Computer Graphics Second Internal Scheme & Solutions

SCHEME

1. Write a program for Cohen-Sutherland Line Clipping.
    a. Program : 10 Marks
2. Explain the following 2D geometric transformation with matrix derivation.
    a. Translation
        i. Explanation : 2 Marks
        ii. Matrix Derivation : 1 Marks
    b. Rotation
        i. Explanation : 3 Marks
        ii. Matrix Derivation : 1 Marks
    c. Scaling.
        i. Explanation : 2 Marks
        ii. Matrix Derivation : 1 Marks
3. What is a display list? How pop-up menus are created using GLUT, Illustrate with an example.
    a. Display List : 2 Marks
    b. Menu : 4 Marks
    c. Example : 4 Marks
4. Explain how an event driven input can be programmed using a keyboard and mouse devices.
    a. Keyboard Event : 5 Marks
    b. Mouse Event :5 Marks
5. Explain different input modes with neat diagram.
    a. Request Mode : 3 Marks
    b. Event Mode : 4 Marks
    c. Sample Mode : 3 Marks
6. With code snippet explain picking concept.
    a. Explanation : 4 Marks
    b. Code Snippet : 6 Marks
7. Explain mapping of world coordinates to a viewport with appropriate diagram.
    a. Explanation : 4 Marks
    b. Diagram : 2 Marks
    c. Matrix : 4 Marks
8. Briefly explain 2D viewing pipeline with a neat diagram.
    a. Explanation : 6 Marks

   b. Diagram : 4 Marks

9. Clip the below polygon using Sutherland Hodgeman polygon clipping algorithm.
   a. Each Step carries 2.5 Mark

10. Explain different types of logical input devices.
    a. Each device explanation carries 2 Marks

SOLUTIONS

1. Write a program for Cohen-Sutherland Line Clipping.

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

// outcode is same as int
#define outcode int

// window boundary
double xmin=50,ymin=50,xmax=100,ymax=100;

// viewport boundary
double xvmin=200,yvmin=200,xvmax=300,yvmax=300;

// bitcode for the right,left,top & bottom
const int RIGHT=8;  // 8 is 1000
const int LEFT=2;   // 2 is 0010
const int TOP=4;    // 4 is 0100
const int BOTTOM=1; // 1 is 0001
int x1,x2,y1,y2;

// used to compute bitcode of a point
outcode ComputeOutcode(double x,double y);

// Cohen-Sutherland clipping algo clips a line from
// P0=(x0,y) to P1=(x1,y1) against a rectangle with
// diagonal from (xmin,ymin) to (xmax,ymax)
void CohenSutherlandLineClipAndDraw(double x0,double y0,double x1,double y1)
{
    outcode outcode0,outcode1,outcodeOut;
    bool accept=false,done=false;

    // compute outcodes
    outcode0=ComputeOutcode(x0,y0);
    outcode1=ComputeOutcode(x1,y1);
    do
    {
        // logical or is 0 Trivially accept & exit
        if(!(outcode0|outcode1))
        {
            accept=true;
            done=true;
        }

        // logical and is not 0.Trivially reject & exit
        else if(outcode0 & outcode1)
        {
            done=true;
        }
        else
        {
            // failed both tests,so calculate the line segment to clip
            // from an outside point to an intersection with clip edge
            double x,y;

            //Atleast one endpoint is outside the clip rectangle,pick it.
            outcodeOut=outcode0?outcode0:outcode1;

            //Now find the intersection point
            //use formula y=y0+slope*(x-x0), x=x0+(1/slope)*(y-y0)
            //point is above the rectangular clip
            if(outcodeOut & TOP)
            {
                x=x0+(x1-x0)*(ymax-y0)/(y1-y0);
                y=ymax;
            }

            // point is below the clip rectangle
            else if(outcodeOut & BOTTOM)
            {
                x=x0+(x1-x0)*(ymin-y0)/(y1-y0);
                y=ymin;
            }

            // point lies right of clipping rectangle
            else if(outcodeOut & RIGHT)
            {
                y=y0+(y1-y0)*(xmax-x0)/(x1-x0);
                x=xmax;
            }

            // point lies leftside of clipping rectangle
            else
            {
                y=y0+(y1-y0)*(xmin-x0)/(x1-x0);
                x=xmin;
            }

            // now we move outside point to intersection point to
            // clip and get ready for next pass
            if(outcodeOut==outcode0)
            {
                x0=x;
                y0=y;
                outcode0=ComputeOutcode(x0,y0);
            }
            else
            {
                x1=x;
                y1=y;
                outcode1=ComputeOutcode(x1,y1);
            }
        }
    }
    while(!done);
    if(accept)
    {
        // Window to viewport mappings

        // scaling parameters
        double sx=(xvmax-xvmin)/(xmax-xmin);
        double sy=(yvmax-yvmin)/(ymax-ymin);

        double vx0=xvmin+(x0-xmin)*sx;
        double vy0=yvmin+(y0-ymin)*sy;
        double vx1=xvmin+(x1-xmin)*sx;
        double vy1=yvmin+(y1-ymin)*sy;

        // draw a red coloured viewport
        glColor3f(1.0,0.0,0.0);
        glBegin(GL_LINE_LOOP);
        glVertex2f(xvmin,yvmin);
        glVertex2f(xvmax,yvmin);
        glVertex2f(xvmax,yvmax);
        glVertex2f(xvmin,yvmax);
        glEnd();

        // draws blue colour viewport
        glColor3f(0.0,0.0,1.0);
        glBegin(GL_LINES);
        glVertex2d(vx0,vy0);
        glVertex2d(vx1,vy1);
        glEnd();
    }
}

// compute the bitcode for a point(x,y) using the clip rectangle
// bounded diagonally by (xmin,ymin) and (xmax,ymax)
outcode ComputeOutcode(double x,double y)
{
    outcode code=0;
    if(y>ymax)
    {
        code|=TOP;
    }
    else if(y<ymin)
    {
        code|=BOTTOM;
    }
    if(x>xmax)
    {
        code|=RIGHT;
    }
    else if(x<xmin)
    {
        code|=LEFT;
    }
    return code;
}

void display()
{
    // double x0=60,y0=20,x1=80,y1=120;
    glClear(GL_COLOR_BUFFER_BIT);

    // draw the line with red colour
    glColor3f(1.0,0.0,0.0);

    glBegin(GL_LINES);

    <!--ToDo Why are these two lines commented?-->
    //glVertex2d(x0,y0);
    //glVertex2d(x1,y1);
    glVertex2d(x1,y1);
    glVertex2d(x2,y2);
    glEnd();

    // draw blue coloured window
    glColor3f(0.0,0.0,1.0);
    glBegin(GL_LINE_LOOP);
    glVertex2f(xmin,ymin);
    glVertex2f(xmax,ymin);
    glVertex2f(xmax,ymax);
    glVertex2f(xmin,ymax);
    glEnd();
    CohenSutherlandLineClipAndDraw(x1,y1,x2,y2);
    glFlush();
}

void myinit()
{
    glClearColor(1.0,1.0,1.0,1.0);
    glColor3f(1.0,0.0,0.0);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,500.0,0.0,500.0);
}


void main(int argc,char **argv)
{
    printf("Enter End Points:(x0,x1,y0,y1)");
    scanf("%d%d%d%d",&x1,&x2,&y1,&y2);
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
    glutCreateWindow("Cohen Sutherland Line Clipping Algorithm");
    glutDisplayFunc(display);
    myinit();
    glutMainLoop();
}
```

2. Explain the following 2D geometric transformation with matrix derivation.

a. Translation.

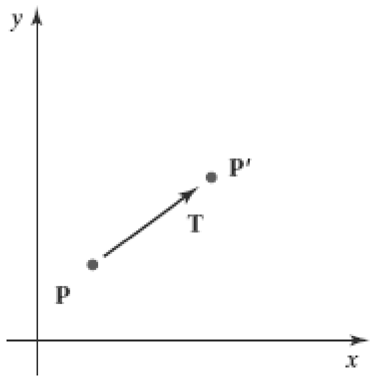
b. Rotation.

c. Scaling.

Solution:

a. Translation

- We perform a translation on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position.
- We are moving the original point position along a straight-line path to its new location.
- To translate a two-dimensional position, we add translation distances tx and ty to the original coordinates (x, y) to obtain the new coordinate position (x', y') as shown in Figure.



The translation values of x' and y' is calculated as

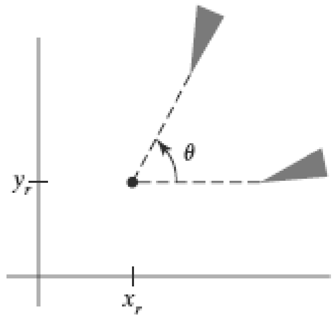
$$x' = x + t_x, \qquad y' = y + t_y$$

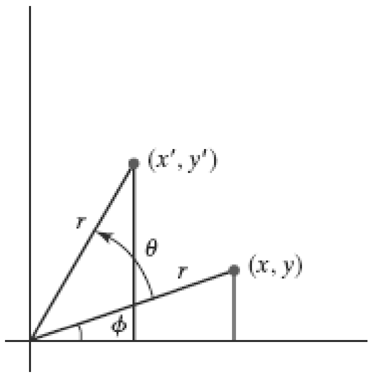
$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

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

b. Rotation

- We generate a rotation transformation of an object by specifying a rotation axis and a rotation angle.
- A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the xy plane.
- In this case, we are rotating the object about a rotation axis that is perpendicular to the x y plane (parallel to the coordinate z axis).
- Parameters for the two-dimensional rotation are the rotation angle θ and a position (xr, yr), called the rotation point (or pivot point), about which the object is to be rotated.





- 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.
- we can express the transformed coordinates in terms of angles θ and φ as,

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

$$x = r\cos\phi, \qquad y = r\sin\phi$$

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

Rotation Matrix is,

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

c. Scaling

- To alter the size of an object, we apply a scaling transformation.
- A simple two dimensional scaling operation is performed by multiplying object positions (x, y) by scaling factors sx and sy to produce the transformed coordinates (x', y'):

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y$$

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

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

3. What is a display list? How pop-up menus are created using GLUT, Illustrate with an example.

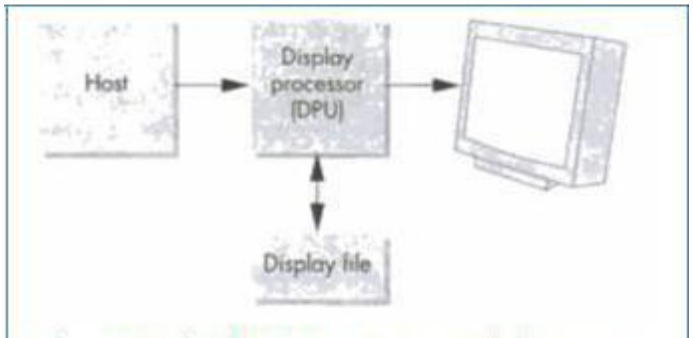
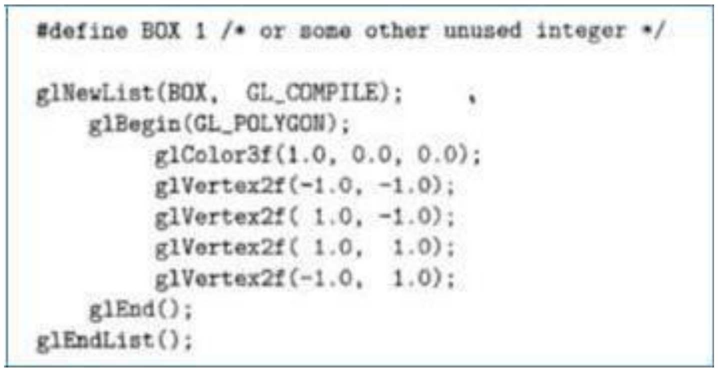


Figure: Display processor architecture

The user program is processed by the host computer which results a compiled list of instruction that was then sent to the display processor, where the instruction are stored in a display memory called as “display file” or “display list”. Display processor executes its display list contents repeatedly at a sufficient high rate to produce flicker-free image.

Display lists are defined similarly to the geometric primitives. i.e., glNewList() at the beginning and glEndList() at the end is used to define a display list.

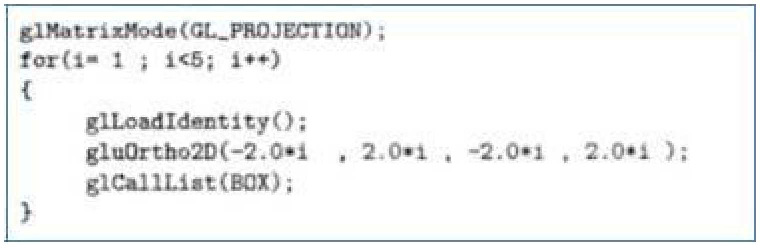
- Each display list must have a unique identifier – an integer that is usually a macro defined in the C program by means of #define directive to an appropriate name for the object in the list. For example, the following code defines red box:



```
#define BOX 1 /* or some other unused integer */

glNewList(BOX,  GL_COMPILE);
    glBegin(GL_POLYGON);
         glColor3f(1.0, 0.0, 0.0);
         glVertex2f(-1.0, -1.0);
         glVertex2f( 1.0, -1.0);
         glVertex2f( 1.0,  1.0);
         glVertex2f(-1.0,  1.0);
    glEnd();
glEndList();
```

- The flag GL_COMPILE indicates the system to send the list to the server but not to display its contents. If we want an immediate display of the contents while the list is being constructed then GL_COMPILE_AND_EXECUTE flag is set.
- Each time if the client wishes to redraw the box on the display, it need not resend the entire description. Rather, it can call the following function:
    - glCallList(Box)
- The Box can be made to appear at different places on the monitor by changing the projection matrix as shown below:



```
glMatrixMode(GL_PROJECTION);
for(i= 1 ; i<5; i++)
{
      glLoadIdentity();
      gluOrtho2D(-2.0*i  , 2.0*i , -2.0*i , 2.0*i );
      glCallList(BOX);
}
```

4. Explain how an event driven input can be programmed using a keyboard and mouse devices.

KEYBOARD EVENTS

- Keyboard devices are input devices which return the ASCII value to the user program. Keyboard events are generated when the mouse is in the window and one of the keys is pressed or released.
- GLUT supports following two functions
    - glutKeyboardFunc() is the callback for events generated by pressing a key
    - glutKeyboardUpFunc() is the callback for events generated by releasing a key.
    - The information returned to the program includes ASCII value of the key pressed and the position (x,y) of the cursor when the key was pressed.

Programming keyboard event involves two steps:

- The keyboard callback function must be defined in the form:
- **void mykey (unsigned char key, int x, int y)** is written by the application programmer. For example,

```
void mykey(unsigned char key, int x, int y)
{
if(key== 'q' || key== 'Q')
exit(0);
}
```

The above code ensures when 'Q' or 'q' key is pressed, the execution of the program gets terminated.

- The keyboard callback function must be registered in the main function by means of GLUT function:
    - **glutKeyboardFunc(mykey);**

MOUSE EVENT

- Is generated when one of the mouse buttons is either pressed or released.

- The information returned to the application program includes button that generated the event, state of the button after event (up or down), position (x,y) of the cursor.
- Programming a mouse event involves two steps:
  - The mouse callback function must be defined in the form: **void myMouse(int button, int state, int x, int y)** is written by the programmer.

    For example,

    ```
    void myMouse(int button, int state, int x, int y)
    {
    if(button==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    exit(0);
    }
    ```

- The above code ensures whenever the left mouse button is pressed down, execution of the program gets terminated.
- Register the defined mouse callback function in the main function, by means of GLUT function:
  - **glutMouseFunc(myMouse);**

5. Explain different input modes with neat diagram.

REQUEST MODE: In this mode, measure of the device is not returned to the program until the device is triggered.

- For example, consider a typical C program which reads a character input using scanf(). When the program needs the input, it halts when it encounters the scanf() statement and waits while user type characters at the terminal. The data is placed in a keyboard buffer (measure) whose contents are returned to the program only after enter key (trigger) is pressed.
- Another example, consider a logical device such as locator, we can move out pointing device to the desired location and then trigger the device with its button, the trigger will cause the location to be returned to the application program.

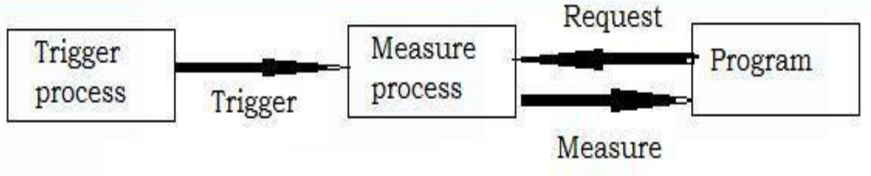


Figure: Request Mode

SAMPLE MODE: In this mode, input is immediate. As soon as the function call in the user program is executed, the measure is returned. Hence no trigger is needed.

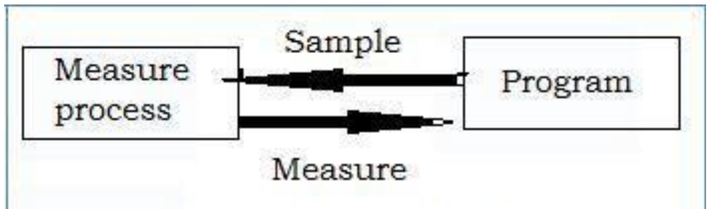


Figure: Sample Mode

Both request and sample modes are useful for the situation if and only if there is a single input device from which the input is to be taken. However, in case of flight simulators or computer games variety of input devices are used and these mode cannot be used. Thus, event mode is used.

EVENT MODE:

This mode can handle the multiple interactions.

- Suppose that we are in an environment with multiple input devices, each with its own trigger and each running a measure process.
- Whenever a device is triggered, an event is generated. The device measure including the identifier for the device is placed in an event queue.
- If the queue is empty, then the application program will wait until an event occurs. If there is an event in a queue, the program can look at the first event type and then decide what to do.

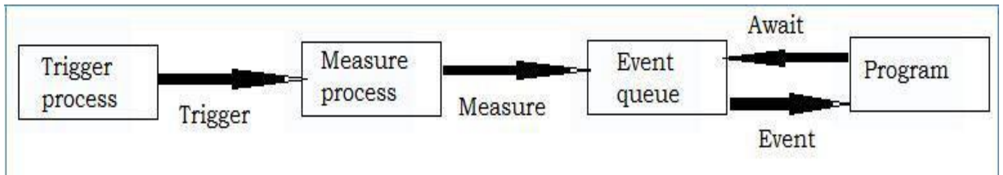
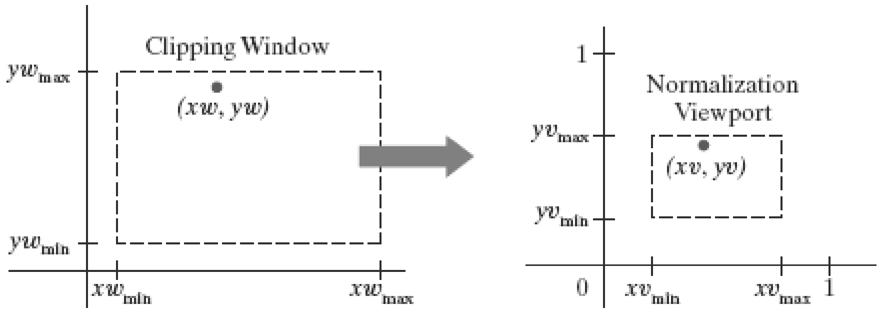


Figure: Event-mode model

6. Explain mapping of world coordinates to a viewport with appropriate diagram.

We first consider a viewport defined with normalized coordinate values between 0 and 1.

- Object descriptions are transferred to this normalized space using a transformation that maintains the same relative placement of a point in the viewport as it had in the clipping
- window Position (xw, yw) in the clipping window is mapped to position (xv, yv) in the associated viewport.



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

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

Solving these expressions for the viewport position (xv, yv), we have

xv = sxxw + tx

yv = syyw + ty

Where,

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

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

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

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

We could obtain the transformation from world coordinates to viewport coordinates with the following sequence:

- Scale the clipping window to the size of the viewport using a fixed-point position of (xwmin, ywmin).
- Translate (xwmin, ywmin) to (xvmin, yvmin).

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{min}(1 - s_x) \\ 0 & s_y & yw_{min}(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

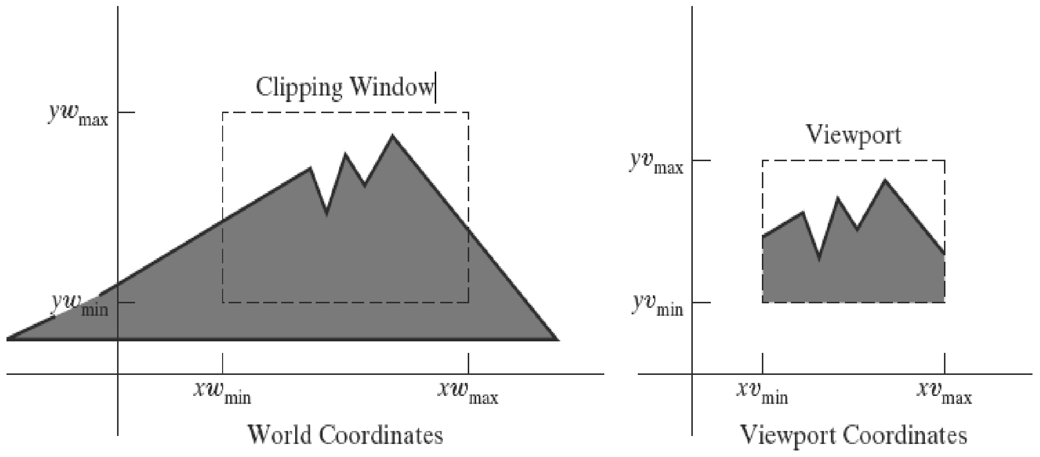
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{min} - xw_{min} \\ 0 & 1 & yv_{min} - yw_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T}\cdot\mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

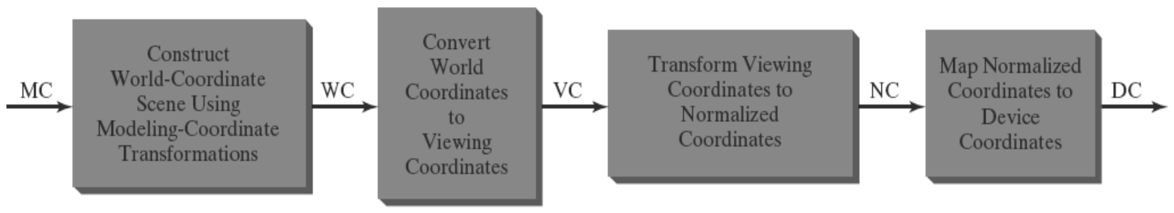
7. Briefly explain 2D viewing pipeline with a neat diagram.

A section of a two-dimensional scene that is selected for display is called a clipping Window.

- Sometimes the clipping window is alluded to as the world window or the viewing window
- Graphics packages allow us also to control the placement within the display window using another "window" called the viewport.
- The clipping window selects what we want to see; the viewport indicates where it is to be viewed on the output device.
- By changing the position of a viewport, we can view objects at different positions on the display area of an output device
- Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.
- We first consider only rectangular viewports and clipping windows, as illustrated in Figure.



Viewing Pipeline


MC
Construct World-Coordinate Scene Using Modeling-Coordinate Transformations
WC
Convert World Coordinates to Viewing Coordinates
VC
Transform Viewing Coordinates to Normalized Coordinates
NC
Map Normalized Coordinates to Device Coordinates
DC

- The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a two-dimensional viewing transformation.

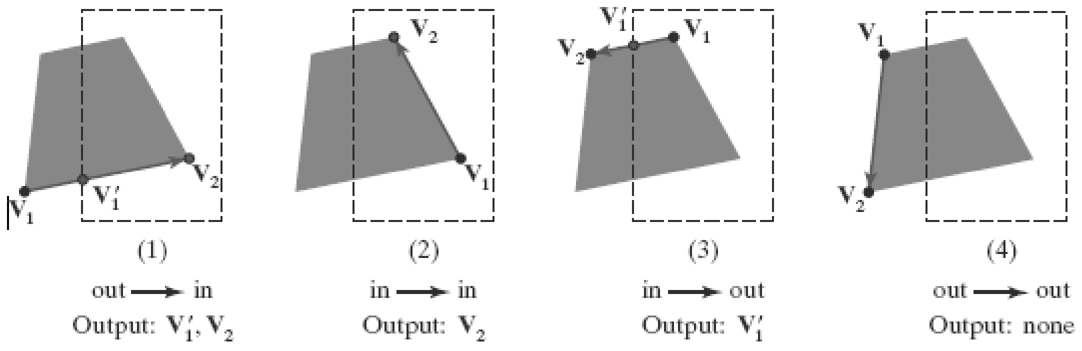
- This transformation is simply referred to as the window-to-viewport transformation or the windowing transformation
- We can describe the steps for two-dimensional viewing as indicated in Figure Once a world-coordinate scene has been constructed, we could set up a separate two dimensional, viewing coordinate reference frame for specifying the clipping window.
- To make the viewing process independent of the requirements of any output device, graphics systems convert object descriptions to normalized coordinates and apply the clipping routines.
- Systems use normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1.
- At the final step of the viewing transformation, the contents of the viewport are transferred to positions within the display window.
- Clipping is usually performed in normalized coordinates.
- This allows us to reduce computations by first concatenating the various transformation matrices.

8. Clip the below polygon using Sutherland Hodgeman polygon clipping algorithm.

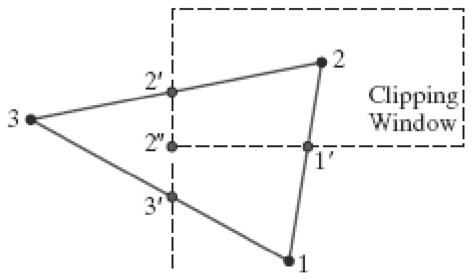
There are four possible cases that need to be considered when processing a polygon edge against one of the clipping boundaries.

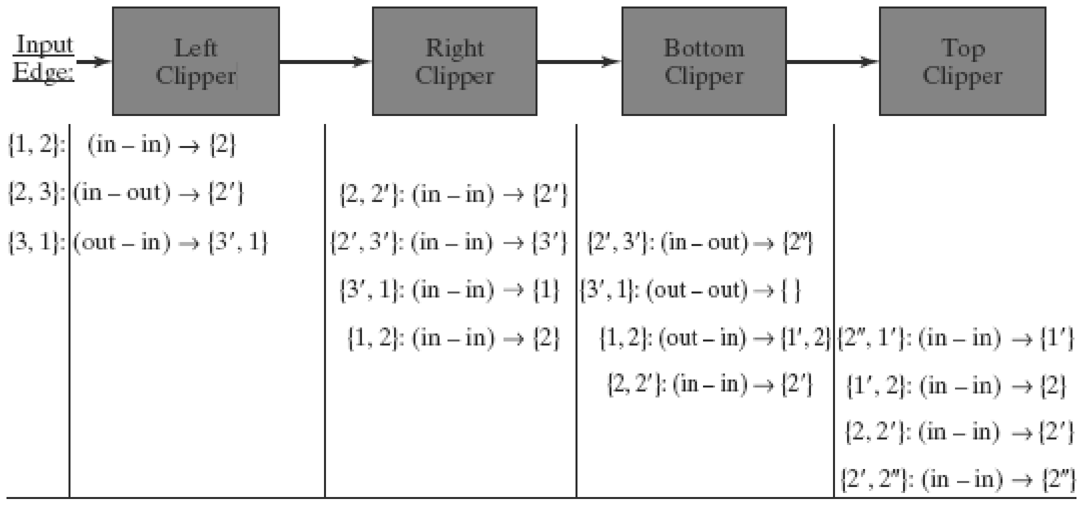
- ○ One possibility is that the first edge endpoint is outside the clipping boundary and the second endpoint is inside.
- ○ Or, both endpoints could be inside this clipping boundary.
- ○ Another possibility is that the first endpoint is inside the clipping boundary and the second endpoint is outside.
- ○ And, finally, both endpoints could be outside the clipping boundary

- To facilitate the passing of vertices from one clipping stage to the next, the output from each clipper can be formulated as shown in Figure below.



Example:





Input Edge	Left Clipper	Right Clipper	Bottom Clipper	Top Clipper
{1, 2}:	(in – in) → {2}			
{2, 3}:	(in – out) → {2′}	{2, 2′}: (in – in) → {2′}		
{3, 1}:	(out – in) → {3′, 1}	{2′, 3′}: (in – in) → {3′}	{2′, 3′}: (in – out) → {2″}	
		{3′, 1}: (in – in) → {1}	{3′, 1}: (out – out) → { }	
		{1, 2}: (in – in) → {2}	{1, 2}: (out – in) → {1′, 2}	{2″, 1′}: (in – in) → {1′}
			{2, 2′}: (in – in) → {2′}	{1′, 2}: (in – in) → {2}
				{2, 2′}: (in – in) → {2′}
				{2′, 2″}: (in – in) → {2″}

9. Explain different types of logical input devices.

API defines six classes of logical input devices which are given below:

- **STRING:** A string device is a logical device that provides the ASCII values of input characters to the user program. This logical device is usually implemented by means of physical keyboard.
- **LOCATOR:** A locator device provides a position in world coordinates to the user program. It is usually implemented by means of pointing devices such as mouse or track ball.
- **PICK:** A pick device returns the identifier of an object on the display to the user program. It is usually implemented with the same physical device as the locator but has a separate software interface to the user program. In OpenGL, we can use a process of selection to accomplish picking.
- **CHOICE:** A choice device allows the user to select one of a discrete number of options. In OpenGL, we can use various widgets provided by the window system. A widget is a graphical interactive component provided by the window system or a toolkit. The Widgets include menus, scrollbars and graphical buttons. For example, a menu with n selections acts as a choice **device, allowing user to select one of 'n' alternatives.**
- **VALUATORS:** They provide analog input to the user program on some graphical systems; there are boxes or dials to provide value.
- **STROKE:** A stroke device returns array of locations. Example, pushing down a mouse button starts the transfer of data into specified array and releasing of button ends this transfer.

10. With code snippet explain picking concept.

- Picking is logical input operation that allows identifying an object on the display.
- The difficulty in performing picking is the forward nature of the rendering pipeline; we cannot go backward directly from the position of the mouse to primitives that are rendered close to the point on the screen.
- OpenGL supports a rendering mode called selection mode to do picking at the cost of an extra rendering each time we do a pick.
- To use OpenGL's selection mechanism, you first draw your scene into the frame buffer, and then you enter selection mode and redraw the scene. However, once you're in selection mode, the contents of the frame buffer don't change until you exit selection mode.
- When you exit selection mode, OpenGL returns a list of the primitives that intersect the viewing volume. Each primitive that intersects the viewing volume causes a selection hit.
- The list of primitives is actually returned as an array of integer-valued names and related data - the hit records - that correspond to the current contents of the name stack. You construct the name stack by loading names onto it as you issue primitive drawing commands while in selection mode.
- Thus, when the list of names is returned, you can use it to determine which primitives might have been selected on the screen by the user.

Picking Example:

```
#include<stdio.h>
#include<GL/glut.h>
void draw()
{
    glColor3f(1,0,0);
    glRectf(100,100,200,200);
    glColor3f(1,1,0);
    glRectf(300,300,400,400);
}

void process(int hit,int buffer[])
{
      int i,j,*ptr,names;
      printf("hits  %d\n",hit);
      ptr=buffer;
      //loop over number of hits
      for(i=0;i<hit;i++)
      {
            names=*ptr;
            //skip over number of names and depths
            ptr+=3;
            //check each name in recoed
            for(j=0;j<names;j++)
```

```
                {
                        if(*ptr==11)printf("Yellow square\n");
                        else if(*ptr==10)printf("Red square\n");
                        ptr++; //go to next hit record
                }
        }
}

void select(int x,int y)
{
        int hit,viewport[4];
        int buffer[100] //name stack buffer initialization
        glGetIntegerv (GL_VIEWPORT, viewport);

        glSelectBuffer(100,(GLuint*)buffer);
        glInitNames();
        glPushName(0);
        glRenderMode(GL_SELECT);
        glPushMatrix();  //save original viewing matrix

        /*setting up view for selection mode*/
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        /*50*50 pick area around the cursor click position*/
        /*must invert mouse y to get in world coordinates*/
        gluPickMatrix(x,viewport[3]-y,50,50,viewport);     //Picking
matrix
        gluOrtho2D(0,500,0,500);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glPushName(10); //push name of object to name stack
        glRectf(100,100,200,200); //redraw object in selection mode
        glLoadName(11);
        glRectf(300,300,400,400);
        glPopMatrix(); //restore viewing matrix
        hit=glRenderMode(GL_RENDER); //return to normal render mode
        process(hit,buffer);  //process  hits  from  selection  mode
rendering
        glutPostRedisplay(); //normal render
}

void mouse(int button,int state,int x,int y)
{
//Enter selection mode and do picking on left click
        if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
                select(x,y);
}

void display()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        draw();
        glFlush();
}
```

```
void init( )
{
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
      gluOrtho2D(0,500,0,500);
     glMatrixMode(GL_MODELVIEW);
}

int main(int argc,char **argv)
{
     glutInit(&argc,argv);
     glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
     glutInitWindowSize(500,500);
     glutCreateWindow("3d Gasket");
     init();
     glutDisplayFunc(display);
     glutMouseFunc(mouse);
     glutMainLoop();
}
```