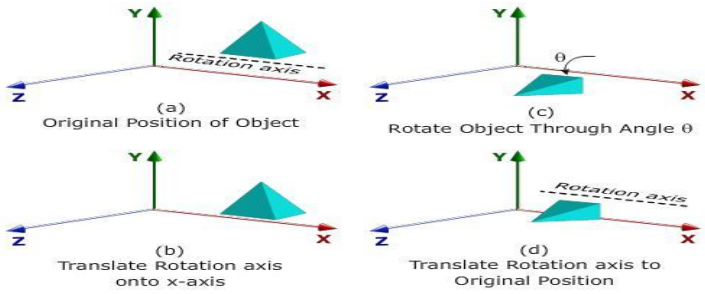

**Inverse Rotation Matrix :**

- An inverse rotation matrix is formed by replacing the rotation angle θ by -θ.
- Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produced when any rotation matrix is multiplied by its inverse. Since only the sine function is affected by the change in **sign** of the rotation angle, the inverse matrix can also be obtained by **interchanging** rows and columns.

- That is, we can calculate the inverse of any rotation matrix R by evaluating its **transpose** ($R^{-1} = R^T$). This method for obtaining an inverse matrix holds also for any composite rotation matrix.

## 4.3.2 GENERAL THREE DIMENSIONAL ROTATIONS

- **A** rotation matrix for any axis that does ***not coincide with a coordinate axis*** can be set up as a composite transformation involving **combinations of translations** and the **coordinate-axes rotations**.
- When an object is to be rotated about an ***axis that is parallel to one of the coordinate axes***, we can attain the desired rotation with the following transformation sequence.
  - Translate the object so that the rotation axis coincides with the parallel coordinate axis.
  - Perform the specified rotation about that axis.
  - Translate the object so that the rotation axis is moved back to its original position.




**Sequence of Transformations for Rotating an object about an axis that is Parallel to the x-axis**

Any coordinate position **P** on the object in this figure is transformed with the sequence shown as

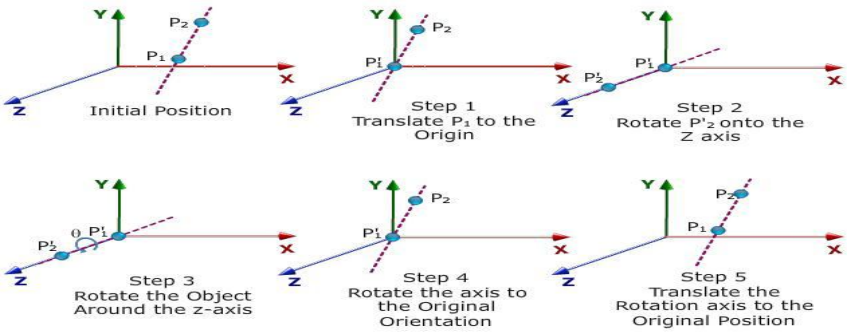
$$P' = [T^{-1}. R_x(\theta) . T]. P$$

where the composite matrix for the transformation is, $R(\theta) = T^{-1}. R_x(\theta) . T$, which is of the same form as the two-dimensional transformation sequence for rotation about an arbitrary pivot point.

### 4.3.3 ***Rotation about an Arbitrary Axes***

- When an object is to be rotated about an axis that **is *NOT PARALLEL TO ONE OF THE COORDINATE AXES*** : we need to perform some additional transformations.
- In this case, we also need **rotations, to align the axis** with a selected coordinate axis and to bring the axis back to its original orientation.
- Given the specifications for the rotation axis and the rotation angle, we can accomplish the required rotation in **FIVE STEPS** :
    1. Translate the object so that the rotation axis pass through the coordinate origin.
    2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
    3. Perform the specified rotation about that coordinate axis.
    4. Apply inverse rotation to bring the rotation axis back to its original orientation.
    5. Apply the inverse translation to bring the rotation axis back to its original position.

Note : We can transform the rotation axis onto any of the three coordinate axes. **The z axis is a reasonable choice**.




**Five Transformation steps for obtaining a Composite Matrix for Rotation about an Arbitrary axis, with the Rotation axis Projected on to the z-axis**

**How to set up the transformation matrices for getting the rotation axis onto the z axis and returning the rotation axis to its original position** (Fig. 11-9).

- A rotation axis can be defined with two coordinate positions, as in Fig. 11-0,

OR

- with one coordinate point and direction angles (or direction cosines) between the rotation axis and two of the coordinate axes.

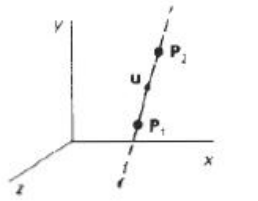



*Figure 11-10*
An axis of rotation (dashed line) defined with points $P_1$ and $P_2$. The direction for the unit axis vector u is determined by the specified rotation direction.

- We will assume that
  - ***the rotation axis*** is defined by 2 points, as illustrated, and
  - that the ***direction of rotation*** is to be counter-clockwise when looking along the axis from P2 to P1.

- An ***AXIS VECTOR*** is then defined by the two points as
  **V = P2 - P1 = (x2-x1, y2-y1, z2-z1)**

- A **UNIT VECTOR u** is then defined along the axis as

$$\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = (a, b, c) \qquad (11\text{-}15)$$

$$a = \frac{x_2 - x_1}{|\mathbf{v}|}, \quad b = \frac{y_2 - y_1}{|\mathbf{v}|}, \quad c = \frac{z_2 - z_1}{|\mathbf{v}|} \qquad (11\text{-}16)$$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (11\text{-}17)$$

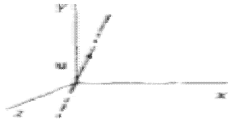



*Figure 11-11*
Translation of the rotation axis to the coordinate origin.

- If the rotation is to be in the opposite direction (clockwise when viewing from P2 to PI), then we would reverse axis vector V and unit vector u so that they point from P2 to P1.
- The first step in the transformation sequence for the desired rotation is to set up the translation matrix that repositions the rotation axis so that it passes through the coordinate origin.
- For the desired direction of rotation (Fig. 11-10), we accomplish this by moving point P1 to the origin. (If the rotation direction had been specified in the opposite direction, we would move P2 to the origin.) This translation matrix is.

which repositions the rotation axis and the object, as shown in Fig. 11-11.

- Now we need the transformations that will put the rotation axis on the ***z*** axis.
- We can use the coordinate-axis rotations to accomplish this alignment in two steps. There are a number of ways to perform the two steps.
  - We will first rotate about the ***x*** axis to transform vector u into the **xz** plane.
  - Then we swing **u** around to the z axis using a y-axis rotation.

These two rotations are illustrated in Fig. 11-12 for one possible orientation of vector **u.**

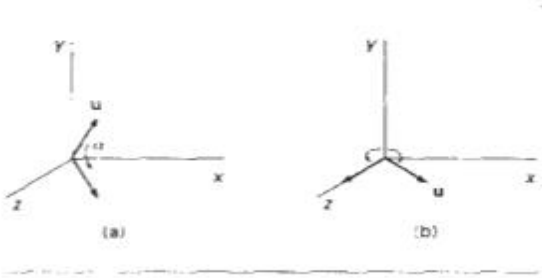



*Figure 11-12*
Unit vector **u** is rotated about the *x* axis to bring it into the *xz* plane (a), then it is rotated around the *y* axis to align it with the *z* axis (b).

- Since rotation calculations involve sine and cosine functions, we can use standard vector operations to obtain elements of the two rotation matrices.
- **Dot-product** operations allow us to determine the ***cosine terms***, and
- vector **cross products** provide a means for obtaining the ***sine terms***.
- We establish the transformation matrix for rotation around the **x** axis by determining the values for the sine and cosine of the rotation angle necessary to get u into the **xz** plane.
- This rotation angle is the angle between the projection of u in the yz plane and the positive ***z*** axis (Fig. *11-13),*
- If we designate the projection of u in the yz plane as the vector **u'** = (0, b, **c),** then the cosine of the rotation angle $\alpha$ can be determined from the dot product of u' and the unit vector **uz** along the ***z***-axis.

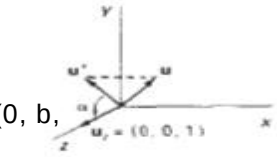



*Figure 11-13*
Rotation of u around the *x* axis into the *xz* plane is accomplished by rotating u' (the projection of u in the *yz* plane) through angle $\alpha$ onto the *z* axis.

$$\cos\alpha = \frac{u' \cdot u_z}{|u'||u_z|}$$

$$= c / d$$

where d is the magnitude of u' : $d = \sqrt{b^2 + c^2}$

Similarly, we can determine the sine of **α** from the cross product of u' and **uz.**

The coordinate-independent form of this cross product is

$$u' \times u_z = u_x |u'||u_z| \sin\alpha$$

and the **Cartesian form for the cross product** gives us

$$u' \times u_z = u_x \cdot b$$

Equating the RHS of both Cross Product values and noting that $U_Z$=1 and U'=d we have

$$d \sin\alpha = b$$

$$\sin\alpha = b/d$$

Now that we have determined the values for cosa and sina in terms of the components of vector u, we can set up the matrix for rotation of u about the ***x*** axis:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & -b/d & 0 \\ 0 & b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

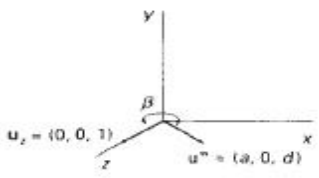



*Figure 11-14*
Rotation of unit vector u'' (vector u after rotation into the *xz* plane) about the *y* axis. Positive rotation angle $\beta$ aligns u'' with vector $u_z$.

This matrix rotates unit vector **u** about the **x** axis into the **xz** plane.

**Next** we need to determine the form of the transformation matrix that will swing the unit vector in the *xz* plane counterclockwise around they axis onto the positive ***z*** axis. The orientation of the unit vector in the ***xz*** plane (alter rotation about the ***x*** axis) is shown in Fig. 11-14.

This vector, labeled u", has the value a for its ***x*** component, since rotation about the ***x*** axis leaves the ***x*** component unchanged.

Its *z* component is d (the magnitude of u'), because vector **u'** has been rotated onto the z axis. And they component of u" is 0, because it now lies in the **xz** plane.

Again, we can determine the cosine of rotation angle ***P*** from expressions for the dot product of unit vectors **u"** and $\mathbf{u}_z$**:**

$$\cos\beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''|\,|\mathbf{u}_z|} = d \qquad (11\text{-}24)$$

since $|\mathbf{u}_z| = |\mathbf{u}''| = 1$. Comparing the coordinate-independent form of the cross product

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y |\mathbf{u}''|\,|\mathbf{u}_z| \sin\beta \qquad (11\text{-}25)$$

with the Cartesian form

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y \cdot (-a) \qquad (11\text{-}26)$$

we find that

$$\sin\beta = -a \qquad (11\text{-}27)$$

Thus, the transformation matrix for rotation of $\mathbf{u}''$ about the *y* axis is

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (11\text{-}28)$$

With transformation matrices 11-17, 11-23, and 11-28, we have aligned the rotation axis with the positive *z* axis. The specified rotation angle $\theta$ can now be applied as a rotation about the *z* axis:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The transformation matrix for rotation about an arbitrary axis then can be expressed as the composition of the seven individual transformations

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha) \cdot \mathbf{T}$$

## 4.4 SCALING

The matrix expression tor the scaling transformation of a position P = **(x,** y, *z)* relative to the coordinate origin can be written as

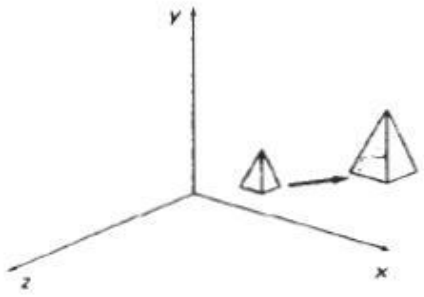



*Figure 11-17*
Doubling the size of an object with transformation 11-42 also moves the object farther from the origin.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad (11\text{-}42)$$

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \qquad (11\text{-}43)$$

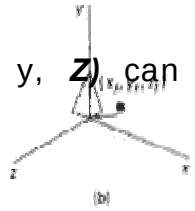
where scaling parameters sx, sy**,** and sz are assigned any positive values.

- Explicit expressions for the coordinate transformations for scaling relative to the origin are x'= x. $s_x$
  - y'=y.$s_y$
  - z'=z.$s_z$
- Scaling an object with transformation **11-42** changes the size of the object and repositions the object relative to the coordinate origin.
- Also, if the transformation parameters are not all equal, relative dimensions in the object are changed: We preserve the original shape of an object with a uniform scaling **(sx**=sy = ***sz).***

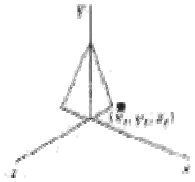
- The result of scaling an object uniformly with each scaling parameter set to 2 is shown in Fig

**4.4.1 SCALING RELATIVE TO FIXED POINT :**

- Scaling with respect to a selected fixed position **(x,** y, ***Z)*** can be represented with the following transformation sequence:



1. Translate the fixed point to the origin.
2. Scale the object relative to the coordinate origin using Eq.
3. Translate the fixed point back to its original position.



- This sequence of transformations is demonstrated in Fig.

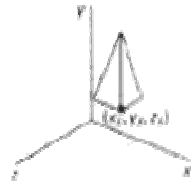


Figure 11-18
Scaling an object relative to a selected fixed point is equivalent to the sequence of transformations shown

The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations as

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**Inverse Scaling Matrix** by re-placing the scaling parameters sx, sy and sz***:*** with their reciprocals.

- The inverse matrix generates an opposite scaling transformation, so the concatenation of any scaling matrix and its inverse produces the identity matrix.

## 4.5 REFELCTIONS

- A three-dimensional retlection can be performed relative to a selected ***reflection axis*** or with respect to a selected ***reflection plane***.
- In general, three-dimensional reflection matrices are set up similarly to those for two dimensions.
- Reflections **relative to a given axis** are equivalent to 180 rotations about that axis.
- Reflections with **respect to a plane** are equivalent to 180 rotations in **four-dimensional space**.
- When the reflection plane is a coordinate plane (either xy, *xz, or* yz), we can think of the transformation as a conversion between Left-handed and right-handed systems.
- An example of a reflection that converts coordinate specifications from a right-handed system to a left-handed system (or vice versa) is shown in Fig. 11-19

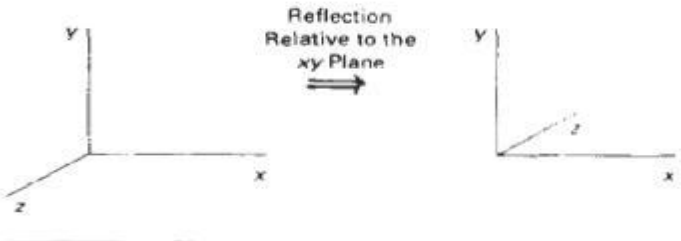



*Figure 11-19*
Conversion of coordinate specifications from a right-handed to a left-handed system can be carried out with the reflection transformation 11-46.

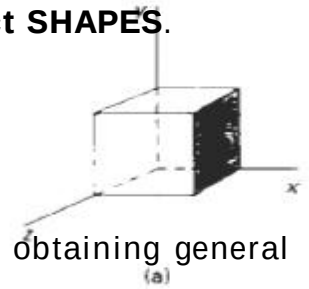
- This transformation changes the sign of the z coordinates, leaving the *x* and y-coordinate values unchanged.
- The matrix representation for this reflection of points relative to the xy plane is

$$RF_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Transformation matrices for inverting **x** and y values are defined similarly, as reflections relative to the yz plane and ***xz*** plane, respectively.
- Reflections about other planes can be obtained as a combination of rotations and coordinate-plane reflections.

## 4.6 SHEAR

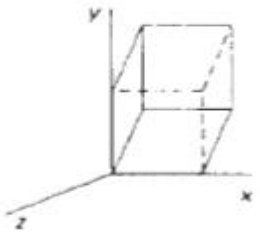
- Shearing transformations can be used to **modify object SHAPES**.



(a)

- They are also useful in three-dimensional viewing for obtaining general projection transformations.

- In two dimensions, we discussed transformations relative to the **x** or y axes to produce distortions in the shapes of objects.



(b)

- In three dimensions, we can also generate shears relative to the *z* axis.

*Figure 11-20*
A unit cube (a) is sheared (b) by transformation matrix 11-47, with $a = b = 1$.

- As an example of three-dimensional shearing. the following transformation
- produces ***a z-axis shear***:

$$SH_z = \begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Parameters a and *b* can be assigned any real values. The effect of this transformation matrix is to alter **x-** and y-coordinate values by an amount that is proportional to the Z value, while leaving the ***z*** coordinate unchanged.
- Boundaries of planes that are perpendicular to the ***z*** axis are thus shifted by an amount proportional to ***z.***
- An **EXAMPLE** of the effect of this shearing matrix on a unit cube is shown in Fig. 11-20, for shearing values a = *b* = 1.
- Shearing matrices for the x axis and y axis are defined similarly.

## 4.7 3D-COMPOSITE TRANSFORMATIONS

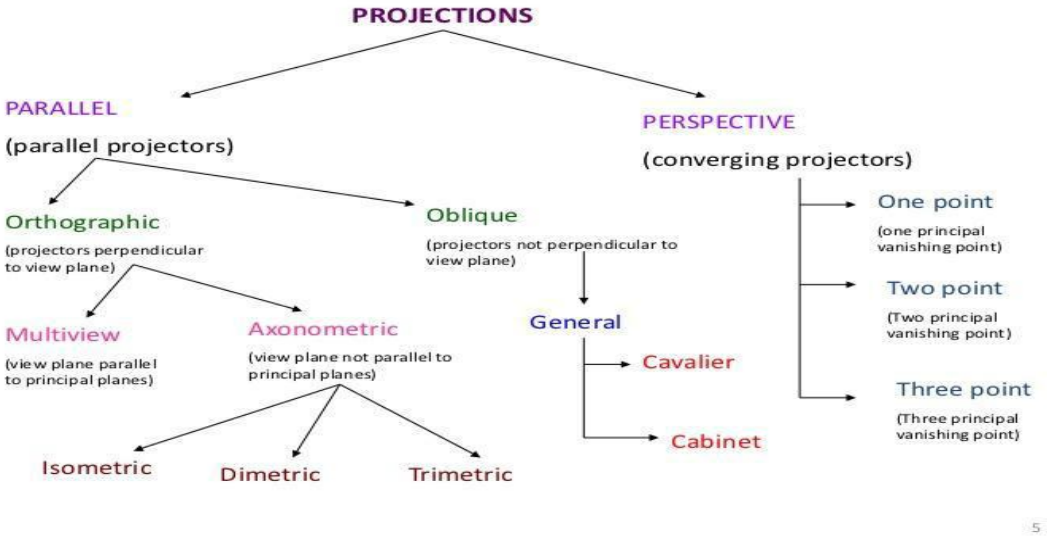
As with two-dimensional transformations. We form a composite three dimensional transformation by multiplying the matrix representations for the individual operations in the transformation sequence. This concatenation is carried out from right to left, where the rightmost matrix is the first transformation to be applied to an object and the leftmost matrix is the last transformation.

- Examples of 3D-Composite transformation are:

  i. Fixed Point scale

  ii. General 3D- Rotations

## 3D-Projections




**Projection :**

Once a 3D model has been completed, its co-ordinates need to be converted to 2 dimensions in order to display the scene on a flat computer monitor or to print it on paper. This process of converting from 3D to 2D is called *projection*

Once world-coordinate descriptions of the objects in a scene are converted to viewing coordinates, we can project the three-dimensional objects onto the two dimensional view plane.

**1. Projection / View Plane :**

The plane where our projection is taken .

**2. View Plane normal vector / Projectors**

The lines emerging from the center of Projection onto the view plane.

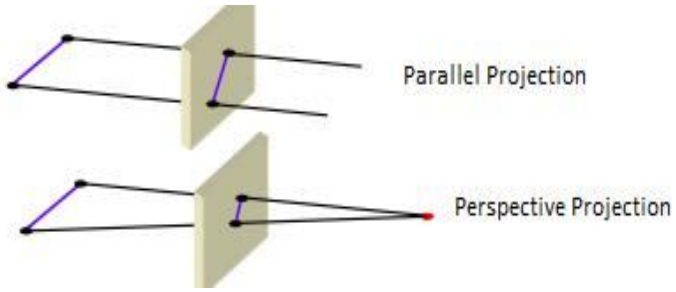
This normal vector is in a direction perpendicular to view plane, that comes from the lines that are extended from our 3d object( view volume).

**3. Center of Projection/ view reference :**

The point from where the projection is to be taken. It is the center of viewing coordinate system. It is chosen to be close to or on surface of object in a scene.

There are two basic projection methods.

1. **Parallel Projection**
2. **Perspective Projection**



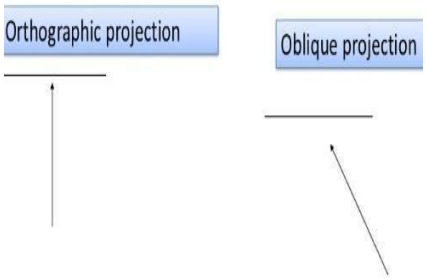

| S.No. | Parallel Projection | Perspective Projection |
|---|---|---|
| 1 | coordinate positions are transformed to the<br>view plane along ***parallel Lines*** | object positions are transformed to the view plane<br>along lines that **converge to a point** called the<br>projection reference point (or center of projection). |
| **2** | Projector is represented using a projection<br>vector | The projected view of an object is determined<br>calculating the ***intersection*** of<br>the projection lines with the view plane. |
| **3** | preserves relative proportions of objects | does not preserve relative proportions |
| **4** | used in ***drafting*** to produce scale drawings of<br><br>three-dimensional objects**.** | used to produce images which look natural. When we<br>view scenes in everyday life far away items appear<br>small relative to nearer items. |
| **5** | Accurate views of the various sides of an<br>object are obtained with a parallel projection, but this ***does not give us a realistic representation*** of the appearance of<br>a three-dimensional object. | Produces realistic views |

Projections of **distant objects are smaller** than the projections of objects of the same size that are closer to the projection plane.

## 4.11 PARALLEL PROJECTONS

- Parallel projection can be specified with a projection **vector** that defines the **DIRECTION** (perpendicular / parallel) **for the projection lines**.

- **We have two types of parallel projection:**



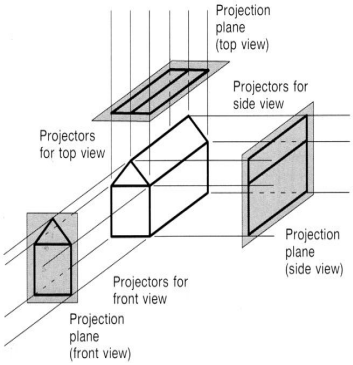

- When the projection is **perpendicula**r to the view plane, we have an ***ORTHOGRAPHIC*** parallel projection.

- Otherwise, we have an **OBLIQUE** parallel projection.

- Figure 12-17 illustrates the two types of parallel projections

### 4.11.1 ORTHOGRAPHIC PROJECTIONS

- used to *produce the front, side, and top views* of an object.
- Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings.
- But, to have a view of complete shape of the object all views should be combined together, as the individual view does not contains the sufficient information of our object completely.

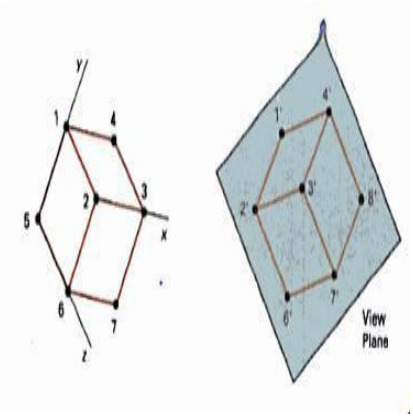



- **Top** orthographic projection is called **PLAN VIEW** and other views are called **Elevations.**
- **Transformation matrix for Orthographic Projection :**

Since the viewing plane is aligned with $(x_v, y_v)$, orthographic projection is performed by:

$$\begin{bmatrix} x_p \\ y_p \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

- **4.11.1.1 TYPES OF ORTHOGRAPHIC PROJECTION :**
  - ***AXONOMETRIC PROJECTION :***
- We can also form orthographic projections that ***display more than one face*** of an object. Such views are called axonometric orthographic projections.
- 3 types of Axonometric Projection :
  - Isometric o Dimetric o Trimetric
- The most commonly used axonometric projection is the **ISOMETRIC PROJECTION**.
- We generate an isometric projection by **aligning the projection plane** so that it **intersects**
  - **each coordinate axis** in which the object is defined (called the ***principal axes***) at the same distance from the origin.
- **EXAMPLE :**Fig shows an isometric projection for a cube.
- The isometric projection is obtained by aligning the projection vector with the cube diagonal.
- There are eight positions, one in each octant, for obtaining an isometric view.
- Foreshortening is the difference between the Projected length and the length of original object length.



- All three principal axes are foreshortened **equally** in an isometric projection so that relative proportions are maintained. This is not the case in a general axonometric projection, where scaling factors may be different for the three principal directions.

- Transformation equations for an orthographic parallel projection are straightforward.
  - **Dimetric Projection :**
- The direction of Projection makes equal angles with exactly any 2 of the principal axes and the foreshortening factor along any 2 axes are equal.

  - **Trimetric Projections :** The direction of Projection makes unequal angles with all the 3 principal axes and the foreshortening factor is unequal along all the 3 axes i,e., FF is not scaled by same factor.

If the view plane is placed at position $z_{vp}$ along the $z_v$ axis (Fig.12-20), then any point (x, y, z) in viewing coordinates is transformed to projection coordinates as

$$x_p = x, \qquad y_p = y$$

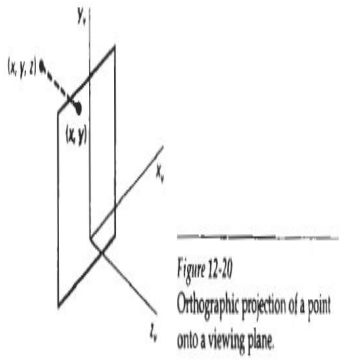


Figure 12-20
Orthographic projection of a point onto a viewing plane.

where the original z-coordinate value is preserved for the depth information needed in depth cueing and visible-surface determination procedures.

## 4.11.2 OBLIQUE PROJECTION

- An oblique projection is obtained by projecting points along parallel lines that are **not perpendicular to the projection plane.**

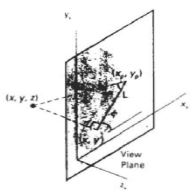


Figure 12-21
Oblique projection of coordinate position (x, y, z) to position $(x_p, y_p)$ on the view plane

- In some applications packages, an oblique projection vector is specified with two angles, as α and φ**,** as shown in Fig. 12-21.

  Point ***(x, y, z)*** is projected to position ***(xp, yp)*** on the view plane. Orthographic projection coordinates on the plane are ***(x, y).*** The oblique projection line from ***(x, y, z)*** to ***(xp, yp)*** makes an angle α with the line on the projection plane that joins ***(xp, yp)*** and ***(x, y).*** T

  This line, of length L, is at an angle φ with the horizontal direction in the projection plane. We can express the projection coordinates in terms of x, y, L, and φ as

  **xp= x+ L cos φ [eq. 12.6]**

  **yp= y+ L sin φ**

  Length L depends on the angle α and the ***z*** coordinate of the point to be projected:

  **Tan α =Z / L**

  Thus, L=Z / Tan α

  = $ZL_1$

  Where L1 is the inverse of tan α, which is also the value of L when z = 1.

  We can then write the oblique projection equations 12-6 as

  **xp= x+ Z($L_1$ cos φ)**

  **yp= y+ Z ($L_1$sin φ)**

$$M_{parallel}=\begin{bmatrix}1 & 0 & L_1\cos\phi & 0\\ 0 & 1 & L_1\sin\phi & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

The transformation matrix for producing any parallel projection onto the $x_v$ $y_v$ plane can be written as

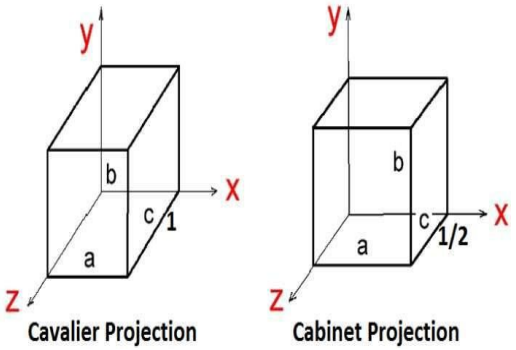
An orthographic projection is obtained when ***L 1*** = 0 (which occurs at a projection angle α of *90").*

Oblique projections are generated with nonzero values for ***L1.***

There are two types of oblique projections Cavalier and Cabinet.

The **Cavalier projection** makes 45° angle with the projection plane. The projection of a line perpendicular to the view plane has the same length as the line itself in Cavalier projection. In a cavalier projection, the foreshortening factors for all three principal directions are equal.

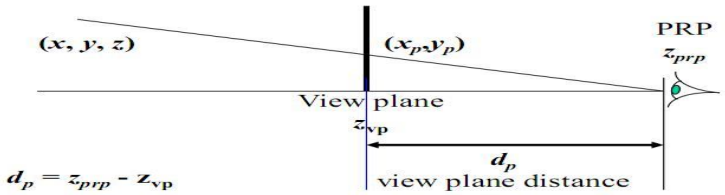
The **Cabinet projection** makes 63.4° angle with the projection plane. In Cabinet projection, lines perpendicular to the viewing surface are projected at ½ their actual length. Both the projections are shown in the following figure −




## ***PERSEPECTIVE PROJECTIONS***

- When an object is viewed from different directions and at different distances, the appearance of the object will be different. Such view is called perspective view.
- **Perspective projections** mimic what the human eyes see. The CoP is at finite distance from the viewing plane.
- To obtain a perspective projection of a three-dimensional object, we transform points along projection lines that meet at the projection reference point.
- Perspective projections are used to produce images which look natural. When we view scenes in everyday life far away items appear small relative to nearer items. This is called perspective foreshortening.

- A side effect of perspective foreshortening is that parallel lines appear to converge on a vanishing point.
- Suppose we set the projection reference point at position $z_{prp}$ along the $z_v$ axis, and we place the view plane $z_{prp}$ at as shown in Fig. 12-25.




We can write equations describing coordinate positions along this perspective projection line in parametric form as :

**x ′ = x – xu**

**y′ = y – yu**

**z′ = z – (z - zprp) u**

- Parameter u takes values from 0 to 1, and coordinate position (x', y', z') represents any point along the projection line.
  - When u = 0, we are at position P = (x,y, z).
  - if u = 1 and we have the projection reference point coordinates (0, 0, $z_{prp}$).
    - On the view plane, z' = z , and we can solve the z' equation for parameter u at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

- Substituting this value of u into the equations for x' and y', we obtain the perspective transformation equations:

$$x_p = x \cdot \left( \frac{z_{prp} - z_{vp}}{z - z_{prp}} \right) = x \cdot \left( \frac{d_p}{z - z_{prp}} \right),$$

$$y_p = y \cdot \left( \frac{z_{prp} - z_{vp}}{z - z_{prp}} \right) = y \cdot \left( \frac{d_p}{z - z_{prp}} \right),$$

$$where \quad d_p = z_{prp} - z_{vp}$$

$\mathbf{d_p}$ is the distance of the view plane from Projection Reference Point (PRP)

$$h = \frac{z - z_{prp}}{\cdot}$$

Using 3D homogenous-coordinate representation, we can write the above equations in matrix form as:

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{z_{vp}}{d_p} & -z_{vp}\left(\frac{z_{prp}}{d_p}\right) \\ 0 & 0 & \frac{1}{d_p} & \left(\frac{-z_{prp}}{d_p}\right) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and the projection coordinates on the view plane are calculated from the homogeneous coordinates as

$$x_p = x_h/h, \qquad y_p = y_h/h \tag{12-16}$$

where the original z-coordinate value would be retained in projection coordinates for visible-surface and other depth processing.

In general, the projection reference point does not have to be along the $z_v$ axis. We can select any coordinate position $(x_{prp}, y_{prp}, z_{prp})$ on either side of the view plane for the projection reference point, and we discuss this generalization in the next section.

There are a number of special cases for the perspective transformation equations 12-13. If the view plane is taken to be the $uv$ plane, then $z_{vp} = 0$ and the projection coordinates are

$$\begin{aligned} x_p &= x\left(\frac{z_{prp}}{z_{prp} - z}\right) = x\left(\frac{1}{1 - z/z_{prp}}\right) \\ y_p &= y\left(\frac{z_{prp}}{z_{prp} - z}\right) = y\left(\frac{1}{1 - z/z_{prp}}\right) \end{aligned} \tag{12-17}$$

And, in some graphics packages, the projection reference point is always taken to be at the viewing-coordinate origin. In this case, $z_{prp} = 0$ and the projection coordinates on the viewing plane are

$$\begin{aligned} x_p &= x\left(\frac{z_{vp}}{z}\right) = x\left(\frac{1}{z/z_{vp}}\right) \\ y_p &= y\left(\frac{z_{vp}}{z}\right) = y\left(\frac{1}{z/z_{vp}}\right) \end{aligned} \tag{12-18}$$

- When a three-dimensional object is projected onto a view plane using perspective transformation equations, any set of parallel lines in the object that are not parallel to the plane are projected into converging lines.
- Parallel Lines that are parallel to the view plane will be projected as parallel lines.
- The point at which a set of projected parallel lines appears to converge is called a **vanishing point.**
- **Each** such set of projected parallel lines will have a separate vanishing point;
- In general, a scene can have any number of vanishing points, depending on how many sets of parallel **lines** there are in the scene.
- The vanishing point for any set of lines that are parallel to one of the principal axes of an object is referred to as a **principal vanishing point**.
- We control the number of principal vanishing points (one, two, or three) with the orientation of the projection plane, and perspective projections are accordingly classified as
  - **One-Point Perspective**

    If the view plane intersects any one of the principal axes ( x, y or z). So we will have one center of Projection and One vanishing point.
  - **Two-Point Perspective**

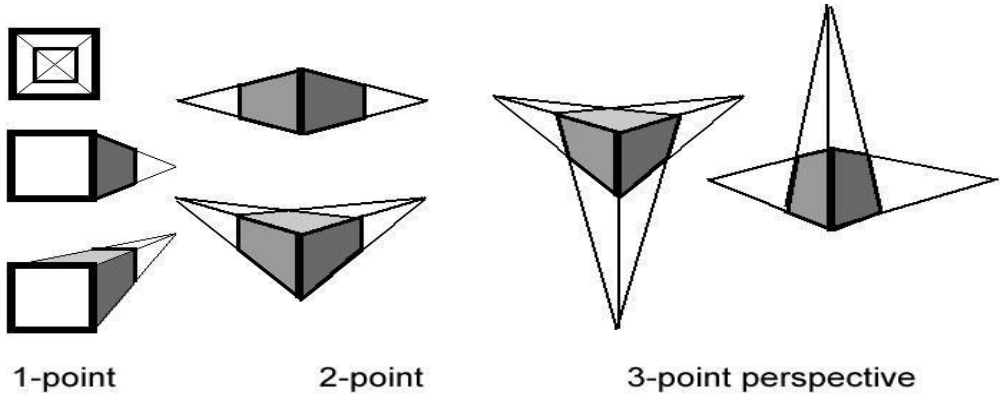
    If the view plane intersects exactly two of the principal axes ( in general x or y).

    So we will have two center of Projections and two vanishing points one on x-axis and other on y-axis.
  - **Three-Point Projections.**

    If the view plane intersects all three of the principal axes ( in general x or y).

    So we will have two center of Projections and two vanishing points one on x-axis and other on y-axis.

- The number of principal vanishing points in a projection is determined by the number of principal axes intersecting the view plane.




Ihe number of principal vanishing points is dependent on how the axes of the coordinate system are positioned against the image plane. If 2 coordinate axes are parallel to the image plane it is called 1-point perspective projection, if only one is parallel to the image plane we call it 2- point perspective projection, and if none of the three axes is parallel to the image plane it is called 3-point perspective projection (because then there are 3 principal vanishing points).

## UNIT-IV
## Assignment-Cum-Tutorial Questions
## SECTION-A

### Objective Questions

**1.** The subcategories of orthographic projection are [ ]

a. cavalier, cabinet, isometric b. cavalier, cabinet

c. isometric, dimetric, trimetric d. isometric, cavalier, trimetric

**2**. Engineering drawing commonly applies for ? [ ]

a. oblique projection b. orthographic projection

c. perspective projection d. None of above

3. The area of computer that is captured by an application is called

a. Window b. View port [ ]

c. Display d. None of these

4. The process of calculating the product of matrices of a number of transformations in sequence is called..... [ ]

a) Concatenation b) Continuation c) Mixing d) None

5. The types of projection are [ ]

a. Parallel projection and perspective projection

b. Perpendicular and perspective projection

c. Parallel projection and Perpendicular projection d. None of these

6. The types of parallel projection are [ ]

a. Orthographic projection and quadric projection

b. Orthographic projection and oblique projection

c. oblique projection and quadric projection d. None of these

7. By which technique, we can take a view of an object from different directions and different distances [ ]

a.Projection b. Rotation c. Translation d. Scaling

8. The process of extracting a portion of a database or a picture inside or outside a specified region are called [ ]

a. Translation b. Shear c. Reflection d. Clipping

9.In Parallel projection, coordinate positions are transformed to the view plane along _____ [ ]

a. vertical lines b. Horizontal lines

c. perpendicular lines d. parallel lines

10. Perspective projections have _____points [ ]

a. composite b. Vanishing c. individual d. separate

**Multiple Choice Questions**

1. Concatenation of how many basic transformation matrices is required to align an arbitrary vector with another vector is 3-D space, if both vectors pass through origin [ ]

a.5 b. 2 c. 1 d. 7

2. Concatenation of how many basic transformation matrices is required to align an arbitrary vector with another vector is 3-D space, if both vectors not pass through origin [ ]

a. 5 b.2 c. 1 d. 7

3. To rotate an object about an arbitrary axis the following operations are required What is their correct sequence [ ]

i) Applying actual rotation

ii)Rotate the arbitrary vector such that it aligns with one of the principal axis

iii)Rotate the vector which is aligned with one of the principal axis to its original position

a. i),ii) and iii) b. ii),i) and iii) c. ii), iii) and i) d. iii), i) and ii)

4. To perform the scaling of a 3-D object, with respect to a selected fixed position, the following operations are required. What is their correct sequence? [ ]

   i) Translate the fixed point back to its original position

   ii) Translate the fixed point to the origin

   iii) Scale the object relative to coordinate origin

a. i), ii) and iii) b. i), iii) and ii) c. ii), iii) and i) d. ii), i) and iii)

5. To perform the mirror reflection of a 3-D object about xy plane, the following operations are required. What is their correct sequence? [ ]

   i) Perform the reflection

   ii) Align the plane normal with z-axis

   iii) Rotate back the plane normal to its original position

a. ii), i) and iii) b. i), ii) and iii) c. iii), i) and ii) d. ii), iii) and i)

6. Find the incorrect statement [ ]

   a. A perspective projection produces realistic views

   b. A perspective projection preserves realistic dimensions

   c. A parallel projection gives realistic representation of 3-D objects

   d. Both B and C

7. In which projection, the plane normal to the projection has equal angles with these three axes [ ]

a.Wire frame model

b. Constructive solid geometry methods

c.isometric projection

d Back face removal

## SECTION-B

## SUBJECTIVE QUESTIONS

1. Describe 3D rotation about x, y ,and z axes and write the corresponding transformation matrices

2. Derive the perspective projection transformation matrix

3. Differentiate between parallel and perspective projections
4. Derive the transformation matrix for rotation about an arbitrary axis which is parallel to any one of the coordinate axes in 3D
5. Derive the transformation matrix for rotation about an arbitrary axis which is not parallel to any one of the coordinate axes in 3D
6. Give the matrix representation for 3D translation, shearing and scaling
7. Give the matrix representation for 3D translation, reflection and scaling
8. Discuss about combined (or) composite 3D transformations
9. Derive the oblique projection transformation matrix
10. explain about types of parallel projections.

**Problem**

1. Calculate a 3D homogenous matrix to rotate by 11 degrees about the line passing through the point (0,0,0) and (1,0,1).

2. Derive the transformation matrix for rotation about an arbitrary axis in 3D, The arbitrary axis is passes through points A[2,1,1,] and B[3,2,2,1]

3. Determine 3D transformation matrices to scale the line PO in the x direction by 3 by keeping point P fixed. Then rotate the line by 45□ anti clockwise about the z axes. Given P(1,1.5,2) and Q(4.5,6,3).

4. Prove that the multiplication of 3D transformation matrices for each of the following sequence of operation is commutative
    i. Any two successive translation
    ii. Any two successive scaling operation
    iii. Any two successive rotation about any one of the coordinate axes

5. Translate a triangle with vertices at original coordinates (10,25,5), (5,10,5), (20,10,10) by tx=15, ty=5,tz=5

6. Scale a triangle with vertices at original coordinates (10,25,5), (5,10,5), (20,10,10) by sx=1.5, sy=2, and sz=0.5 with respect to the origin

7. Determine a 3 D transformation matrices to scale the line PQ in the x direction by 3 by keeping point P fixed. Then rotate the line by 45□ anti clockwise about the z axes. Given P (1, 1.5,2) and Q (4.5, 6, 3).

# Visible Surface Detection Methods

**Objective:** To familarize with various visible surface detection methods.

**Syllabus: Visible Surface Detection Methods**

Classification, back-face detection, depth-buffer, BSP tree methods and area sub-division

**Outcomes:**

Students will be able to:

understand different object space visible surface detection methods.

know the importance of image space visible surface detection methods.

## Learning Material

### Introduction:

A major consideration in the generation of realistic graphics displays is identifying those parts of a scene that are visible from a chosen viewing position.

There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of applications.
Some methods require more memory, some involve more processing time, and some apply only to special types of objects.

Deciding upon a method for a particular application can depend on such factors as

- Complexity of the scene.

- Type of objects to be displayed.
- Available equipment.
- and whether static or animated displays are to be generated.

The various algorithms are referred to as visible-surface detection methods.

Sometimes these methods are also referred to as hidden-surface elimination methods, although there can be subtle differences between identifying visible surfaces and eliminating hidden surfaces.

## CLASSIFICATION OF VISIBLE-SURFACE DETECTION ALGORITHMS

Visible-surface detection algorithms are broadly classified into two types

Object-space methods :

- These methods deal with object definitions directly.
- An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible.

Image-space methods:

- These methods deal with their projected images.
- Visibility is decided point by point at each pixel position on the projection plane.

- Most visible-surface algorithms use image-space methods.

Various visible-surface detection algorithms use sorting and coherence methods to improve performance.

**Sorting:** arrange the surfaces in particular order of their depths

- It is used to facilitate depth comparisons by ordering the individual surfaces in a scene according to their distance from the view plane.

**Coherence:** properties of one part of a scene are related in some way to other parts of the scene so that relationship can be used to reduce sprocessing.

- These methods are used to take advantage of regularities in scene.
- Making use of the results calculated for one part of the scene or image for other nearby parts.
- Coherence is the result of local similarity.
- As objects have continuous spatial extent, object properties vary smoothly within a small local region in the scene. Calculations can then be made incremental.

## BACK-FACE DETECTION

A fast and simple object-space method for identifying the back faces of a polyhe dron is based on the "inside-outside" tests.

A point (x, y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if Ax+By+Cz+D<0.

When an inside point is along the line of sight to the surface, the polygon must be a back face.

We can simplify this test by considering the normal vector N to a polygon surface, which has Cartesian components (A, B, C). In general, if V is a vector in the viewing direction from the eye (or "camera") position then this polygon is a back face if V.N>0.

if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z,. axis, then V = (0, 0, $V_Z$) and V.N=$V_z$.C. So that we only need to consider the sign of C, the Z component of the normal vector N.

In a right-handed viewing system with viewing direction along the negative $z_V$ axis, the polygon is a back face if C < 0.

we can label any polygon as a back face if its normal vector has a z component value c<=0.

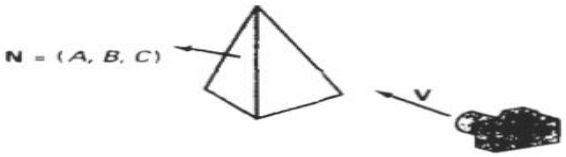



Fig: Vector V in the viewing direction and a back-face normal vector N of a polyhedron.

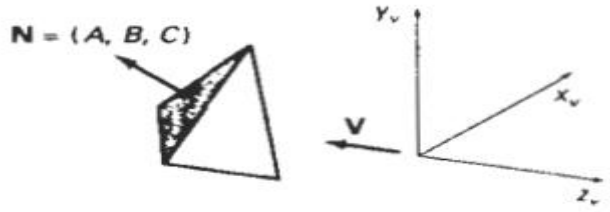



Fig: A polygon surface with plane parameter C < 0 in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative $z_V$ axis.

By examining parameter C for the different planes defining an object, we can immediately identify all the back faces.

For a single convex polyhedron, such as the pyramid, this test identifies all the hidden surfaces on the object, since each surface is either completely visible or completely hidden. Also, if a scene contains only non overlapping convex polyhedra, then again all hidden surfaces are identified with the back-face method.

For other objects, such as the concave polyhedron more tests need to be carried out to determine whether there are additional faces that are totally or partly obscured by other faces.

A general scene can be expected to View of a concave contain overlapping objects along the line of sight. We then need to determine polyhedron with one face where the obscured objects are partially or completely hidden by other objects.

In general, back-face removal can be expected to eliminate about half of the polygon. surfaces in a scene from further visibility tests.

## DEPTH-BUFFER METHOD

A commonly used image-space approach to detecting visible surfaces is the depth-buffer method, which compares surface depths at each pixel position on the projection plane.

This procedure is also referred to as the **z-buffer method**, since object depth is usually measured from the view plane along the z axis of a viewing system.

Each surface of a scene is processed separately, one point at a time across the surface.

The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But the method can be applied to non planar surfaces.

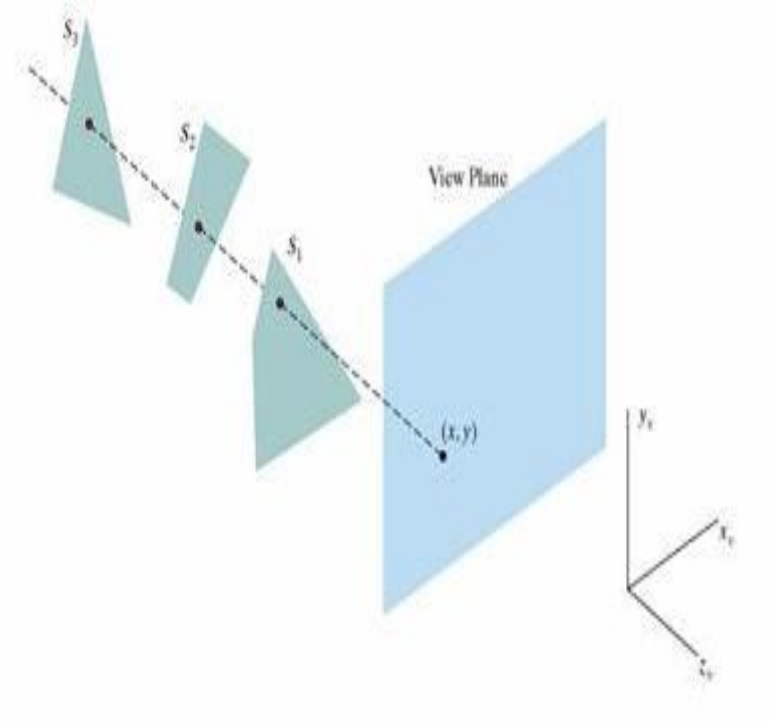
With object descriptions converted to projection coordinates, each (x, y, Z) position on a polygon surface corresponds to the orthographic projection point (x, y) on the view plane.

Therefore, for each pixel position (x, y) on the view plane, object depths can be compared by comparing z values.

Three surfaces at varying distances along the orthographic projection line from position (x,

y) in a view plane taken as the $x_v y_v$ plane. Surface s1, is closest at this position, so its surface intensity value at (x, y) is saved.






We can implement the depth-buffer algorithm in normalized coordinates, so that z values range from 0 at the back clipping plane to $Z_{max}$ at the front clipping .The value of $z_{max}$ can be set either to 1 (for a unit cube) or to the largest value that can be stored on the system.

As implied by the name of this method, two buffer areas are required.

A depth buffer is used to store depth values for each (x, y) position as surfaces are processed.

The refresh buffer stores the intensity values for each position.

Initially, all positions in the depth buffer are set to 0 (minimum depth), and the refresh buffer is initialized to the background intensity.

Each surface listed in the polygon tables is then processed, one scan line at a time, calculating the depth (z value) at each (x, y) pixel position.

The calculated depth is compared to the value previously stored in the depth buffer at that position.

If the calculated depth is greater than the value stored in the depth buffer, the new depth value is stored, and the surface intensity at that position is determined and in the same xy location in the refresh buffer.

We summarize the steps of a depth-buffer algorithm as follows:

1. Initialize the depth buffer and refresh buffer so that for all buffer positions $(x, y)$,

$$\text{depth}(x, y) = 0, \qquad \text{refresh}(x, y) = I_{\text{backgnd}}$$

2. For each position on each polygon surface, compare depth values to previously stored values in the depth buffer to determine visibility.

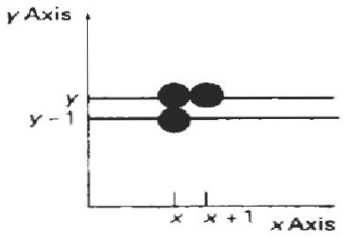
   - Calculate the depth z for each $(x, y)$ position on the polygon.
   - If $z > \text{depth}(x, y)$, then set

$$\text{depth}(x, y) = z, \qquad \text{refresh}(x, y) = I_{\text{surf}}(x,y)$$

where $I_{\text{backgnd}}$ is the value for the background intensity, and $I_{\text{surf}}(x,y)$ is the projected intensity value for the surface at pixel position $(x,y)$. After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the refresh buffer contains the corresponding intensity values for those surfaces.

Depth values for a surface position (x, y) are calculated from the plane equation for each surface:

$$z = \frac{-Ax - By - D}{C}$$




For any scan line adjacent horizontal positions across the line differ by 1, and a vertical y value on an adjacent scan line differs by 1.

If the depth of position (x, y) has been determined to be z, then the depth z' of the next position (x + 1, y) along the scan line is obtained from Equation.

$$z' = \frac{-A(x+1) - By - D}{C}$$

$$z' = z - \frac{A}{C}$$

The ratio -A/C is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.

On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line.

We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line, Starting at a top vertex. we can recursively calculate

x positions down a left edge of the polygon as x' = ***x*** - **l/m,**

where m is the slope of the edge. Depth values down the edge are then obtained recursively as

If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C}$$

An alternate approach is to use a midpoint method or Bresenham-type algorithm for determining x values on left edges for each scan line.

$$z' = z + \frac{A/m + B}{C}$$

The method can be applied to curved surfaces by determining depth and intensity values at each surface projection point.

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene.

But it does require the availability of a second buffer in addition to the refresh buffer.

A system with a resolution of 1024 by 1024, for example, would require over a million positions in the depth buffer, with each position containing enough bits to represent the number of depth increments needed.

One way to reduce storage requirements is to process one section of the scene at a time, using a smaller depth buffer. After each view section is processed, the buffer is reused for the next section.

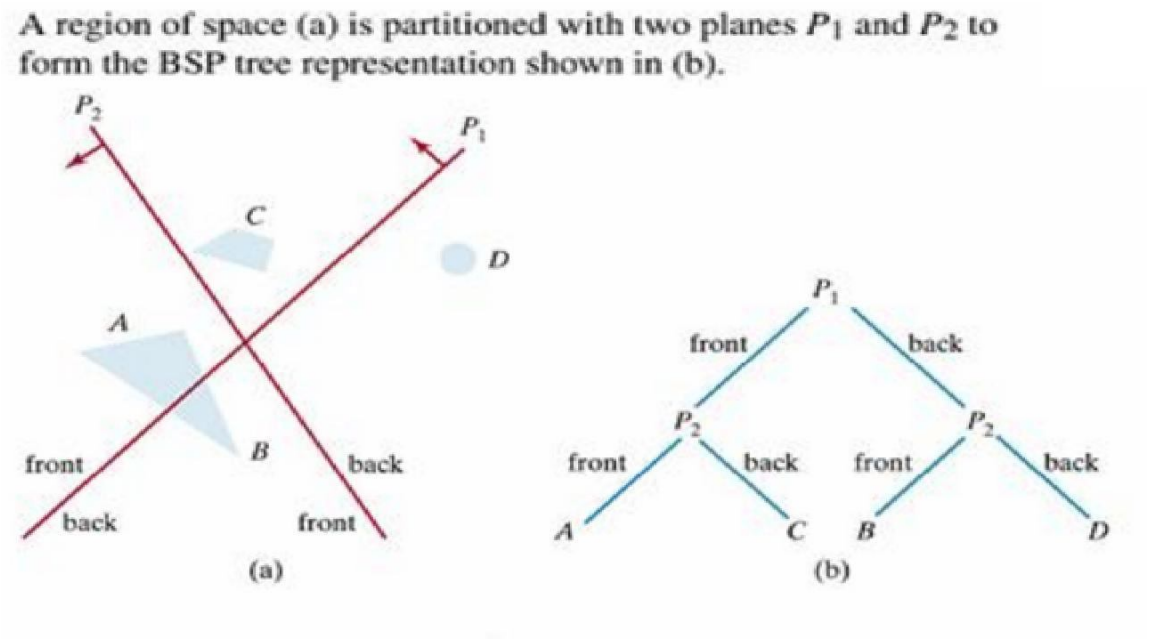
$$z' = z + \frac{B}{C}$$

## BSP-TREE METHOD

A binary space-partitioning (BSP) tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front, as in the painter's algorithm.

The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" and "outside" the partitioning plane at each step of the space subdivision, relative to the viewing direction.




## Process:

With plane P1,we first partition the space into two sets of objects.

One set of objects is behind, or in back of, plane P1 relative to the viewing direction, and the other set is in front of P1.

Since one object is intersected by plane P1, we divide that object into two separate objects, labeled A and B.

Objects A and C are in front of P1 and objects B and Dare behind P1.

We next partition the space again with plane P2 and construct the binary tree representation. In this tree, the objects are represented as terminal nodes, with front objects as left branches

and back objects as right branches.

For objects described with polygon facets, we chose the partitioning planes to coincide with the polygon planes. T

The polygon equations are then used to identify "inside" and "outside" polygons, and the tree is constructed with one partitioning plane for each polygon face.

Any polygon intersected by a partitioning plane is split into two parts.

When the BSP tree is complete, we process the tree by selecting the surfaces for display in the order back to front, so that foreground objects are painted over the background objects.

Fast hardware implementations for constructing and processing BSP trees are used in some

systems.

## AREA-SUBDIVISION METHOD

This technique for hidden-surface removal is essentially an image-space method, but object-space operations can be used to accomplish depth ordering of surfaces.

The area-subdivision method takes advantage of area coherence in a scene by locating those view areas that represent part of a single surface.

We apply this method by successively dividing the total viewing area into smaller and smaller rectangles until each small area is the projection of part of n single visible surface or no surface at all.

To implement this method, we need to establish tests that can quickly identify the area as part of a single surface or tell us that the area is too complex to analyze easily.

Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles.

If the tests indicate that the view is sufficiently complex, we subdivide it.

Next. we apply the tests to each of the smaller areas, subdividing these if the tests indicate that visibility of a single surface is still uncertain.

We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until they are reduced to the size of a single pixel.

An easy way to do this is to successively divide the area into four equal parts at each step.

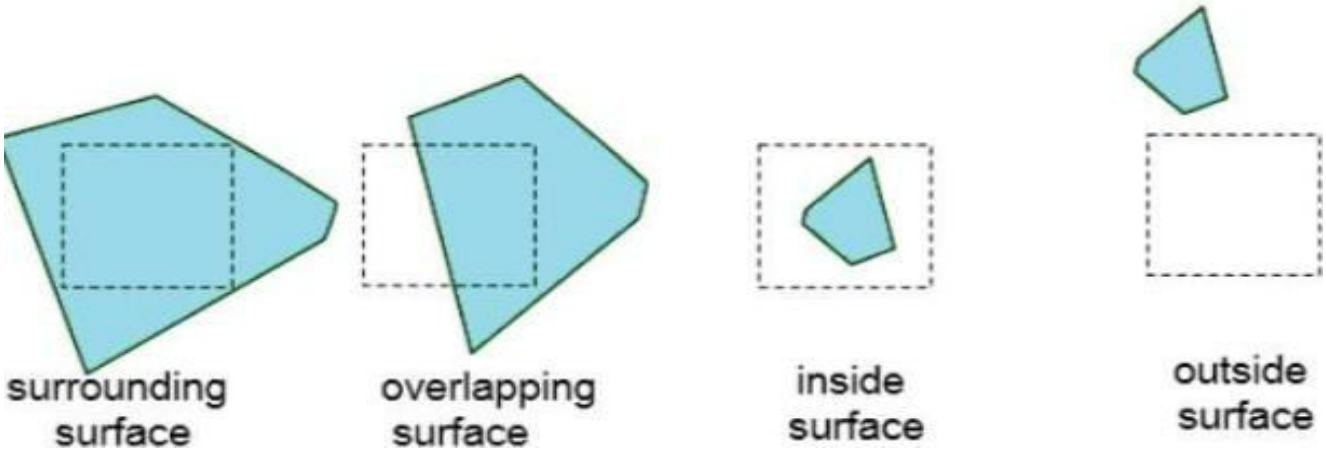
This approach is similar to that used in constructing a quad tree. A viewing area with a resolution of 1024 by 1024 could be subdivided ten times in this way before a subarea is reduced to a point.

Tests to determine the visibility of a single surface within a specified area made by comparing surfaces to the boundary of the area.
There are four possible relationships that a surface can have with a specified area boundary. We can describe these relative surface characteristics in the following way

**Surrounding surface**-One that completely encloses the area.

**Overlapping surface**-One that is partly inside and partly outside the area. **Inside surface**-One that is completely inside the area.

**Outside surface**-One that is completely outside the area.



The tests for determining surface visibility within an area can be stated in terms of these four classifications. No further subdivisions of a specified area are needed if one of the following conditions is true:

**1.All surfaces are outside surfaces with respect to the area.**

**2.Only one inside, overlapping, or surrounding surface is in the area.**

**3.A surrounding surface obscures all other surfaces within the area boundaries.**

**Test 1** can be carried out by checking the bounding rectangles of all surface against the area boundaries.

**Test 2** can also use the bounding rectangles in the **xy** plane to identify an inside surface. For other **types** of surfaces, the bounding rectangles can be used as an initial check.

If a single bounding rectangle intersects the area in some way, additional checks are used to determine whether the surface is surrounding, overlapping, or outside.

Once a single inside, overlapping, or surrounding surface has been identified, its pixel intensities are transferred to the appropriate area within the frame buffer.

One method for implementing **test 3** is to order surfaces according to their minimum depth from the view plane.

- For each surrounding surface, we then compute the maximum depth within the area under consideration.

o If the maximum depth of one of these surrounding surfaces is closer to the view plane than the minimum depth of all other surfaces within the area, **test 3** is satisfied

Another method for carrying out test 3 that does not require depth sorting is to use plane equations to calculate depth values at the four vertices of the area for all surrounding, overlapping, and inside surfaces.

If the calculated depths for one of the surrounding surfaces are less than the calculated depths for all other surfaces, test 3 is true.

Then the area can be filled with the intensity values of the surrounding surface.

For some situations, both methods of implementing test 3 will fail to identify correctly a surrounding surface that obscures all the other surfaces.

Further testing could be carried out to identify the single surface that covers the area, but it is faster to subdivide the area than to continue with more complex testing.

Once outside and surrounding surfaces have been identified for an area, they will remain outside and surrounding surfaces for all subdivisions of the area.

Furthermore, some inside and overlapping surfaces can be expected to be eliminated as the subdivision process continues, so that the areas become easier to analyze.

In the limiting case, when a subdivision the size of a pixel is produced, we simply calculate the depth of each relevant surface at that point and transfer the intensity of the nearest surface to the frame buffer.

## UNIT-V

## Assignment-Cum-Tutorial Questions

## SECTION-A

### Objective Questions

1. Depth buffer method is also called as_____ [ ]
   a) Back-face Detection b)Z-buffer C) Scan-line Method d) Octree Method
2. The method which is based on the principle of comparing objects and parts of each other to find which are visible and which are hidden are called. [ ]
   a) Object-Space method b) image-space method C)Both a&b d) None
3. The method which is based on the principle of checking the visibility point at each pixel position on the projection plane are called. [ ]
   a) Object-Space method b) image-space method C)Both a & b d) None
4. Scan lines are used to scan from [ ]
   a) Top to Bottom b) Bottom to Top c) Both a & b d) None
5. The Z-Buffer algorithm is also referred as [ ]
   a)Depth Sorting Algorithm b) Depth Buffer Algorithm c) Both a & b d) none
6. Which one of the following are image space methods? [ ]
   a) Scan line Method b) Depth Buffer Method c) Both a & b d) none
7. Which surface algorithm is based on perspective depth [ ]
   a) Depth comparison c) Z-Buffer or Depth-Buffer Algorithm
   b) Sub Division Method d) Back-Face Removal.
8. Depth Buffer method is also called as_____
9. BSP Method refers to [ ]
   a) Binary Space Partitioning c) Business Systems Planning
   b) only c d) None
10. if z>depth(x,y) in depth buffer algorithm the
    1. depth(x,y)=_____ 2. Refresh(x,y)=_____
11. If N is a normal vector to a polygon surface and V is is a vector in the viewing direction from the eye then this polygon is back-face if [ ]

a) V.N<0 b) V.N=0 c) V.N>0 d) V.N<=0

12. Initial values of depth buffer and refresh buffer are [ ]

a) depth(x,y)=1,refresh(x,y)=0 c) depth(x,y)=0,refresh(x,y)=0

b) depth(x,y)=1,refresh(x,y)=1 d) depth(x,y)=1,refresh(x,y)=0

13. Which of the following method is not applicable for cyclically overlapped surface. [ ]

a) Depth-Sorting b) Scan line Method c) Object-Space method d) none

14. Which of the following is an efficient method for determining object visibility by painting surface onto the screen from back to front [ ]

a) Back-Face Detection c) Area-Sub Division Method

b) BSP-Tree Method d) Z-Buffer

15. Depth values for a surface position(x,y) are calculated by using the following plane equation [ ]

a) Z=-AX-BY c) Z=AX+BY

b) Z=-AX-D d) Z=-AX-BY-D/C

16.The following method is using a combination of both image space and object space operations [ ]

a) Back-Face Detection c) Octree

b) Scan-Line Method d) Area-Sub Division Method

17. No further sub division of a specified area are needed if one of the following conditions is true [ ]

a) All Surface are outside surface with respect to area.

b) Only one inside, overlapping, or surrounding surface is in the area.

c) A Surrounding surface obscures all other surfaces within the are boundaries.

d) Any one of the above condition.

18. Sorting is used to facilitate. [ ]

a) Depth comparisons by ordering the individual surfaces in a scene according to their distance from the view plane.

b) Depth comparisons by disordering the individual surfaces in a scene according to their distance from the view plane.

c) Depth comparisons by order the all surfaces in a scene according to their distance from the view plane.

d) None.

19. Coherence methods are used. [ ]

a) To Take advantage of regularities in a scene.

b) To Take advantage of irregularities in a scene.

c) Both of the Above d) None

SECTION-B

Descriptive Questions

1. Distinguish between object-space and image space methods of visible surface detection algorithms. Give example for each?
2. Distinguish depth-sort and z-buffer algorithms?
3. Demonstrate Back-Face Detection methods?
4. Give a brief explanation about Depth Buffer method?
5. Discuss Binary space partitioning method?
6. Explain in detail about area sub division method?
7. List and explain different cases in area sub division algorithm?
8. Give the point P1(3,6,20), P2(2,4,6) and P3(2,4,6) a view point C(0.0,-10) determine which points obscure the others when viewed from C.
9. Give the point P1(3,6,20), P2(2,4,8) and P3(2,4,8) a view point C(0.0,-20) determine which points obscure the others when viewed from C.
10. Assuming Z-Buffer algorithm allows 256 depth value level to be used, approximately how much memory would a 512X512 pixel require to store the Z-Buffer?
11. Assuming that allows $2^{24}$ depth value level to be used, how much memory would a 1024X768 pixel require to store the Z-Buffer?
12. Assuming that allows $2^{24}$ depth value level to be used, how much memory would a 1024X1024pixel require to store the Z-Buffer?

13. If the camera viewing directions is V and the surface normal plane is N, how to determine whether the surface is visible with respect to viewing direction or not?

# Unit – VI

## Introduction:

- Computer animation generally refers to any time sequence of visual changes in a scene.
- In addition to changing object position with translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture.
- Applications of computer-generated animation are
    - Entertainment (motion pictures and cartoons)
    - Advertising
    - Scientific and engineering studies
    - Training and education.
- Animations often transition one object shape into another: transforming a can of motor oil into an automobile engine
- Computer animations can also be generated by changing camera parameters, such as position, orientation, focal length and lighting effects.

## DESIGN OF ANIMATION SEQUENCES

- In general, an animation sequence is designed with the following steps:
    - Storyboard layout
    - Object definitions
    - Key-frame specifications.
    - Generation of in-between frames

Storyboard layout

- It is an outline of the action.
- It defines the motion sequence as a set of basic events that

are to take place.

- Depending on the type of animation to be produced, the storyboard could consist of a set of rough sketches or it could be a list of the basic ideas for the motion.

Object definition

- It is given for each participant in the action.
- Objects can be defined in terms of basic shapes, such as polygons or splines.
- In addition, the associated movements for each object are specified along with the shape.

Key-frame specifications

- It is a detailed drawing of the scene at a certain time in the animation sequence.
- Within each key frame, each object is positioned according to the time for that frame.
- The time interval between key frames is not *too* great.

In-between frames

- These are the intermediate frames between the key frames.
- The number of in-betweens needed is determined by the media to be used to display the animation.
- Film requires 24 frames per second, and graphics terminals are refreshed at the rate of ***30*** to **60** frames per second.
- Depending on the speed specified for the motion, some key frames can b**e** duplicated.
- For a 1-minute film sequence with no duplication, we would need 1440 frames. With five in-betweens for each pair of key frames, we would need 288 key frames.
- There are several other tasks that may be required,

depending on the application.

- They include motion verification, editing, and production and synchronization of a soundtrack.
- Many of the functions needed to produce general animations are now computer- generated.

## GENERAL COMPUTER-ANIMATION FUNCTIONS

- Some steps in the development of an animation sequence are well-suited to computer solution.
- These include object manipulations and rendering, camera motions, and the generation of in-betweens.
- Animation packages, such as Wave front, for example, provide special functions for designing the animation and processing individual objects.
- One function available in animation packages is provided to store and manage the object database.
- Object shapes and associated parameters are stored and updated in the database.
- Other object functions include those for motion generation and those for object rendering.
- Motions can be generated according to specified constraints using two-dimensional or three-dimensional transformations.
- Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.
- Another typical function simulates camera movements. Standard motions are zooming, panning, and tilting.
- Finally, given the specification for the key frames, the in-between can be automatically generated.
- Animation functions include a graphics editor, a key-frame

generator, an in-between generator, and standard graphics routines.

- The graphics editor allows us to design and modify object shapes, using spline surfaces, constructive solid-geometry methods, or other representation schemes.

## RASTER ANIMATIONS

- On raster systems, we can generate real-time animation in limited applications using raster operations.
- A simple method for translation in the xy plane is to transfer a rectangular block of pixel values from one location to another.
- Two dimensional rotations in multiples of 90" are also simple to perform, although we can rotate rectangular blocks of pixels through arbitrary angles using ant aliasing procedures.
- To rotate a block of pixels, we need to determine the percent of area coverage for those pixels that overlap the rotated block.
- Sequences of raster operations can be executed to produce real-time animation of either two- dimensional or three-dimensional objects.
- We can also animate objects along two-dimensional motion paths using the **color-table transformations.**
- Here we predefine the object at successive positions along the motion path.
- Set the successive blocks of pixel values to color-table entries.
- We set the pixels at the first position of the object to "on" values, and we set the pixels at the other object positions to the background color.
- The animation is then accomplished by changing the color-table values so that the object is "on" at successively positions along the animation path as the preceding position is set to the background intensity.

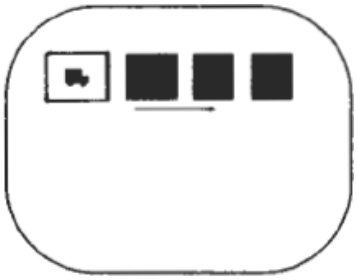


Fig: Real-time raster color-table animation

## KEY-FRAME SYSTEMS

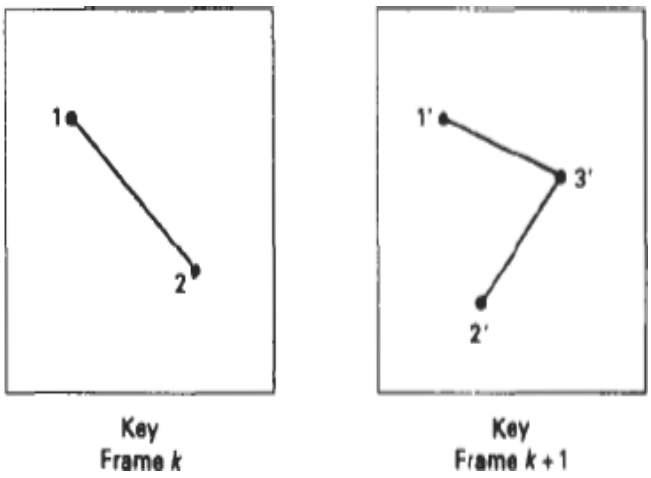
- We generate each set of in-betweens from the specification of two (or more) key frames.
- Motion paths can be given with a kinematic description as a set of spline curves, or the motions can be physically based by specifying the forces acting on the objects to be animated.
- For complex scenes, we can separate the frames into individual components or objects called cels (celluloid transparencies), an acronym for cartoon animation.
- With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features.
- If all surfaces are described with polygon meshes, then the number of edges per polygon can change from one frame to the next. Thus, the total number of line segments can be different in different frames.

### Morphing

- Transformation of object shapes from one form to another is called morphing.
- Morphing methods can be applied to any motion or transition involving a change in shape.
- Given two key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (or the number of vertices) is the same for the two

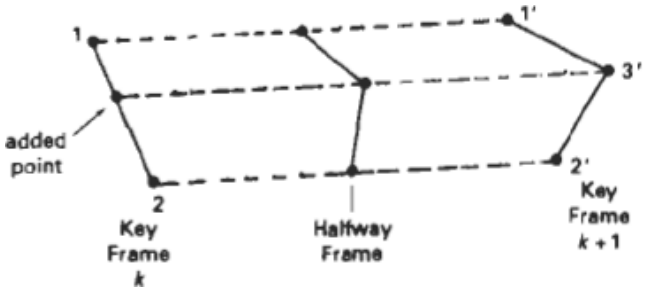
frames.

- A straight-line segment in key frame k is transformed into two line segments in key frame k+1. Since key frame k + 1 has an extra vertex, we add a vertex between vertices 1 and 2 in key frame k to balance the number of vertices (and edges) in the two key frames.
- Using linear interpolation to generate the in-betweens. We transition the added vertex in key frame k into vertex 3' along the straight-line path.

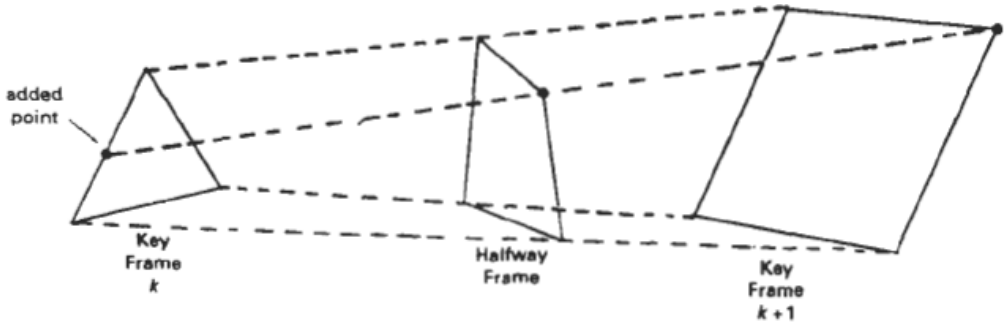



An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame k

+ 1




Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame k + 1.




Linear interpolation for transforming a triangle into a quadrilateral

- We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added b a key frame.
- Suppose we equalize the edge count, and parameters $L_k$ and $L_{K+1}$ denote the number of line segments in two consecutive frames. We then define

$L_{max} = \max(L_k, L_{k+1})$ $L_{min} = \min(L_k, L_{k+1})$ $N_e = L_{max} \bmod L_{min}$ $N_s = \text{int}(L_{max} / L_{min})$

- Then the preprocessing is accomplished by

1. Dividing $N_e$ edges of $\text{keyframe}_{min}$ into $N_s + 1$ sections.
2. Dividing the remaining lines of $\text{keyframe}_{min}$ into $N_s$ sections.

- As an example, if $L_k = 15$ and $L_{K+1} = 11$, we would divide 4 lines of $\text{keyframe}_{k+1}$ into 2 sections each. The remaining lines of $\text{keyframe}_{k+1}$ are left intact.
- If we equalize the vertex count, we can use parameters $V_k$ and $V_{k+1}$ to denote the number of vertices in the two consecutive frames. In

this case, we define

$V_{max}$= max($V_k$,$V_{k+1}$)  $V_{min}$= min($V_k$,$V_{k+1}$)

$$N_{ls} = (V_{max} - 1) \bmod (V_{min} - 1)$$
$$N_p = \text{int}\left(\frac{V_{max} - 1}{V_{min} - 1}\right)$$

Preprocessing using vertex count is performed by

1. Adding $N_P$ points to $N_{ls}$ line sections of $keyframe_{min}$.
2. Adding $N_p$ - 1 points to the remaining edges of $keyframe_{min}$

- For the triangle-to quadrilateral example, $V_k$ = 3 and $V_{k+1}$ = 4. Both $N_p$ and $N_{ls}$ are 1, so we would add one point to one edge of $keyframe_k$ No points would be added to the remaining lines of $keyframe_{k+1}$.

Simulating Accelerations

- Curve-fitting techniques are often used to specify the animation paths between key frames.
- Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths.
- To simulate accelerations, we can adjust the time spacing for the in-betweens.
- For constant speed (zero acceleration) we use equal-interval time spacing for the in-betweens.
    - Suppose we want n in-betweens for key frames at times t1 and t2. The time interval between key frames is then divided

$$\Delta t = \frac{t_2 - t_1}{n + 1}$$

      into n +1 subintervals, yielding an in-between spacing of
    - We can calculate the time for any in-between as

$$tB_j = t_1 + j\,\Delta t, \qquad j = 1, 2, \ldots, n$$

and determine the values for coordinate positions, color, and other physical parameters.

- Nonzero accelerations:
  - These are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence.
  - We can model the start-up and slowdown portions of an animation path with spline or trigonometric functions.
  - Parabolic and cubic time functions haw been applied to acceleration modeling, but trigonometric functions are more commonly used in animation packages.
  - To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster.
  - We can obtain an increasing interval size with the function

$$1 - \cos\theta, \qquad 0 < \theta < \pi/2$$

  - For n in-betweens, the time for the jth in-between would then be calculated as

$$tB_j = t_1 + \Delta t\left[1 - \cos\frac{j\pi}{2(n+1)}\right], \qquad j = 1, 2, \ldots, n$$

where $\Delta t$ is the time difference between the two key frames.

We can model decreasing speed (deceleration) with $\sin\theta$ in the range $0 < \theta < \pi/2$. The time position of an in-between is now defined as

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \qquad j = 1, 2, \ldots, n$$

- Motions contain both speed-ups and slow-downs. We can model a combination of increasing-decreasing speed by first increasing the in-between time spacing, then we decrease this sparing. A function to accomplish these time changes is

$$\frac{1}{2}(1 - \cos\theta), \qquad 0 < \theta < \pi/2$$

The time for the $j$th in-between is now calculated as

$$tB_j = t_1 + \Delta t\left\{\frac{1 - \cos[j\pi/(n+1)]}{2}\right\}, \qquad j = 1, 2, \ldots, n$$

where $\Delta t$ is the time difference between the two key frames.

**Graphics programming using OPENGL:**

OpenGL is a software interface that allows you to access the graphics hardware without taking care of the hardware details or which graphics adapter is in the system. OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

**Libraries**

OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections and rendering surfaces.

OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

**Include Files**

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with: #include <GL/gl.h>

#include<GL/glu.h>

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

#include<GL/glu.h>

The following files must be placed in the proper folder to run a OpenGL Program.

Libraries (place in the lib\ subdirectory of Visual C++)

- opengl32.
- lib glu32.
- lib glut32.lib

Include files (place in the include\GL\ subdirectory of Visual C++)

- gl.h
- glu.h
- glut.h

Dynamically-linked libraries (place in the \Windows\System subdirectory)

- opengl32.dll
- glu32.dll
- glut32.dll

**Working with OpenGL:**

**Opening a window for Drawing:**

1. The First task in making pictures is to open a screen window for drawing. The following five functions initialize and display the screen window in our program.

glutInit (&argc, argv)

The first thing we need to do is call the glutInit () procedure. It should be called before any other GLUT routine because it initializes the GLUT library. The parameters to glutInit () should be the same as those to main (), specifically main (int argc, char** argv) and glutInit (&argc, argv).

2. glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB) The next thing we need to do is call the glutInitDisplayMode () procedure to specify the display mode for a window. We must first decide whether we want to use an RGBA (GLUT_RGB) or color-index (GLUT_INDEX) color model. The RGBA mode stores its color buffers as red, green, blue, and alpha color components. Color-index mode, in contrast, stores color buffers in indices. And for special effects, such as shading, lighting, and fog, RGBA mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is the default.

3. glutInitWindowSize(640,480) :

We need to create the characteristics of our window. A call to glutInitWindowSize() will be used to specify the size, in pixels, of our initial window. The arguments indicate the height and width (in pixels) of the requested window.

4. glutInitWindowPosition (100,15)

Similarly, glutInitWindowPosition() is used to specify the screen location for the upper left corner of our initial window. The arguments, x and y, indicate the location of the window relative to the entire display. This function

positioned the screen 100 pixels over from the left edge and 150 pixels down from the top.

5. glutCreateWindow ("Example")

To create a window, the with the previously set characteristics (display mode, size, location, etc), the programme uses the glutCreateWindow () command. The command takes a string as a parameter which may appear in the title bar.

6. glutMainLoop ()

The window is not actually displayed until the glutMainLoop () is entered. The very last thing is we have to call this function

**Event Driven Programming:**

The method of associating a call back function with a particular type of event is called as event driven programming. OpenGL provides tools to assist with the event management. There are four Glut functions available.

1. glutDisplayFunc (mydisplay)

The glutDisplayFunc() procedure is the first and most important event call-back function. A call-back function is one where a programmer-specified routine can be registered to be called in response to a specific type of event. For example, the argument of glutDisplayFunc(mydisplay) is the function that is called whenever GLUT determines that the contents of the window needs to be redisplayed. Therefore, we should put all the routines that you need to draw a scene in this display call-back function

2. glutReshapeFunc(myreshape)

The glutReshapeFunc () is a call-back function that specifies the function that is called whenever the window is resized or moved. Typically, the function that is called when needed by the reshape function displays the window to the new size and redefines the viewing characteristics as desired.

3. glutKeyboardFunc (my keyboard) GLUT interaction using keyboard inputs is handled. The command glutKeyboardFunc() is used to run the call-back function specified and pass as parameters, the ASCII code of the pressed key, and the x and y coordinates of the mouse cursor at the time of the event. Special keys can also be used as triggers. The key passed to the call-back function, in this case, takes one of the following values (defined in glut.h). Special keys can also be used as triggers. The key passed to the call-back function, in this case, takes one of the following values (defined in glut.h).

| | |
|---|---|
| • GLUT_KEY_UP | Up Arrow |
| • GLUT_KEY_RIGHT | Right Arrow |
| • GLUT_KEY_DOWN | Down Arrow |
| • GLUT_KEY_PAGE_UP | Page Up |
| • GLUT_KEY_PAGE_DOWN | Page Down |
| • GLUT_KEY_HOME | Home |
| • GLUT_KEY_END | End |
| • GLUT_KEY_INSERT | Insert |

4. glutMouseFunc (mymouse) GLUT supports interaction with the computer mouse that is triggered when one of the three typical buttons is presses. A mouse call-back fuction can be initiated when a given mouse button is pressed or released. The command glutMouseFunc () is used to specify the call-back function to use when a specified button is is a given state at a certain location. This buttons are defined as GL_LEFT_BUTTON, GL_RIGHT_BUTTON, or GL_MIDDLE_BUTTON and the states for that button are either GLUT_DOWN (when pressed) or GLUT_UP (when released).

**Example : Skeleton for OpenGL Code**

```
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(465, 250);
glutInitWindowPosition(100, 150);
glutCreateWindow("My First Example");
glutDisplayFunc(mydisplay);
glutReshapeFunc(myreshape);
glutMouseFunc(mymouse);
glutKeyboardFunc(mykeyboard);
myinit();
glutMainLoop();
return 0;
}
```

**Basic graphics primitives**

OpenGL Provides tools for drawing all the output primitives such as points, lines, triangles, polygons, quads etc and it is defined by one or more vertices. To draw such objects in OpenGL we pass it a list of vertices. The list occurs between the two OpenGL function calls glBegin() and glEnd(). The argument of glBegin() determine which object is drawn.

These functions are

glBegin(int mode);

glEnd( void );

The parameter mode of the function glBegin can be one of the following:

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_TRIANGLES GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN GL_QUADS

GL_QUAD_STRIP

GL_POLYGON

glVertex( ) : The main function used to draw objects is named as glVertex. This function defines a point (or a vertex) and it can vary from receiving 2 up to 4 coordinates.

**Example**

//the following code plots three dots

glBegin(GL_POINTS);

```
glVertex2i(100, 50);
 glVertex2i(100, 130);
 glVertex2i(150, 130); glEnd( );
 / / the following code draws a triangle
glBegin(GL_TRIANGLES);
glVertex3f(100.0f, 100.0f, 0.0f);
glVertex3f(150.0f, 100.0f, 0.0f);
glVertex3f(125.0f, 50.0f, 0.0f);
glEnd( );
/ / the following code draw a lines
glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f);/ / origin of the line
glVertex3f(200.0f, 140.0f, 5.0f);/ / ending point of the line
glEnd( );
```

**OpenGl State** :

OpenGl keeps track of many state variables, such as current size of a point, the current color of a drawing, the current background color, etc.

The value of a state variable remains active until new value is given.

**glPointSize()** : The size of a point can be set with glPointSize(), which takes one floating point argument.

Example: glPointSize(4.0);

**glClearColor ()** : establishes what color the window will be cleared to. The background color is set with glClearColor (red, green, blue, alpha), where alpha specifies a degree of transparency.

Example : glClear(GL_COLOR_BUFFER_BIT)

glColor3f() : establishes to use for drawing objects. All objects drawn after this point use this color, until it□s changed with another call to set the color.

**Example**:

glColor3f(0.0, 0.0, 0.0); / / black

glColor3f(1.0, 0.0, 0.0); / / red

glColor3f(0.0, 1.0, 0.0); / / green

glColor3f(1.0, 1.0, 0.0); / / yellow

glColor3f(0.0, 0.0, 1.0); / / blue

glColor3f(1.0, 0.0, 1.0); / / magenta

glColor3f(0.0, 1.0, 1.0); / / cyan

glColor3f(1.0, 1.0, 1.0); / / white

**Example** : **White Rectangle on a Black Background (3-Dimension co-ordinates)**

```
#include "stdafx.h"

#include "gl/glut.h"

 #include<gl/gl.h>

 Void Display(void) {

 glClearColor (0.0, 0.0, 0.0, 0.0);

glClear (GL_COLOR_BUFFER_BIT);
```

```
glColor3f (1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
 glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();
glFlush();
}
int main (int argc, char **argv)
{
 glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(640,480);
glutCreateWindow("Intro");
glClearColor(0.0,0.0,0.0,0.0);
glutDisplayFunc(Display);
glutMainLoop();
return 0;
 }
```

**Making Line Drawings**

OpenGL makes it easy to draw a line: use GL_LINES as the argument to glBegin(), and pass it the two end points as vertices. Thus to draw a line between (40,100) and (202,96) use:

glBegin(GL_LINES); / / use constant GL_LINES here

glVertex2i(40, 100);

glVertex2i(202, 96);

glEnd();

OpenGL provides tools for setting the attributes of lines. A line□s color is set in the same way as for points, using glColor3f(). To draw thicker lines use glLineWidth(4.0). The default thickness is 1.0 To make stippled (dotted or dashed) lines, you use the command glLineStipple() to define the stipple pattern, and then we enable line stippling with glEnable().

glLineStipple(1, 0x3F07);

glEnable(GL_LINE_STIPPLE);

**Drawing Polylines and Polygons:**

Polyline is a collection of line segments joined end to end. It is described by an ordered list of points,

In OpenGL a polyline is called a “line strip”, and is drawn by specifying the vertices in turn between glBegin(GL_LINE_STRIP) and glEnd().

For example, the code:

```
glBegin(GL_LINE_STRIP); // draw an open polyline
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();


glBegin(GL_LINE_LOOP); // draw an polygon
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();
```



Attributes such as color, thickness and stippling may be applied to Polylines in the same way they are applied to single lines. If it is desired to connect the last point with the first point to make the polyline into a polygon simply replace GL_LINE_STRIP with GL_LINE_LOOP.

Polygons drawn using GL_LINE_LOOP cannot be filled with a color or pattern. To draw filled polygons we have to use glBegin (GL_POLYGON).

**3. Drawing three dimensional objects & Drawing three dimensional scenes:**

OpenGL has separate transformation matrices for different graphics features glMatrixMode (GLenum mode), where mode is one of:

- GL_MODELVIEW - for manipulating model in scene
- GL_PROJECTION - perspective orientation
- GL_TEXTURE - texture map orientation

glLoadIdentity (): loads a 4-by-4 identity matrix into the current matrix

glPushMatrix() : push current matrix stack

glPopMatrix() : pop the current matrix stack

glMultMatrix () : multiply the current matrix with the specified matrix

glViewport() : set the viewport

Example: glViewport(0, 0, width, height);

gluPerspective() : function sets up a perspective projection matrix.

Format : gluPerspective(angle, as ratio, ZMIN, ZMAX);

Example : gluPerspective(60.0, width/height, 0.1, 100.0);

**gluLookAt()** - view volume that is centered on a specified eye point.

Example: gluLookAt (3.0, 2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

glutSwapBuffers () : glutSwapBuffers swaps the buffers of the current window if double buffered.

Example for drawing three dimension Objects:

```
glBegin (GL_QUADS); // Start drawing a quad primitive
glVertex3f (-1.0f, -1.0f, 0.0f); // The bottom left corner
glVertex3f (-1.0f, 1.0f, 0.0f); // The top left corner
glVertex3f (1.0f, 1.0f, 0.0f); // The top right corner
glVertex3f (1.0f, -1.0f, 0.0f); // The bottom right corner
glEnd();
//Triangle
glBegin( GL_TRIANGLES );
glVertex3f( -0.5f, -0.5f, -10.0 );
glVertex3f( 0.5f, -0.5f, -10.0 );
glVertex3f( 0.0f, 0.5f, -10.0 );
```

```
glEnd();
// Quads in different colours
glBegin(GL_QUADS);
glColor3f(1,0,0); //red
glVertex3f(-0.5, -0.5, 0.0);
glColor3f(0,1,0); //green
glVertex3f(-0.5, 0.5, 0.0);
glColor3f(0,0,1); //blue
glVertex3f(0.5, 0.5, 0.0);
glColor3f(1,1,1); //white
glVertex3f(0.5, -0.5, 0.0);
glEnd();
```

GLUT includes several routines for drawing these three-dimensional objects:

cone

icosahedrons

teapot

cube

octahedron

tetrahedron

dodecahedron

sphere

torus

**OpenGL Functions for drawing the 3D Objects**

glutWireCube (double size);

glutSolidCube (double size);

glutWireSphere (double radius, int slices, int stacks);

glutSolidSphere (double radius, int slices, int stacks);

glutWireCone (double radius, double height, int slices, int stacks);

glutSolidCone (double radius, double height, int slices, int stacks);

glutWireTorus (double inner radius, double outer radius, int sides, int rings);

glutSolidTorus (double inner radius, double outer radius, int sides, int rings);

glutWireTeapot (double size);

glutSolidTeapot (double size);

**3D Transformation in OpenGL**

glTranslate (): multiply the current matrix by a translation matrix

glTranslated(GLdouble x, GLdouble y, GLdouble z);

void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

x, y, z - Specify the x, y, and z coordinates of a translation vector.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after a call to glTranslate are translated. Use glPushMatrix and glPopMatrix to save and restore the untranslated coordinate system.

**glRotate()** : multiply the current matrix by a rotation matrix void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z); void glRotated(GLfloat angle, GLfloat x, GLfloat y, GLfloat z); angle : Specifies the

angle of rotation, in degrees. x, y, z : Specify the x, y, and z coordinates of a vector, respectively.

**glScale() :** multiply the current matrix by a general scaling matrix voidglScaled(GLdouble x, GLdouble y, GLdouble z); void glScalef(GLfloat x, GLfloat y, GLfloat z); x, y, z : Specify scale factors along the x, y, and z axes, respectively

**Example : Transformation of a Polygon**

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void Display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  glColor3f(0.0, 1.0, 0.0);
 glBegin(GL_POLYGON);
       glVertex3f( 0.0, 0.0, 0.0);        // V0 ( 0, 0, 0)
       glVertex3f( 1.0f, 0.0, 0.0);       // V1 ( 1, 0, 0)
```

```
                glVertex3f( 1.0f, 1.0f, 0.0);        // V2 ( 1, 1, 0)
                glVertex3f( 0.5f, 1.5f, 0.0);        // V3 (0.5, 1.5, 0)
                glVertex3f( 0.0, 1.0f, 0.0);         // V4 ( 0, 1, 0)
                glEnd();
                 glPushMatrix();
                 glTranslatef(1.5, 2.0, 0.0);
                 glRotatef(90.0, 0.0, 0.0, 1.0);
                glScalef(0.5, 0.5, 0.5);
                 glBegin(GL_POLYGON);
                glVertex3f( 0.0, 0.0, 0.0);          // V0 ( 0, 0, 0)
                glVertex3f( 1.0f, 0.0, 0.0);         // V1 ( 1, 0, 0)
                glVertex3f( 1.0f, 1.0f, 0.0);        // V2 ( 1, 1, 0)
                glVertex3f( 0.5f, 1.5f, 0.0);        // V3 (0.5, 1.5, 0)
                glVertex3f( 0.0, 1.0f, 0.0);         // V4 ( 0, 1, 0)
        glEnd();
        glPopMatrix();
glFlush();
glutSwapBuffers();
}
void Init(void)
{
glClearColor(0.0, 0.0, 0.0, 0.0);
}
void Resize(int width, int height)
{
   glViewport(0, 0, width, height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, width/height, 0.1, 1000.0);
   glMatrixMode(GL_MODELVIEW);

      glLoadIdentity();
   }
   int main(int argc, char **argv)
   {
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
      glutInitWindowSize(400, 400);
      glutInitWindowPosition(200, 200);
      glutCreateWindow("Polygon in OpenGL");
      Init();
      glutDisplayFunc(Display);
      glutReshapeFunc(Resize);
      glutMainLoop();
      return 0;
   }
```

## UNIT-VI

## Assignment-Cum-Tutorial Questions

## SECTION-A

### Objective Questions

1.The animation can be defined as a collection of images played in [ ]

a) Not sequence b) Defined sequence c) Both a & b d) None

2.To equalize vertex count in morphing no of points Np is calculated as

[ ]

a)int(Vmax-1/Vmin-1) b) int(Vmax+1/Vmin-1)

c) int(Vmax-1/Vmin+1) d)int(Vmax+1/Vmin+1)

3. _____consist of a set of rough sketches or it could be a list of the basic ideas for the motion. [ ]

a)Story board layout b) Object definitions

c) Key-frame system d) In-between frames

4. To equalize vertex count in morphing no of line sections Nls is calculated as [ ]

a) (Vmax-1)mod(Vmin-1) b) (Vmax+1)mod(Vmin-1)

c) (Vmax-1)mod(Vmin+1) d) (Vmax+1)mod(Vmin+1)

5. We can also animate objects along two-dimensional motion paths using

a)color-table transformations b) key-frames [ ]

c) languages d) functions

6. To equalize the edge count, and parameters Lk and Lk+1 denote the number of line segments in two consecutive frames. We then define

a) Lmax=max(Lk,Lk-1), Lmin=min(Lk,Lk-1)

b)Lmax=max(Lk,Lk+2),Lmin=min(Lk,Lk+2)

c) Lmax=min(Lk,Lk+1), Lmin=max(Lk,Lk+1)

d) Lmax=max(Lk,Lk+1), Lmin=min(Lk,Lk+1) [ ]

7. To equalize the vertex count, and parameters Vk and Vk+1 denote the number of vertices in two consecutive frames. We then define [ ]

a) Vmax=max(Vk,Vk-1), Vmin=min(Vk,Vk-1)

b) Vmax=max(Vk,Vk+2), Vmin=min(Vk,Vk+2)

c) Vmax=min(Vk,Vk+1), Vmin=max(Vk,Vk+1)

d) Vmax=max(Vk,Vk+1), Vmin=min(Vk,Vk+1)

8.Divide Ne edges of keyframemin into _____sections in preprocessing of morphing using edge count

a)Ns+1 b) Ns-1 c) Ns+2 d) Ns-2 [ ]

9. Divide the remaining lines of key frame min into ___ sections in [ ] preprocessing of morphing using edge count

a)Ns+1 b) Ns-1 c) Ns+2 d) Ns

10. Adding _____points to remaining edges of keyframem in preprocessing of morphing using vertex count [ ]

a) Np+1 b) Np-1 c) Np+2 d) Np

11. _____are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence. [ ]

a) Zero acceleration b) Non Zero acceleration

c) constant acceleration d) increasing acceleration

12._____ generally refers to any time sequence of visual changes in a scene.

13. Computer animations can also be generated by [ ]

a) Changing camera position b) Changing camera orientation

c) Changing camera focal length d) all of the above

14. An animation sequence is designed with which one of the following step

a) Storyboard layout c)Object definitions [ ]

b) Both a and c d)none of the above

15. _____ is given for each participant in the action [ ]

a) Storyboard layout b) Object definition

c) key-frame systems d) generation of in-between frames

16. _____ is an outline of the action.

17. _____is a detailed drawing of the scene at a certain time in the animation sequence.

18. _____are the intermediate frames between the key frames

19. Animation functions includes [ ]

a) Graphics editor b) a key-frame generator

c) An in-between generator d) all the above

20. List applications of computer animation?

21. When animating, OpenGL provides [ ]

a) A complete suite of tools and downloadable applications for making classic 2D and 3D animation right out of the box

b) FBOs, VBOs, VAOs and integer-related functions such as glFrameNumber and glMovieType

c) Accumulation buffers, frame-buffer objects, VBOs, depth and stencil buffers, blending modes, and other types of buffers that allow a developer to achieve the desired effect

d)No way to draw pixels on the screen

22. Generally, what primitive polygon is used for creating a mesh to represent a complex object? [ ]

a) Square b) Circle c) Triangle d) Rectangle

23.OpenGL stands for [ ]

a)Open General Liability b) Open Graphics Library

c) Open Guide Line d) Open Graphics Layer

## SECTION-B

### Descriptive Questions

1. What are the steps in design of animation sequence? Describe about each step briefly.
2. Discuss about general purpose languages used for animation.
3. Discuss about general computer animation functions
4. Write short note on raster animation
5. Define the term morphing and explain its use in key frame systems of animation
6. Describe linear list notation of animation languages
7. Explain about key frame systems in detail
8. Explain about motion specifications in animation
9. Explain in detail about Simulating Accelerations
10. Discuss how to equalize edge count and vertex count during preprocessing steps of morphing?

### Problems

1. Consider Lk = 15 and Lk+1 = 11 specify preprocessing rules for equalizing key frames in terms of edges?
2. Consider Vk = 3 and Vk+1 = 4 specify preprocessing rules for equalizing key frames in terms of vertices?.
3. Consider Lk = 18 and Lk+1 = 12 specify preprocessing rules for equalizing key frames in terms of edges?
4. Consider Vk = 6 and Vk+1 = 7 specify preprocessing rules for equalizing key frames in terms of vertices?.