USN | | | | | | | | | | |

13MCA34

Third Semester MCA Degree Examination, Dec.2016/Jan.2017
Computer Graphics

Time: 3 hrs. Max. Marks:100

Note: ***Answer any FIVE full questions.***

1 a. Explain the basic syntax of open GL with simple program. **(08 Marks)**
b. Briefly discuss the open GL point functions. **(06 Marks)**
c. Explain in detail open GL line functions with example. **(06 Marks)**

2 a. Describe DDA line drawing algorithm's merits and demerits. **(04 Marks)**
b. Demonstrate midpoint circle algorithm with one example. **(10 Marks)**
c. Briefly explain – Boundary filling algorithm. **(06 Marks)**

3 a. Explain the basic 2D geometric transformations with equations. **(10 Marks)**
b. Discuss the inverse transformations. **(04 Marks)**
c. Write short notes on:
(i) Reflection (ii) Shear **(06 Marks)**

4 a. Explain in detail 3D translation and 3D scaling. **(10 Marks)**
b. Write a open GL program to rotate the cube about 90° in clockwise with respect to Z axis. **(10 Marks)**

5 a. With neat diagram explain 2D viewing transformation pipeline. **(06 Marks)**
b. Explain the Cohen-Sutherland line clipping algorithm with diagram. **(10 Marks)**
c. Briefly explain about text clipping. **(04 Marks)**

6 a. How modeling co-ordinates are translated into viewing co-ordinates in 3D pipeline? **(10 Marks)**
b. Explain oblique parallel projections with diagram. **(10 Marks)**

7 a. Explain in detail Beizer-Spline curves. **(10 Marks)**
b. Describe the basic approach to design animation sequence. **(06 Marks)**
c. Differentiate traditional animation and computer animation techniques. **(04 Marks)**

8 Write a short notes on:
a. Bresenham's line drawing.
b. Affine Transformations.
c. Depth cueing.
d. Orthogonal projection. **(20 Marks)**

* * * * *

Computer Graphics - VTU Final Exam (December 2016)

Q1.(a) Explain the basic syntax of openGL with a simple program.
Sol:
<u>Introduction to OpenGL</u>

- It is a basic library of functions for specifying graphics primitives, attributes, geometric transformations, viewing transformations and many other operations.
- It is hardware independent and thus input and output routines are not part of the basic library. Normally we have these input output and other useful routines provided by an auxillary library.

<u>Basic OpenGL syntax</u>

- OpenGL Function names are prefixed with gl and each component has its name capitalized e.g. glBegin
- Symbolic constants begin with capital letters GL. component names arewritten in capital with underscore between them e.g. GL_LINES
- Different data types available in openGL are. It has its own data type because data type sizes are not sttandardized and may be different on different machines. e.g. a typical integer may be 16 bit, 32 bit or 64 bit depending on the machine. Therefore OpenGL has its own datatype specification which is independent of machines. The data types are
    - GLbyte
    - GLshort
    - GLint
    - GLfloat
    - GLdouble
    - GLboolean
- The OpenGL Utility(GLU) library provides routines for viewing and projection matrices describing complex objects with line and polygon approximations, surface rendering, and other complex tasks. GLUT (OpenGL Utility Toolkit) routines, prefixed with glut, provide a toolkit that is dependent on the window system. This set of routines also contains methods for describing and rendering quadric curves and surfaces.
- Header Files:
- HEADER FILES

#include <GL/gl.h>	Includes the OpenGL core header file. This file is required by all OpenGL applications.
#include <GL/glu.h>	Includes the OpenGL Utility Library header file. This file is needed by most OpenGL applications.

- And we would also need a window interface library. Alternatively, we can just include the GLUT header file:

#include <GL/glut.h>	Includes the OpenGL Utility Toolkit header file. This statement automatically includes gl.h, glu.h, and glx.h. And with Microsoft Windows, it includes the appropriate header file to access WGL.

#include <GL/glut.h>	Includes the OpenGL Utility Toolkit header file. This statement automatically includes gl.h, glu.h, and glx.h. And with Microsoft Windows, it includes the appropriate header

	file to access WGL.

SETTING UP DISPLAY WINDOWS USING GLUT

glutInit (options);	Initializes GLUT and specifies command-line options for the window system in use. This function should be called before any other GLUT routine.
glutInitWindowPosition (x, y);	Specifies the screen position for top left corner of the window. Postition is specified in integer coordinates whose origin is the upper left corner of the screen
glutInitWindowSize (w, h);	Specifies the width (w) and height (h) for the window in pixels (integer values).
glutCreateWindow (string);	Opens a window with the previously specified size, position, and other properties. The specified text string may be displayed in the window title bar, depending on the options available in the window system.
glutInitDisplayMode	Various options for the window are set with this function. These options include the color mode and single or double buffering. The default is RGB color mode with single buffering. e.g. glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB)
glutDisplayFunc(<function pointer>)	Specifies the content of the display window. We first create the picture to be displayed and then pass on the picture definition to the function.
glutMainLoop()	Activates all display windows created including their content. This function must be the last line in the program . It puts the program into an infinite loop and checks inputs from various devices.

Other GL functions

glClearColor(1.0,1.0,1.0,0.0)	Sets the background color where the first three arguments are the RGB values in the range [0,1] and the fourth is the alpha value. One use of alpha is as a "blending parameter" . When multiple objects can overlap in a scene the blending parameter determines the resulting color of two objects. A 0 value could mean a totally transperent object and 1 would mean opaque.
glClear(GL_COLOR_BUFFER_BIT)	To get the assigned color displayed we need to invoke this function. In this case GL_COLOR_BUFFER_BIT indicates that the bits in the buffer bits need to be set to color specified in glClearColor
glColor*	To set the color of the objects to be displayed. The star can be replaced by text according the data type and the type of parameter passed. E.g. glColor3f(1.0,0.0,0.0) . Here 3 means three color

	components(R,G,B) would be passed and 'f' means that the arguments are floating point numbers whose values are in the range [0,1].
glMatrixMode(GL_PROJECTION)	Sets the projection type.
gluOrho2D(0.0,200.0,0.0,150.0)	This specifies the viewing parameter i.e. the orthogonal projection is to be used to map the contents pf a 2D rectangular area of world coordinates to the screen and the a coordinates have range[0,200] and y coordinates are in range [0,150]. Whatever objects are in this window are displayed and others are not displayed. Thus gluOrtho2D defines the coordinate reference frame within the display window to be (0,0) at the lower left corner of display window and (200,150) to be the top right corner. When we set up the geometry describing a picture, all positions for the OpenGL primitives must be given in absolute coordinates, with respect to the reference frame defined in the gluOrtho2D function.
glFlush()	Forces execution of OpenGL functions stored in various buffers.

The glutDisplayFunc and other similar functions normally require a function name to be passed as a parameter . E.g. we pass the function "linesegment" to glutDisplayFunc . These functions are called display callback functions. This procedure is registered with glutDisplayFunc as the routine to be invoked when the display window might need to be re-displayed. For e.g. this can happen when the display window is moved. . OpenGL is organized as a set of callback functions that are to be invoked when certain actions occur.

(b) Briefly discuss openGL point function.
Sol:
OpenGL POINT FUNCTIONS

Unless we specify other attribute values, OpenGL primitives are displayed with a default size and color. The default color for primitives is white and the default point size is equal to the size of one screen pixel. The function for plotting a point is glVertex* ( );
where the asterisk (*) indicates that suffix codes which may consist of three parts
spatial dimension,
the numerical data type to be used for the coordinate values,
a possible vector form for the coordinate specification.

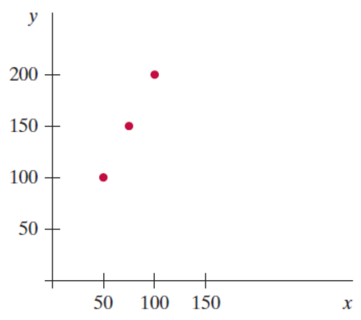
A glVertex function must be placed between a glBegin function and a glEnd function. The argument of the glBegin function is used to identify the kind of output primitive that is to be displayed. For point plotting, the argument of the glBegin function is the symbolic constant GL POINTS. Example:

```
glBegin (GL_POINTS);
        glVertex* ( );
glEnd ( );
```

We use a suffix value of 2, 3, or 4 on the glVertex function to indicate the dimensionality of a coordinate position. A four-dimensional specification indicates a homogeneous-coordinate representation. The data type is to be used for the numerical-value specifications of the coordinates. is specified using the second suffix code on the glVertex function. Suffix codes for data type
are : i (integer), s (short), f (float), and d (double). Finally, the coordinate values can be listed explicitly in the glVertex function, or a single argument can be used that references a coordinate position as an array. If we use an array specification for a coordinate position, we need to append a third suffix code: v (for“vector”). In the following:

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

This will plot the three points as in fig.



The vertices are specified in 2 dimensions - justifying the suffix '2' and are specified in integer coordinates and thus suffix 'i'. Alternatively, we could specify the coordinate values for the preceding points in arrays such as

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

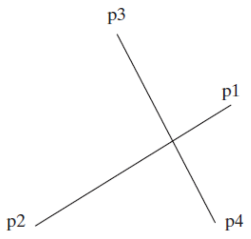
and call the OpenGL functions for plotting the three points as

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

Here suffix 'v' stands for vector since the points are specified using array.

(c) Explain in detail OpenGL line function with example.
Sol:
Endpoint coordinate position of lines are specified using the glVertex function. While defining a line the symbolic constant used in glBegin is GL_LINES, GL_LINE_STRIP or GL_LINE_LOOP.
GL_LINES: A set of straight-line segments between each successive pair of endpoints in a list is generated using the primitive line constant GL LINES. In general, this will result in a set of unconnected lines unless some coordinate positions are repeated. If there are odd number of points specified then the last line corresponding to the unpaired vertex is not drawn. For.e.g.

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

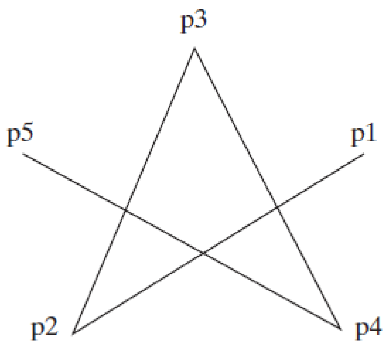


We obtain one line segment between the first and second coordinate positions, and another line segment between the third and fourth positions. In this case, the number of specified endpoints is odd, so the last coordinate position is ignored.

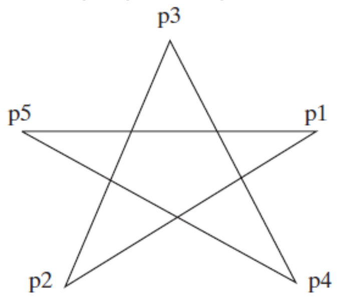
GL-LINE-STRIP: We obtain a polyline. In this case, the display is a sequence of connected line segments between the first endpoint in the list and the last endpoint. The first line segment in the polyline is displayed between the first endpoint and the second endpoint; the second line segment is between the second and third end points; and so forth, up tto the last line endpoint. Nothing is displayed if we do not list at least two coordinate positions. Thus a specification like:

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

will display the figure



GL_LINE_LOOP: It produces a closed polyline. An additional line is added to the line sequence from the previous example, so that the last coordinate endpoint in the sequence is connected to the first coordinate endpoint of the polyline. For the same set of commands as above the following figure is generated.



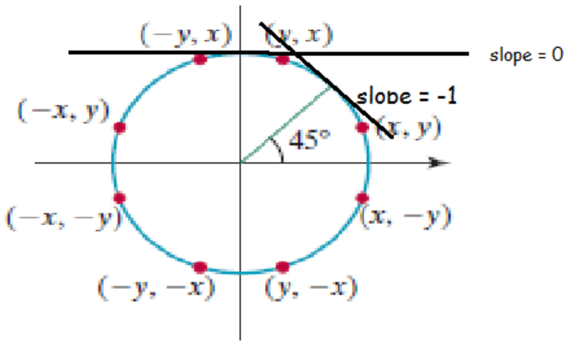
Q2. (a) Describe DDA algorithm's merits and demerits
Sol:
Advantages of DDA algorithm

It is faster than the method of drawing line using Eq.(1) since it avoids multiplication and only involves addition or subtraction at every step.

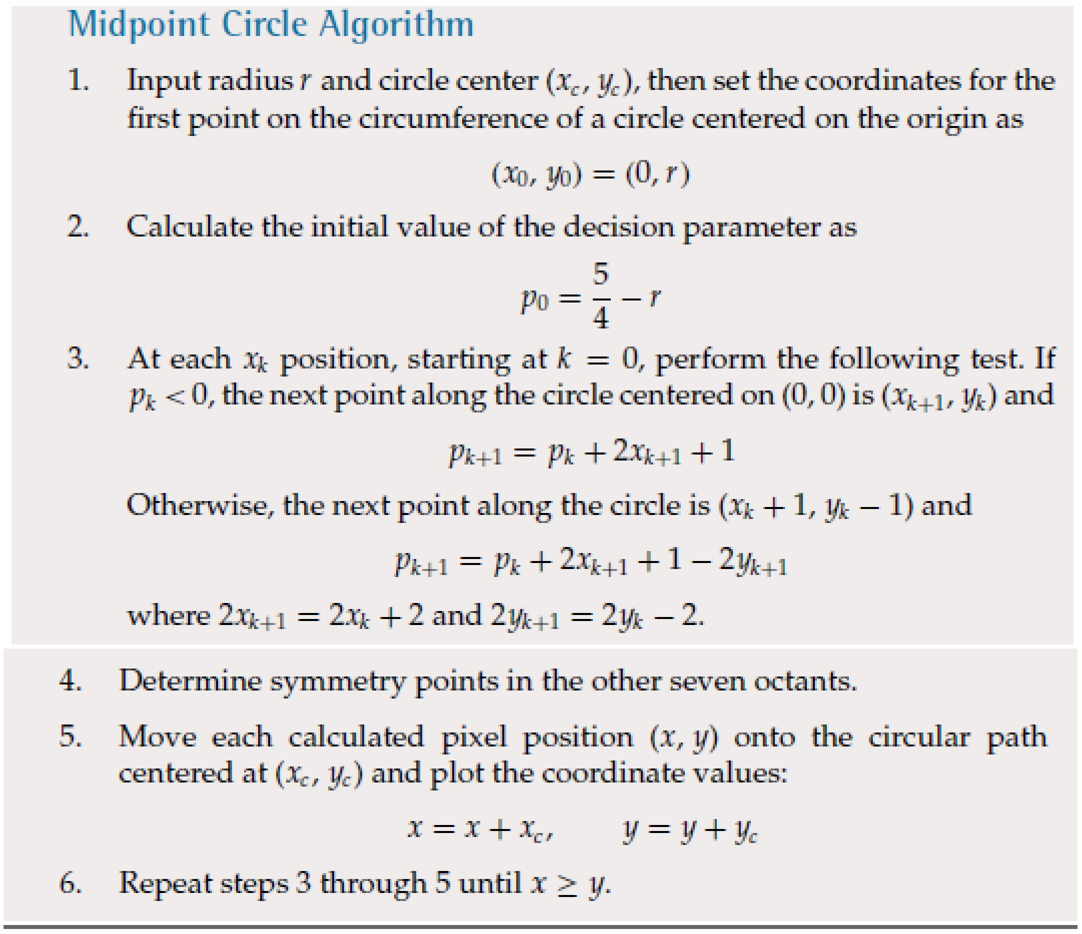
Disadvantages

- 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.
- The rounding operations and floating-point arithmetic in this procedure are still time consuming.

(b) Demonstrate midpoint circle drawing algorithm with an example:
Sol:



The basic idea behind the midpoint circle algorithm is to vary x by one unit and find corresponding positions of y i.e. to determine whether the next point is (x+1,y) or (x+1,y-1). For this test the halfway position between two pixels i.e. (x+1,y) or (x+1,y-1) to determine if this midpoint is inside or outside the circle boundary.
The algorithm is given by



Midpoint Circle Algorithm

1. Input radius r and circle center (x_c, y_c), then set the coordinates for the first point on the circumference of a circle centered on the origin as
$$(x_0, y_0) = (0, r)$$
2. Calculate the initial value of the decision parameter as
$$p_0 = \frac{5}{4} - r$$
3. At each x_k position, starting at $k = 0$, perform the following test. If $p_k < 0$, the next point along the circle centered on $(0, 0)$ is (x_{k+1}, y_k) and
$$p_{k+1} = p_k + 2x_{k+1} + 1$$
Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and
$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$
where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$.
4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x, y) onto the circular path centered at (x_c, y_c) and plot the coordinate values:
$$x = x + x_c, \qquad y = y + y_c$$
6. Repeat steps 3 through 5 until $x \geq y$.

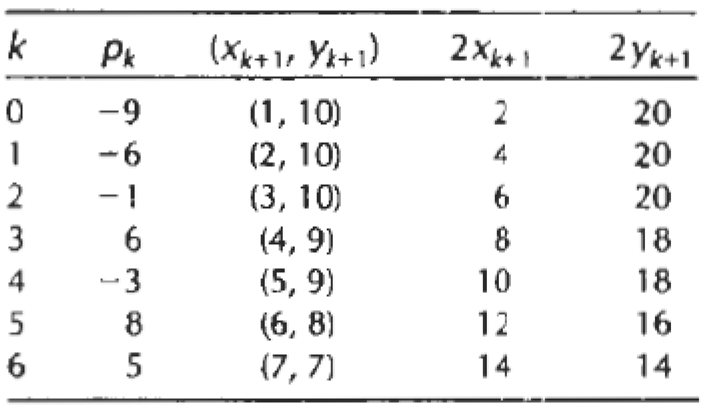
Given a circle radius r = 10, we demonstrate the midpolnt circle algorithm by determining positions along the circle octant in the first quadrant hum x = 0 to x = y. The initial value of the decision parameter is

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

For the circle centered on the coordinate origin, the initial point is (x,, yo) - (0, lo), and initial increment terms for calculating the dxision parameters are

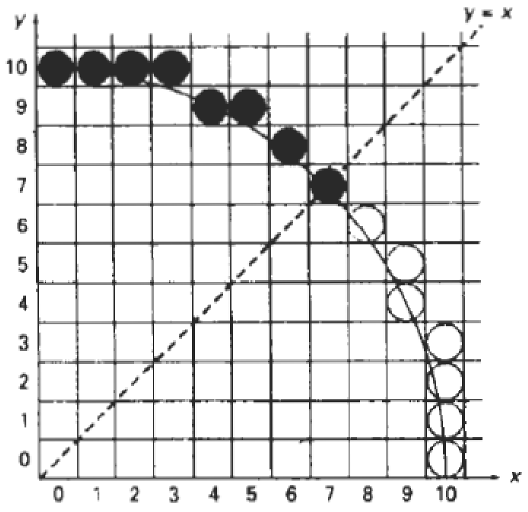
$$2x_0 = 0, \qquad 2y_0 = 20$$

Successive decision parameter values and positions along the circle path are calculated using the midpoint method as



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

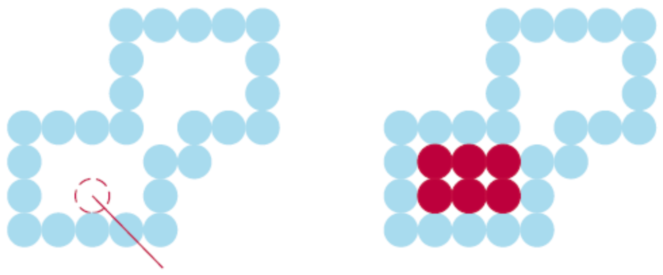
A plot c)f the generated pixel positions in the first quadrant is shown in



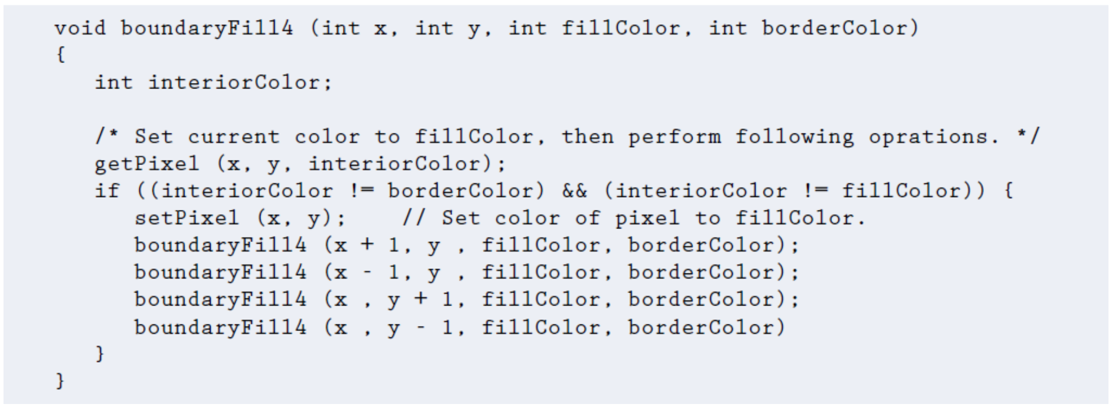
(c) Briefly explain boundary fill algorithm

Sol: Boundary-Fill Algorithm

If the boundary of some region is specified in a single color, we can fill the interior of this region, pixel by pixel, until the boundary color is encountered. This method called the boundary-fill algorithm, is employed in interactive painting packages, where interior points are easily selected. a boundary-fill algorithm starts from an interior point (x, y) and tests the color of neighboring positions. If a tested position is not displayed in the boundary color, its color is changed to the fill color and its neighbors are tested. This procedure continues until all pixels are processed upto the designated boundary color for the area. four neighboring points are tested. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called 4-connected. The second method, called 8 connected is used to fill more complex figures. Here the set of neighboring positions to be tested includes the four diagonal pixels, as well as those in the cardinal directions. Some figures which ca be correctly filled by 8 - connected method cannot be handled by 4-connected. Example is the figure given below, which can only be partially filled using 4-connected.



The psuedocode for 4-connected boundary fill is given below:



```
void boundaryFill4 (int x, int y, int fillColor, int borderColor)
{
   int interiorColor;

   /* Set current color to fillColor, then perform following oprations. */
   getPixel (x, y, interiorColor);
   if ((interiorColor != borderColor) && (interiorColor != fillColor)) {
      setPixel (x, y);    // Set color of pixel to fillColor.
      boundaryFill4 (x + 1, y , fillColor, borderColor);
      boundaryFill4 (x - 1, y , fillColor, borderColor);
      boundaryFill4 (x , y + 1, fillColor, borderColor);
      boundaryFill4 (x , y - 1, fillColor, borderColor)
   }
}
```

This recursive method paints a 4-connected area with a solid color, specified in parameter fillColor, up to a boundary color specified with parameter borderColor. If some pixel in the interior region is already filled with this color then it might cause the recursive algorithm to not fill some of the pixels. A solution is to remove the color of pixels in the interior which are colored with fill color before starting to fill.

Disadvantage:

- For a large area the number of neighbouring pixels stacked could be very large because of recursion.
- Is difficult to use for a shape not bounded by the same color.

Q3. (a) Explain basic 2D transformations with equations.
Sol:
It is often necessary in applications to apply changes in orientation, size, and shape of objects and are accomplished with geometric transformations that alter the coordinate descriptions of objects. The basic geometric transformations are translation, rotation, and scaling. There could be other transformations defined too.

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, f, and t,, to the original coordinate position (x, y) to move the point to a new position (x ' , y')

$$x' = x + t_x , \qquad y' = y + t_y ,$$

(t_x,t_y) is called a translation vector or shift vector. If P is the original point, P' is the new position and T is the translation vector

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

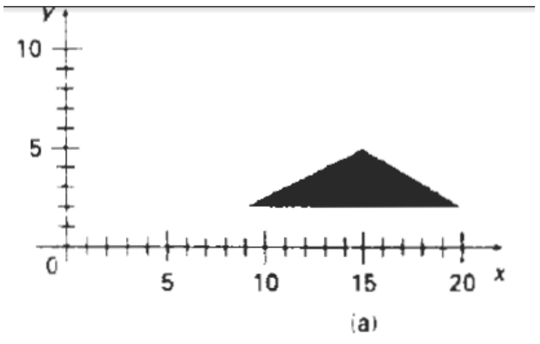
Then the translation operation can be expressed as:
P'=P+T

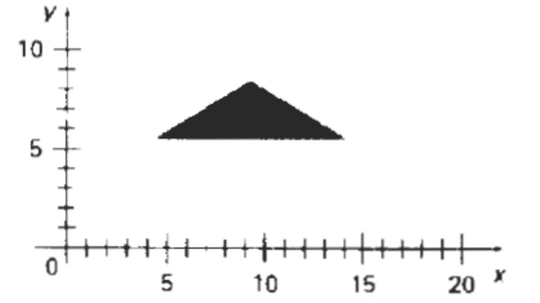
Written in homogeneous coordinate system:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translation is a rigid-body transformation that moves objects without deformation. That is, every point on the object is translated by the same amount and the length of any object part remains unchanged. Polygons are moved by translating the vertices and redrawing the polygon. Similarly for circle the centre can be translated and the circle redawn.

Example of translation





Rotation

A rotation transformation is specified using a rotation axis and a rotation angle. All points are rotated about the axis through the rotation angle. For a 2D rotation happens in XY plane with axis of rotation the Z axis.

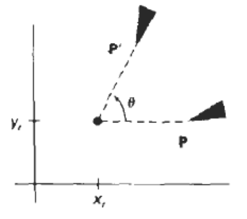


Figure 5-3
Rotation of an object through angle θ about the pivot point (x_r, y_r).

The rotation point is also called the pivot point. Let us assume that the pivot point is (0,0) and angle of rotation. Let (x,y) be the point to be rotated. The figure below shows the point before and after rotation.

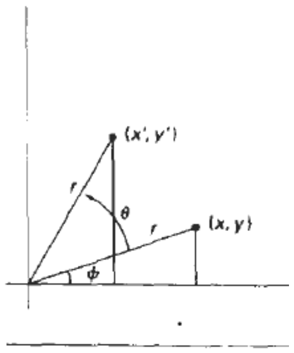


Figure 5-4
Rotation of a point from position (x, y) to position (x', y') through an angle θ relative to the coordinate origin. The original angular displacement of the point from the x axis is ϕ.

Let $\emptyset$ be the angle that the line joining the origin to (x,y) makes with the x axis. According the the definition of cos and sin

$$x' = r\cos(\emptyset + \theta) = r\cos\emptyset\cos - r\sin\emptyset\sin\theta \qquad ---(1)$$

$$y' = r\sin(\emptyset + \theta) = r\cos\emptyset\sin\theta + r\sin\emptyset\cos\theta \qquad ---(2)$$

where $r = \sqrt{x^2 + y^2}$

We also know that $\cos\emptyset = \frac{x}{r}$, and therefore $x = r\cos\emptyset$

Similarly $y = r\sin\emptyset$. This substituting these values in (1) and (2)

$$x' = x\cos\theta - r\sin\theta \quad \text{and} \quad y' = x\sin\theta + y\cos\theta$$

If P is the original point P=$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ and P' is the transformed point then P ' = R(θ).P

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

- Polygons are rotated by applying transformations to each vertex and redrawing the polygon. Similarly other figures such as circle is rotated by rotating its centre and redrawing it with the given radius.
- Rotation is a rigid body transformation i.e. it does not change the size or shape of the object, only its position.

Two Dimensional Scaling

Scaling is used to alter the size of an object.The figure below shows a square being scaled twice in the x direction.

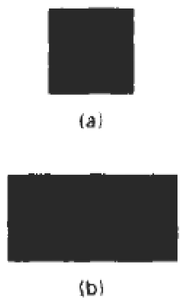


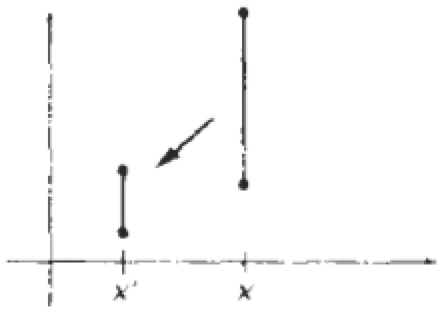
Figure 5-6
Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

A 2D scaling is performed by multiplying object positions(x,y) by scaling factors sx and sy.
x=x' . sx   and y' = sy . y
where sx is the scaling factor in x direction and sy is the scaling factor in the y direction.
The transformation can be written in the matrix form as :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or P' = S . P where S is the scaling matrix.

- Values less than 1 reduce the size of the object
- Values > 1 enlarge the object
- When sx = sy a uniform scaling is produced and when sx is not equal to sy it is called differential scaling.
- When negative scaling parameters are specified then not only scaling but also reflection happens about one of the coordinate axes
- Scaling also repositions objects , i.e. scaling factor > 1 moves objects away from origin
- and scaling factor < 1 brings them closer to the origin as shown in the figure below where the scaling factor is 0.5 in both directions:



- Polygons are scaled by applying transformations to each vertex and redrawing the polygon.
- For circle the radius is scaled and for ellipse both the axes are scaled. and the points are redrawn.

(b) Discuss inverse transformations.
Sol:
Inverse transformations bring the object already transformed to its original position.

Inverse translation:
Consider an object translated by (tx, ty) . The inverse transformation would be to translate the object by an amount (-tx,-ty). Thus a composite transformation of a translation followed by inverse translation would result in the object being in its original position. The transformation matrix for inverse translation is:

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

We find that the composite transformation of a translation followed by the inverse translation T(-tx,-ty). T(tx,ty) = I(identity matrix)

$$T(-tx,-ty).\ T(tx,ty) = \begin{bmatrix} 1 & 0 & -tx \\ 0 & 1 & -ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Inverse Rotation
An inverse rotation is to perform a rotation by angle $-\theta$ where the original rotation was by angle θ.
The rotation matrix

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

Thus the composite transformation comprising of rotation followed by inverse

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

Inverse Scaling
An inverse scaling is to perform a scaling by 1/sx and 1/sy where the original scaling was by sx and sy in the x and y direction respectively.
The inverse scaling matrix

$$\begin{bmatrix} 1/sx & 0 & 0 \\ 0 & 1/sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus the composite transformation comprising of rotation followed by inverse

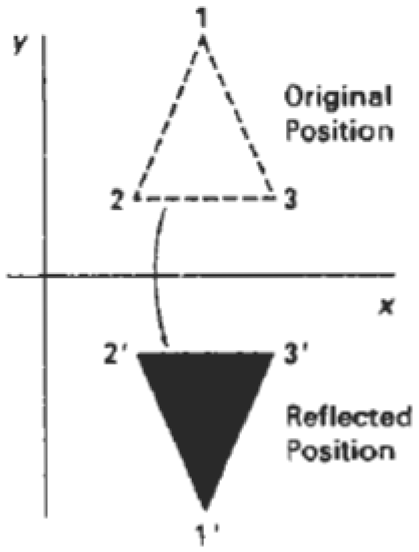
$$S(1/sx,1/sy).S(sx,sy)= \begin{bmatrix} 1/sx & 0 & 0 \\ 0 & 1/sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(c) Write short notes on reflection and shear.
Sol:
Reflection

A reflection is a transformation that produces a mimr image of an obpct. 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. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane. When the reflection axis is a line in the xy plane, the rotation path about this axis is in a plane perpendicular to the xy plane. For reflection axes that are perpendicular to the xy plane, the rotation path is in the xy plane. Following are examples of some common reflections. 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. The resulting orientation of an object after it has been reflected about the x axis is shown below:



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

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

Shear
A transformation that distorts 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 w values and those that shift y values. An x-direction shear relative to the x axis is produced with the transformation matrix

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

Any real number can be assigned to the shear parameter sh,. A coordinate position (.u, y) is then shifted horizontally by an amount proportional to its distance (y value) from the x axis (y = 0). Negative values for sh, shift coordinate positions to the left.
We can generate x-direction shears relative to other reference lines with

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

Q4.Explain the detail 3D translation and 3D scaling.
Sol:
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 3-dimensional point by adding translation distances (tx,ty,tz) to the original coordinate position (x,y,z) to move the point to a new position (x', y',z').
x'=x+tx, y'=y+ty, z'=z+tz

(t_x,$t_{y,}$ tz) is called a translation vector or shift vector. If P is the original point, P' is the new position and T is the translation vector P'=P+T

Written in homogeneous coordinate system:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translation is a rigid-body transformation that moves objects without deformation. That is, every point on the object is translated by the same amount and the length of any object part remains unchanged. Polygons are moved by translating the vertices and redrawing the polygon. Similarly for circle the centre can be translated and the circle redawn.
<u>3D Scaling</u>
Scaling is used to alter the size of an object.A 3D scaling is performed by multiplying object positions(x,y,z) by scaling factors sx,sy and sz.
x=x' . sx and y' = sy . y, z' = sz . z
where sx is the scaling factor in x direction, sy is the scaling factor in the y direction, sz is the scaling factor in the z direction.
The transformation can be written in the matrix form as :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or P' = S . P where S is the scaling matrix.

- Values less than 1 reduce the size of the object
- Values > 1 enlarge the object
- When sx = sy a uniform scaling is produced and when sx is not equal to sy it is called differential scaling.
- When negative scaling parameters are specified then not only scaling but also reflection happens about one of the coordinate axes

- Scaling also repositions objects , i.e. scaling factor > 1 moves objects away from origin
- and scaling factor < 1 brings them closer to the origin as shown in the figure below where the scaling factor is 0.5 in both directions:

b) Write an openGL program to rotate a cube by 90 degree in clockwise direction about Z axis.

Sol:

```
#include <GL/glut.h>
#include <math.h>
#include <stdio.h>
#include <string.h>

#define X 1
#define Y 2
#define Z 3

float iden[4][4]={0};

typedef struct point // will store the lowest  point in the cube
{
        float x, y, z,h; //x , y,z coordinates
}point;

// it is expected that the first4 i.e. x[0]..x[3] points represent one face of the cube
// and x[4]..x[7] represent the opposite face
typedef struct cube
{
        point x[8];
}cube;


GLint w=600,h=600;
GLfloat x0=30,ye=10,z0=200;
//GLfloat x0=100,ye=0,z0=100;
GLfloat xref=0,yref=0,zref=0;
GLfloat vx=0,vy=1,vz=0;

GLfloat xwMin=-300, ywMin=-300, xwMax = 300, ywMax=300;
GLfloat dnear = 0, dfar=400;


void init(void)
{
        glClearColor(1.0,1,1,0.0);
        glMatrixMode(GL_MODELVIEW);
        gluLookAt(x0,ye,z0,xref,yref,zref,vx,vy,vz);
        glMatrixMode(GL_PROJECTION);
        glOrtho(xwMin, xwMax,ywMin, ywMax,dnear,dfar);
        //gluPerspective(120, 1, 0, 400);
}

point transform(float mat[4][4], point p)
{
        point trans_p;

        trans_p.x=mat[0][0]*p.x+mat[0][1]*p.y+mat[0][2]*p.z+mat[0][3]*p.h;
        trans_p.y=mat[1][0]*p.x+mat[1][1]*p.y+mat[1][2]*p.z+mat[1][3]*p.h;
        trans_p.z=mat[2][0]*p.x+mat[2][1]*p.y+mat[2][2]*p.z+mat[2][3]*p.h;
        trans_p.h=mat[3][0]*p.x+mat[3][1]*p.y+mat[3][2]*p.z+mat[3][3]*p.h;

        return trans_p; // CHANGE
}
void applytrans_obj(float mat[4][4], point p[], int n)
{
        int i;
        for (i=0;i<n;i++)
        {
                p[i]=transform(mat,p[i]);
                printf("Point after transformation is \n");
                printf("%f %f %f ",p[i].x,p[i].y,p[i].z);
                printf("\n");
        }
}

cube applytrans_cube(float mat[4][4], cube c)
{
        int i;
        for (i=0;i<8;i++)
        {
                c.x[i]=transform(mat,c.x[i]);
                printf("Point after transformation is \n");
                printf("%f %f %f ",c.x[i].x,c.x[i].y,c.x[i].z);
                printf("\n");
        }
        return c; // CHANGE
}

void rotate(float angle, char axis,float mat[4][4])
{
        int i,j;
        float c=cos(angle*3.14/180), s=sin(angle*3.14/180);

        memcpy(mat,iden,sizeof(float)*4*4);
        if ( axis == X)
        {
                mat[1][1]=mat[2][2]=c;
```

```
		mat[1][2]-=mat[2][1]=s;
	}
	else if (axis == Y)
	{
		mat[0][0]=mat[2][2]=c;
		mat[2][0]-=mat[0][2]=s;
	}
	else
	{
		mat[0][0]=mat[1][1]=c;
		mat[0][1]-=mat[1][0]=s;
	}
}



void drawCube(cube c)
{
	//back surface of sube

	glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
	glBegin(GL_LINE_LOOP);
		glVertex3f(c.x[0].x,c.x[0].y,c.x[0].z);
		glVertex3f(c.x[1].x,c.x[1].y,c.x[1].z);
		glVertex3f(c.x[2].x,c.x[2].y,c.x[2].z);
		glVertex3f(c.x[3].x,c.x[3].y,c.x[3].z);
	glEnd();

	// front surface
	glBegin(GL_LINE_LOOP);
		glVertex3f(c.x[4].x,c.x[4].y,c.x[4].z);
		glVertex3f(c.x[5].x,c.x[5].y,c.x[5].z);
		glVertex3f(c.x[6].x,c.x[6].y,c.x[6].z);
		glVertex3f(c.x[7].x,c.x[7].y,c.x[7].z);
	glEnd();

	// Connect two surfaces with lines to form side surfaces
	glBegin(GL_LINES);
		glVertex3f(c.x[0].x,c.x[0].y,c.x[0].z);
		glVertex3f(c.x[4].x,c.x[4].y,c.x[4].z);
		glVertex3f(c.x[1].x,c.x[1].y,c.x[1].z);
		glVertex3f(c.x[5].x,c.x[5].y,c.x[5].z);
		glVertex3f(c.x[2].x,c.x[2].y,c.x[2].z);
		glVertex3f(c.x[6].x,c.x[6].y,c.x[6].z);
		glVertex3f(c.x[3].x,c.x[3].y,c.x[3].z);
		glVertex3f(c.x[7].x,c.x[7].y,c.x[7].z);
	glEnd();

	glFinish();
	glViewport(0,0,w,h);
```

```
}


void display(void)
{
	cube  c;
	triangle t;

	int i,l=50;
	float mat[4][4]={0},mat1[4][4],mat2[4][4];

	c.x[0].x=c.x[0].y=c.x[0].z=0;
	c.x[1].x=l;c.x[1].y=c.x[1].z=0;
	c.x[2].x=c.x[2].y=l;c.x[2].z=0;
	c.x[3].x=0;c.x[3].y=l;c.x[3].z=0;

	c.x[4].x=c.x[4].y=0;c.x[4].z=l;
	c.x[5].x=l;c.x[5].y=0,c.x[5].z=l;
	c.x[6].x=c.x[6].y=c.x[6].z=l;
	c.x[7].x=0;c.x[7].y=c.x[7].z=l;

	c.x[0].h=c.x[1].h=c.x[2].h=c.x[4].h=c.x[5].h=c.x[6].h=c.x[7].h=c.x[3].h=1; //
CHANGE

	t.x[0].x=0; t.x[0].y=50; t.x[0].z=-100;
	t.x[1].x=0; t.x[1].y=50; t.x[1].z=-200;
	t.x[2].x=0; t.x[2].y=70; t.x[2].z=-150;

	t.x[0].h=t.x[1].h=t.x[2].h=t.x[3].h=0;

	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(0.0,0.0,0.0) ;

	glColor3f(0.0,1.0,0.0) ;
	drawCube(c);
	rotate(-90,Z,mat);

	applytrans_obj(mat,c.x,8);
	glColor3f(1.0,0.0,0.0) ;
	drawCube(c);

	glFlush();
}



int main(int argc,char *argv[])
{
	iden[0][0]=iden[1][1]=iden[2][2]=iden[3][3]=1;
```

```
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowPosition(50,50);
        glutInitWindowSize(w,h);
        glutCreateWindow("My First Window");

        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshapeFcn);
        glutMainLoop();

        return 0;
}
```

Q5. With a neat diagram explain 2D viewing transformation pipeline.

Sol:

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.
Often, 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. Sometimes the two-dimensional viewing transformation is simply referred to as the window-to-viewport transformation or the windowing transformation. By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, 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).

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. Object descriptions are then transferred
to normalized device coordinates. We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates.

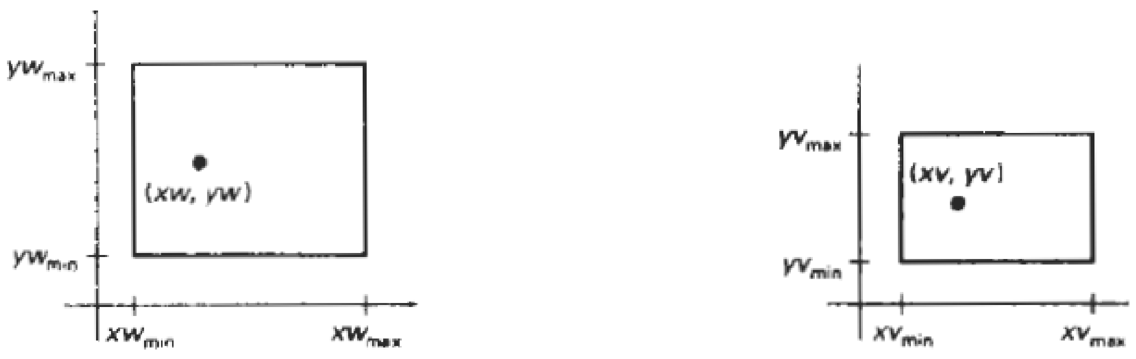


Figure 6-5
A point at position (xw, yw) in a designated window is mapped to viewport coordinates (xv, yv) so that relative positions in the two areas are the same.

In the above figure the point (xw,yw) in clipping coordinates have to be mapped to the point (xv,yv) in the viewport.

To maintain the same relative placement in the viewport as in the window, we require 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 we find that

$$xv - xv_{min} = \frac{(xw - xw_{min})(xv_{max} - xv_{min})}{xw_{max} - xw_{min}}$$

Simplifying:

$$xv = xv_{min} + \frac{(xw - xw_{min})(xv_{max} - xv_{min})}{xw_{max} - xw_{min}}$$

Expanding the second term

$$xv = xv_{min} + \frac{xw.xv_{max} - xw_{min}xv_{max} - xw.xv_{min} + xw_{min}.xv_{min})}{xw_{max} - xw_{min}}$$

$$xv = \frac{xv_{min}(xw_{max} - xw_{min}) + xw.xv_{max} - xw_{min}xv_{max} - xw.xv_{min} + xw_{min}.xv_{min})}{xw_{max} - xw_{min}}$$

$$xv = \frac{xv_{min}xw_{max} - \cancel{xv_{min}xw_{min}} + \boldsymbol{xw.xv_{max} - xw_{min}xv_{max} - xw.xv_{min}} + \cancel{\boldsymbol{xw_{min}.xv_{min}}})}{\boldsymbol{xw_{max} - xw_{min}}}$$

$$\boldsymbol{xv = \frac{xv_{min}xw_{max} + xw.xv_{max} - xw_{min}xv_{max} - xw.xv_{min}}{xw_{max} - xw_{min}}}$$

$$\boldsymbol{xv = xw.\frac{(xv_{max} - xv_{min})}{xw_{max} - xw_{min}} + \frac{xv_{min}xw_{max} - xw_{min}xv_{max}}{\mathrm{xw_{max} - xw_{min}}}}$$

In a similar manner we can derive the expression for yv as

$$yv = yw.\frac{(yv_{max} - yv_{min})}{yw_{max} - yw_{min}} + \frac{yv_{min}yw_{max} - yw_{min}yv_{max}}{yw_{max} - yw_{min}}$$

By solving these equations for the unknown viewport position (xv, yv), the following becomes true:

$$xv = s_x xw + t_x$$

$$yv = s_y yw + t_y$$

where

$$s_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}} \qquad t_x = \frac{xw_{max}\,xv_{min} - xw_{min}\,xv_{max}}{xw_{max} - xw_{min}}$$

$$s_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}} \qquad t_y = \frac{yw_{max}\,yv_{min} - yw_{min}\,yv_{max}}{yw_{max} - yw_{min}}$$

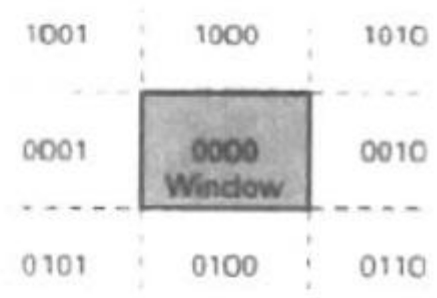
Thus the transformation matrix for the mapping is

$$M=\begin{bmatrix} \frac{(xv_{max}- xv_{min})}{xw_{max}-xw_{min}} & 0 & \frac{xv_{min}xw_{max}-xw_{min}xv_{max}}{xw_{max}-xw_{min}} \\ 0 & \frac{(yv_{max}- yv_{min})}{yw_{max}-yw_{min}} & \frac{yv_{min}yw_{max}-yw_{min}yv_{max}}{yw_{max}-yw_{min}} \\ 0 & 0 & 1 \end{bmatrix}$$

(b) Explain Cohen Sutherland line clipping algorithm with diagram.

Sol:

This is one of the oldest and most popular line clipping procedures. Generally, the method **speeds** up the processiug of line segments performing initial tests that reduce the number of intersections that must he calculated. Everv line endpoint 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 referehce to the boundaries as



shown in the Fig 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. The codes are assignened as

bit **1:** left - is set to **1** if **x** < **xmin,**

bit 2: right- is set to **1** if **x** > **xmax**

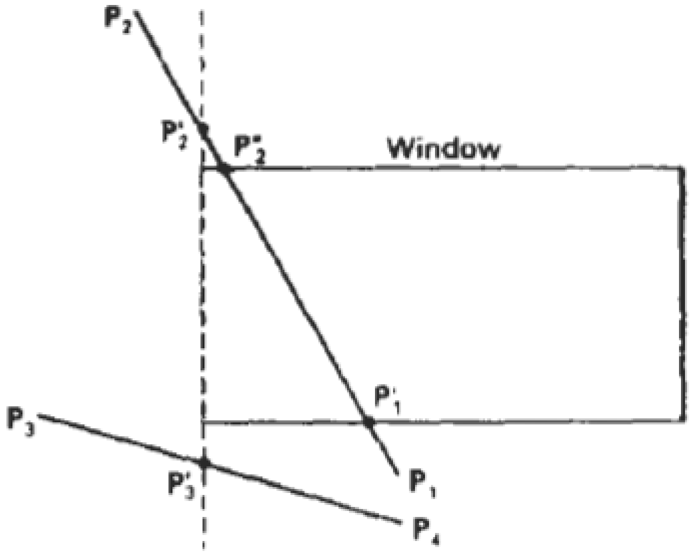
bit 3: below -- is set to **1** if **y** < **ymin,**

bit 4: above - is set to **1** if **y**< **ymax,**

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.**

Bit values in the region code are determined by comparing endpoint coordinate values **(x, y)** to the clip boundaries. Once we have established region codes we use the following rules:
1. Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially accept these lines.
2. 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 trivially reject these lines. We could do this using logical and operation with both region codes. If the result is not **0000,** the line is completely outside the clipping region.
3. Lines that cannot be identified as completely inside or completely outside a clip window by these tests are checked for intersection with the window boundaries.

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.
For example in the example below, Starting with the bottom endpoint of the line from **P1** to **P2.** we check P, 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 P1' with the bottom boundary and discard the line section from PI to **P1'.** The line now has been reduced to the section from **P1'** to **P2.** Since **P,** 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;** is calculated, but this point is above the window. So the final intersection calculation yields **P2',** and the line from P1' to **P2''** is saved.



The algorithm for the algorithm is:

```
Algorithm encode(p,xmin,xmax,ymin,ymax)
{
        code=0000
        if p.x<xmin
                leftbit(code)=1
        if p.x>xmin
                rightbit(code)=1
        if p.y<ymin
                bottombit(code)=1
        if p.y>ymax
                topbit(code)=1
}

// p1 and p2 are the endpoints of the line segment to be clipped
Algorithm CohenSutherland(p1,p2,xmin, xmax, ymin, ymax)
{
        drawflag=true
        c1=encode(p1, xmin, xmax, ymin, ymax)
        c2=encode(p2, xmin, xmax, ymin, ymax)
        if (c1 & c2 != 0) // line is completely outside
                drawflag=false
        else if (c1!=0 || c2!= 0) // atleast one endpoint is inside the window
        {
                m=(p2.y-p1.y)/(p2.x-p1.x)

                // finding intersection with left edge if needed
                // make sure line 1 is outside the boundary
                if  leftbit(c2)==1 //line 2 is outside the left boundary
                {
                        swap(p1,p2)
                        swap(c1,c2)
                }

                if (leftbit(c1) == 1) // one of the points is outside the boundary
                {
                        p1.y  = $p1.y + m.(xmin - p1.x)$ // find point of intersection withleft
edge
                        p1.x=xmin;
                }


                // finding intersection with right edge if needed
                // make sure line 1 is outside the boundary
                if  rightbit(c2)==1 //line 2 is outside the left boundary
                {
                        swap(p1,p2)
                        swap(c1,c2)
                }

                if (rightbit(c1) == 1) // one of the points is outside the boundary
                {
                        p1.y  = $p1.y + m.(xmax - p1.x)$ // find point of intersection withleft
edge
                        p1.x=xmax;
                }

                // finding intersection with bottom edge if needed
                // make sure line 1 is outside the boundary
                if  bottombit(c2)==1 //line 2 is outside the left boundary
                {
                        swap(p1,p2)
                        swap(c1,c2)
                }

                if (bottombit(c1) == 1) // one of the points is outside the boundary
                {
                        p1.x  = $p1.x + (ymin - p1.y).\frac{1}{m}$ // find point of intersection withleft edge
                        p1.y=ymin;
                }

                // finding intersection with top edge if needed
                // make sure line 1 is outside the boundary
                if  topbit(c2)==1 //line 2 is outside the left boundary
                {
                        swap(p1,p2)
                        swap(c1,c2)
                }

                if (topbit(c1) == 1) // one of the points is outside the boundary
                {
                        p1.x  = $p1.x + (ymax - p1.y).\frac{1}{m}$ // find point of intersection withleft edge
                        p1.y=ymax;
                }
        }
}
```

(c) Briefly explain text clipping.

Sol: There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application. The simplest method for processing character strings relative to a window boundary is to use the **all-or-none string-clipping** strategy shown in Fig. 6-28. If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern. The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping. An alternative to rejecting an entire character string that overlaps a window boundary is to use the all-or-none **character-clipping** strategy. Here we discard only those characters that **are** not completely inside the window. In this case, the boundary limits of individual characters are compared to the window. **Any** character that either overlaps or is outside a window boundary is clipped. **A** final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines. If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window. Outline character fonts formed with line segments can be processed in this way using a line clipping algorithm. Characters defined with bit maps would be clipped by comparing the relative position of the individual pixels in the character grid patterns to the clipping boundaries.

Q6. (a) How are modelling coordinates converted to viewing coordinates in 3D pipeline?
Sol:
We first select world coordinate position P0=(x0,y0,z0) for the viewing origin, which is called the view point or the viewing origin. We also specify the viewup vector V, which defines the yview direction. For 3D space we also need to asign a direction
The viewing direction is usually along the zview axis, the view plane is also called the projection place is assumed to be perpendicular to this axis. The orientation of the positive zview axis can be defined usign a view-plane normal vector N.
An additional scalar parameter is used to set the position of the view plane of some coordinate value z_{vp} along the z_{view} axis. We take N to be in the direction from a reference point (look at point) Pref to the viewing origin P0. In this case the viewing direction is opposite that of N. Once a view plane normal N is chosen we set up the view up vector V which is used to establish the positive direction for the yview axis. V is expected to be perpendicular to N but it may not be the case sometimes and so V is projected onto a plane that is perpendicular to the view plane nromal vector. Since N defines the zview and theV is used to derivve yview we can compute a third vector U which is perpendicular to both V and N by taking the cross product of the two vectors V and N . Vecto U then defines the positive xview direction. The cross product of N and U produces the adjusted value of V perpendicular to both N and u aong the positive yview direction. Thus we obtain the set of unit axis viewing vectors using the following computations.

$$n = \frac{N}{|N|}(n_x, n_y, n_z)$$

$$u = \frac{V \times n}{|V|} = (u_x, u_y, u_z)$$

$$v = n \times u = (v_x, v_y, v_z)$$

The coordinate system thus formed id called the uvn coordinate reference framework.
After the establishment of the uvn framework twe can transform from world to viewing coordinates by the steps:
1. Translate the viewing coordinate origin to the origin of the world coordinate system
2. Apply rotations to alig the xview, yview and zview axis with the world axes.
Since the viewing coordinate is at P0=(x0,y0,z0) the translation matrix would be

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For the rotatio transformation we can use the unit vectors u,v,n to form the composite rotation matrix that superimposes the viewing axes onto the world frame. the rotation matris would be :

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The coordinate transformation matrix is then obtained as the product of the two matrices

$$M_{WC,VC} = R \cdot T = \begin{bmatrix} u_x & u_y & u_z & -u \bullet P_0 \\ v_x & v_y & v_z & -v \bullet P_0 \\ n_x & n_y & n_z & -n \bullet P_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

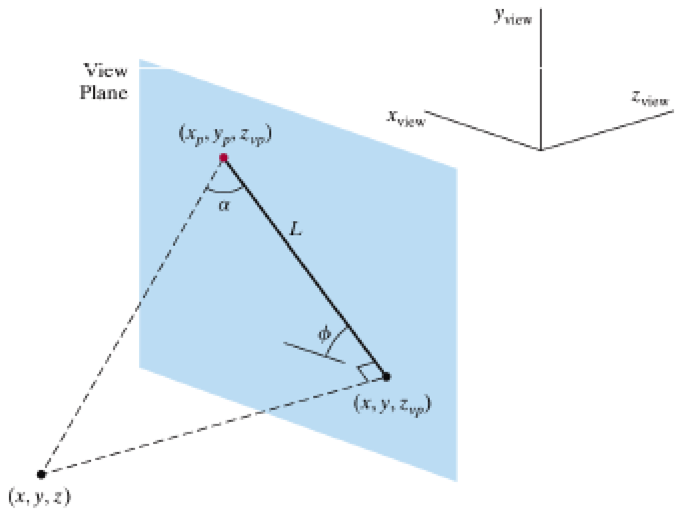
where

$$-u \bullet P_0 = -x_0 u_x - y_0 u_y - z_0 u_z$$
$$-v \bullet P_0 = -x_0 v_x - y_0 v_y - z_0 v_z$$
$$-n \bullet P_0 = -x_0 n_x - y_0 n_y - z_0 n_z$$

(b) Explain oblique parallel projection.

Sol:

In parallel projection the projections paths are all parallel to each other. Oblique parallel projections have projectors which are not perpendicular to the projection plane. For applications in engineering and architectural design, an oblique parallel projection are specified using two angles α and phi as shown the figure below:



Imagine a spatial position (x, y, z) is projected to (xp, yp, z_{vp}). (x, y, z_{vp}) is the orthogonal-projection point. The oblique projection line from (x,y,z) to (xp, yp, z_{vp}) has an intersection angle α with the line on the projection pane that joins (xp, yp, z_{vp}) and (x, y, z_{vp}). This view plane line with length L, is at an angle phi, with the horizontal direction in the projection plane. We can express the projection coordinates in terms of xy,L and phi as:

$$x_p = x + L\cos\phi$$
$$y_p = y + L\sin\phi$$

Length L depends on the angle α and the perpendicular distance of the point (x,y,z) from the view plane

$$\tan\alpha = \frac{z_{up} - z}{L}$$

$$L = L_1(z_{up} - Z)$$

$$L_1 = \cot\alpha$$

We can write the oblique parallel projection equation as

$$x_p = x + L_1(z_{up} - z)\cos\phi$$

$$y_p = y + L_1(z_{up} - z)\sin\phi$$

The above equations represent a Z axis shearing transformation. In graphics libraries however, instead of specifying α and phi the direction of the projection is specified with parallel projection vector Vp. Once the projection vector is established in viewing coordinates, all points on the scene are transferred to the view plane along lines that are parallel to this vector. We can denote the components of the projection vector e==relative to the viewing coordinate frame as Vp=(Vpx,Vpy,Vpz) where Vpy/Vpx=tan(phi). Comparing similar triangles

$$\frac{x_p - x}{z_{vp} - z} = \frac{V_{px}}{V_{pz}},$$

$$\frac{y_p - y}{z_{vp} - z} = \frac{V_{py}}{V_{pz}}$$

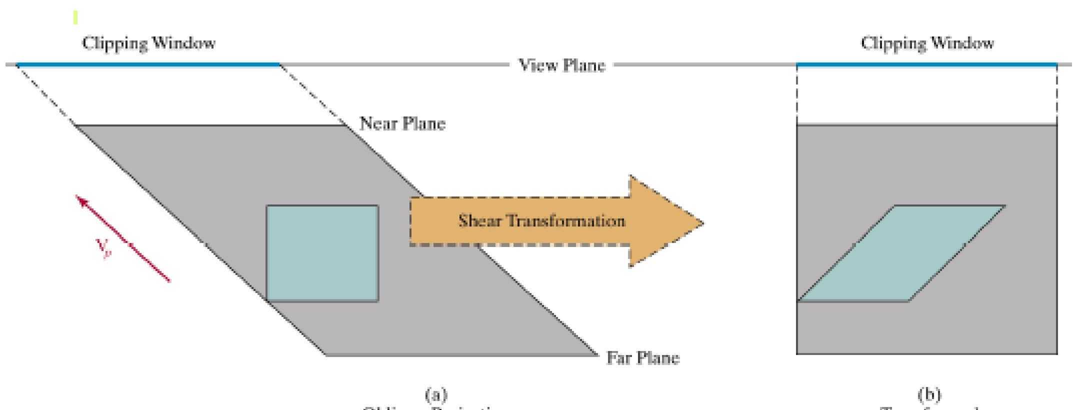
and therefore

$$x_p = x + (z_{vp} - z)\frac{V_{px}}{V_{pz}}, \qquad and\ y_p = y + (z_{vp} - z)\frac{V_{py}}{V_{pz}}$$

The projection matrix therefore is :

$$M_{oblique} = \begin{bmatrix} 1 & 0 & -\frac{V_{px}}{V_{pz}} & z_{up}\frac{V_{px}}{V_{pz}} \\ 0 & 1 & -\frac{V_{py}}{V_{pz}} & z_{up}\frac{V_{py}}{V_{pz}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

to convert the view volume



(a)
Oblique-Projection
View Volume

(b)
Transformed
Oblique
View Volume

Q7. (a) Explain in detail Bezier Spline curve
Sol:
It is a spline approximation method was developed by the French engineer Pierre Bezier for use in the design of Renault automobile bodies. **Bezier** splines have a number of properties that make them highly useful and convenient for curve and surface design. They are also easy to implement.
In general, a Bezier curve section can be fitted to any number of control points. The number of control points to be approximated and their relative position determine the degree of the Bezier polynomial.
Suppose we are given ***n+1*** control-point positions: **pk** = (xk, ***yk,*** zk), with ***k*** varying from **0** to n. These coordinate points can **be** blended to produce the following position vector ***P(u),*** which describes the path of an approximating Bezier
polynomial function between **p0** and p1.

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k BEZ_{k,n}(u), \qquad 0 \le u \le 1$$

The Bezier blending functions ***$BEZ_{k,n}$(u) are defined as:***

$$BEZ_{k,n}(u) = C(n, k)u^k(1 - u)^{n-k}$$

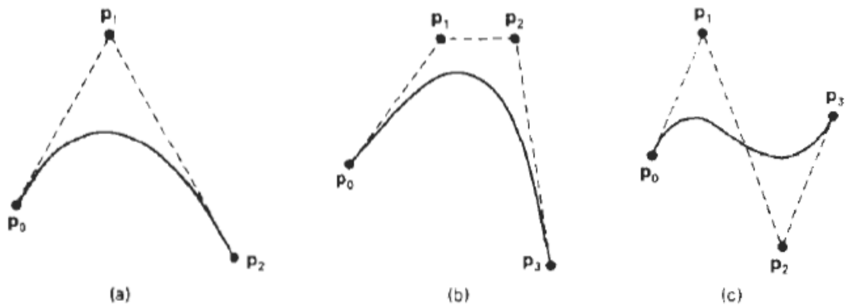
where the ***C(n, k)*** are the binomial coefficients. The vector equation of the curve given above can also be written as three parametric equations for the individual curve coordinates:

$$x(u) = \sum_{k=0}^{n} x_k\, BEZ_{k,n}(u)$$

$$y(u) = \sum_{k=0}^{n} y_k\, BEZ_{k,n}(u)$$

$$z(u) = \sum_{k=0}^{n} z_k\, BEZ_{k,n}(u)$$

a Bezier curve is a polynomial of degree one less than the number of control points used.
The figure below demonstrates the appearance of some Bezier curves.



Bezier curves are commonly found in painting and drawing packages, as well as CAD systems. Efficient methods for determining coordinate positions along a Bezier curve can be set up using recursive calculations. For example, successive binomial coefficients can be calculated as:

$$C(n,k) = \frac{n-k+1}{k} C(n,k-1)$$

(b) Describe basic approach to design animation sequence.
Sol:
In general, an animation sequence is designed with the following steps:

- Storyboard layout: The ***storyboard*** 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.
- An ***object definition*** is given for each participant in the action. Objects can be defined in terms of basic shapes, such as polygons or spines. In addition, the associated movements for each object **are** specified along with the shape.
- **A key frame** 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. Some key frames are chosen at extreme positions in the action; others are spaced **so** that the time interval between key frames is not *too* great. More key frames are specified for intricate motions than for simple, slowly varying motions.
- Generation of in-between frames: In-betweens 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. Typically, time intervals for the motion are **set** up so that there ***are*** from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames can **be** duplicated. For a I-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. If the motion is not too complicated, we could space the key frames a little farther apart.

Apart from the above steps the other steps that may need to be done are: motion verification, editing, and production and synchronization of a soundtrack.

(c) Differentiate traditional animation and computer animation techniques.
Sol

Traditional Computer Animation Technique

Traditional Animation TEchniques: The traditional animation or also known as the hand-drawn animation has been used in most animated films of the 20th century. The individual frames of a traditionally animated film are photographs of drawings, which are first drawn on paper.
Traditional animation is one of the most popular as well as the oldest technique of animation. It is also known by other popular names such as classical animation, cel animation, or hand-drawn animation. The traditional animation technique works in a manner that each of the frame in a cartoon made by using this technique is made by hand as contrasted to computer animation where the frames are generated by computer.
In order to produce a false impression of movement, each piece of drawing would be little bit different from the one preceding it. The different drawings are traced or photocopied onto transparent acetate sheets which are known as cells, and are filled in with paints in designated colors or shades on the side opposite the line drawings. The finished character cels are then photographed one-by-one onto motion picture film against a painted background with the help of a rostrum camera.

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. Some Animation packages, such as Wave front, , 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.

Q8. Write short notes on:
(a) Breshenham's line drawing algorithm
This algorithm is an efficient raster line-generating algorithm, developed by Bresenham, that uses only incremental integer calculations.
Advantages of Breshenham
1. Only integer calculations
2. can be adapted for circle and ellipse drawing too

Algorithm for Breshenham's Line Drawing
1. Input the to line endpoints and store the left endpoint in (x0,y0)
2. Set the color for frame-buffer position(x0,y0) ,i.e. plot the first point
3. If the |m| <=1 then calculate the constants Δx, Δy, $2\Delta y$ and either $2\Delta y-2\Delta x$ if slope is positive or $-2\Delta y-2\Delta x$ if slope is negative and obtain the starting value for the decision parameter as
$p_0 = 2\Delta y - \Delta x$ if slope is positive and $p_0 = -2\Delta y - \Delta x$ if slope is negative
else go to step 6 for |m|>1
4. If the slope is positive then at each point x_k along the line starting at k=0, perform the following test.
If pk <0, the next point to plot is (x_k+1,y_k) and

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

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

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

If the slope is negative then at each point x_k along the line starting at k=0, perform the following test.

If pk <0, the next point to plot is (x_k+1,y_k) and

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

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

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

5. Perform step 4 Δx-1 times and exit

6. Calculate the constants Δx, Δy, 2Δx and either 2Δx-2Δy if slope is positive or -2Δx-2Δy if slope is negative and obtain the starting value for the decision parameter as $p_0 = 2\Delta x - \Delta y$ if slope is posiitve and $p_0 = -2\Delta x - \Delta y$ if slope is negative.

7. If the slope is positive then at each point y_k along the line starting at k=0, perform the following test.

If p_k <0, the next point to plot is (x_k,y_k+1) and

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

Otherwise, the next point to plot is (x_k+1,y_k+1) and

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

If the slope is negative then at each point x_k along the line starting at k=0, perform the following test.

If p_k <0, the next point to plot is (x_k,y_k+1) and

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

Otherwise, the next point to plot is (x_k,y_k+1) and

$p_{k+1} = p_k - 2\Delta x$ -22Δy

8. Perform step 7 Δy-1 times and exit

(b) Affine transformations:

Three-dimensional geometric transformations such as translation, rotation, scaling, reflection and shearing are affine transformations. That is, they can be expressed as a linear function of coordinates **x** y and z. Affine transformations transform parallel lines to parallel lines and transform finite points to finite points. Geometric transformations that do not involve scaling or
shear also preserve angles and lengths. The general affine transformation can be expressed as

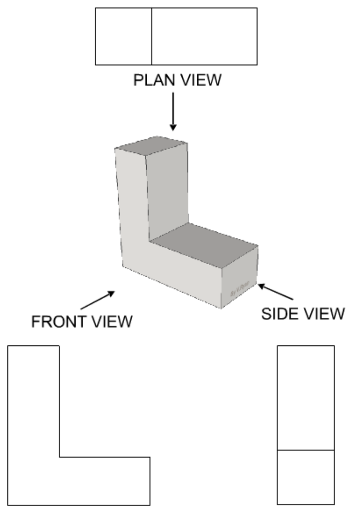
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

(c) Depth cueing:

Depth information is important ***so*** that we can easily identify, for a particular viewing direction, which is the front and which is the back of displayed objects. There are several ways in which we can include depth information in the two-dimensional representation of solid objects. A simple method for indicating depth with wireframe displays is to vary the intensity of objects according to their distance from the viewing position. The lines closest to away are displayed with decreasing intensities. Depth cueing is applied by choosing maximum and minimum intensity (or color) values and a range of distances over which the intensities are to vary. Another application of depth cueing is modelling the effect of the atmosphere on the perceived intensity of objects. More distant objects appear dimmer to us than nearer objects due to light scattering by dust particles, haze, and smoke. Some atmospheric effects can change the perceived color of an object, and we can model these effects with depth cueing.

(d) Orthogonal projection:

A transformation of object descriptions to a view plane along lines that are all parallel to the view-plane normal vector N is called an orthogonal projection. This produces a parallel-projection transformation in which the projection lines are perpendicular to the view plane. Orthogonal projections are most often used to produce the front, side, and top views of an object. Front, side, and rear orthogonal projections of an object are called elevations; and a top orthogonal projection is called a plan view. Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings


PLAN VIEW
FRONT VIEW
SIDE VIEW