COMUTER GRAPHICS AND VISUALIZATION

Module-5

5.1.1 INTERACTION

In the field of computer graphics, interaction refers to the manner in which the application program communicates with input and output devices of the system.

Ex: Image varying in response to the input from the user.

OpenGL doesn't directly support interaction in order to maintain portability. However, OpenGL provides the GLUT library. This library supports interaction with the keyboard, mouse etc and hence enables interaction. The GLUT library is compatible with many operating systems such as X windows, Current Windows, Mac OS etc and hence indirectly ensures the portability of OpenGL.

5.1.2 INPUT DEVICES

Input devices are the devices which provide input to the computer graphics application program. Input devices can be categorized in two ways:

- *Physical input devices*
- *Logical input devices*

PHYSICAL INPUT DEVICES

Physical input devices are the input devices which has the particular hardware architecture. The two major categories in physical input devices are:

***Keyboard devices** like standard keyboard, flexible keyboard, handheld keyboard etc. These are used to provide character input like letters.*

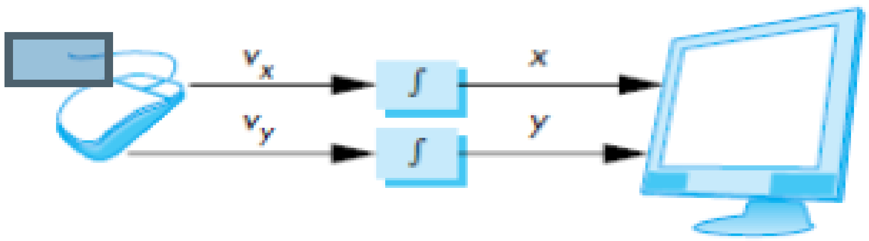
KEYBOARD

It is a general keyboard which has set of characters. We make use of ASCII value to represent the character i.e. it interacts with the programmer by passing the ASCII value of key pressed by programmer. Input can be given either single character of array of characters to the program.

MOUSE AND TRACKBALL: *These are pointing devices used to specify the position. Mouse and trackball interact with the application program by passing the position of the clicked button. Both these devices are similar in use and construction.*

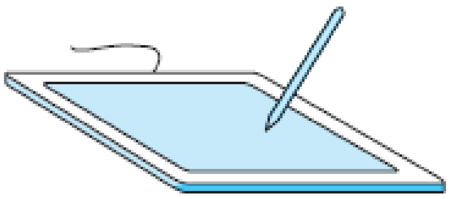
In these devices, the motion of the ball is converted to signal sent back to the computer by pair of encoders inside the device. These encoders measure motion in 2-orthogonal directions.

The values passed by the pointing devices can be considered as positions and converted to a 2-D location in either screen or world co-ordinates. Thus, as a mouse moves across a surface, the integrals of the velocities yield x,y values that can be converted to indicate the position for a cursor on the screen as shown below: These devices are relative positioning devices because changes in the position of the ball yield a position in the user program.



Data Tablets

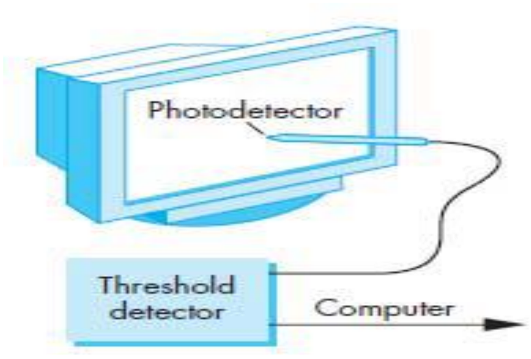
- *It provides absolute positioning.*
- *It has rows and columns of wires embedded under its surface.*
- *The position of the stylus is determined through electromagnetic interactions between signals travelling through the wires and sensors in the stylus.*



•

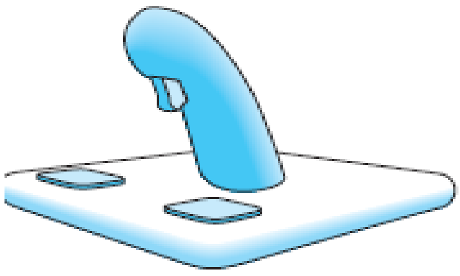
Light Pen

- *It consists of light-sensing device such as "photocell".*
- *The light pen is held at the front of the CRT.*
- *When the electron beam strikes the phosphor, the light is emitted from the CRT. If it exceeds the threshold, then light sensing device of the light pen sends a signal to the computer specifying the position*



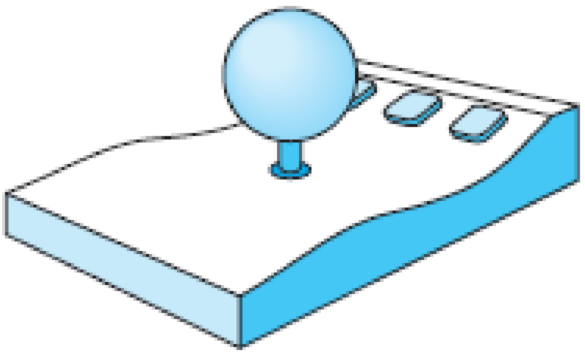
Joystick:

- *The motion of the stick in two orthogonal directions is encoded, interpreted as two velocities and integrated to identify a screen location.*
- *The integration implies that if the stick is left in its resting position, there is no change in cursor position.*
- *The faster the stick is moved from the resting position the faster the screen location changes.*
- *Joystick is variable sensitivity device.*
- *The advantage is that it is designed using mechanical elements such as springs and dampers which resist the user while pushing it.*
- *The mechanical feel is suitable for application such as the flight simulators, game controllers etc.*



Space Ball:

- *It is a 3-Dimensional input device which looks like a joystick with a ball on the end of the stick*
- *Stick doesn't move rather pressure sensors in the ball measure the forces applied by the user.*
- *The space ball can measure not only three direct forces (up-down, front-back, left-right) but also three independent twists.*
- *So totally device measures six independent values and thus has six degree of freedom.*



LOGICAL INPUT DEVICES

These are characterized by its high-level interface with the application program rather than by its physical characteristics. Two major characteristics describe the logical behavior of an input device:

(1) the measurements that the device returns to the user program.

(2) the time when the device returns those measurements.

Consider the following fragment of C code:

```
int x;
scanf("%d",&x);
printf("%d",x);
```

The above code reads and then writes an integer. Although we run this program on workstation providing input from keyboard and seeing output on the display screen, the use of scanf() and printf() requires no knowledge of the properties of physical devices such as keyboard codes or resolution of the display.

These are logical functions that are defined by how they handle input or output character strings from the perspective of C program. From logical devices perspective inputs are from inside the application program.

Six classes of logical input devices

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.*

glutKeyboardFunc()

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.*

glutMouseFunc()

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.*

bounding box technique

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.*

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.*

Analog Input, Control Dials, Sensing Devices

Stroke: *A stroke device returns array of locations.*

pushing down a mouse button starts the transfer of data into specified array and releasing of button ends this transfer.

Input Modes

How input devices provide input to an application program can be described in terms of two entities:

- ***Measure*** *of a device is what the device returns to the application program.*
- ***Trigger*** *of a device is a physical input on the device with which the user can send signal to the computer*

Example 1: *The measure of a keyboard is a single character or array of characters where as the trigger is the enter key.*

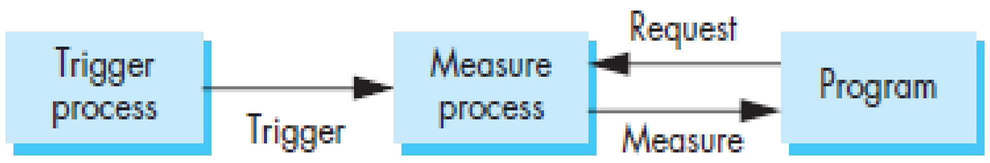
Example 2: *The measure of a mouse is the position of the cursor whereas the trigger is when the mouse button is pressed.*

The application program can obtain the measure and trigger in ***three distinct modes****:*

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

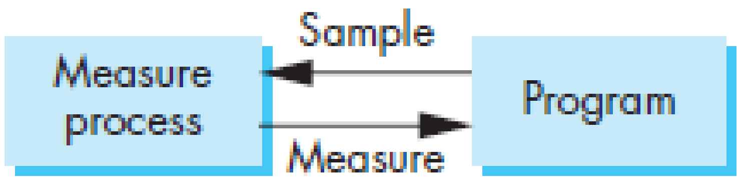
- *For example, consider a typical C program which reads a character input using scanf(). When the program needs the input, it halts when it encounters the scanf() statement and waits while user type characters at the terminal.*

- *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.*



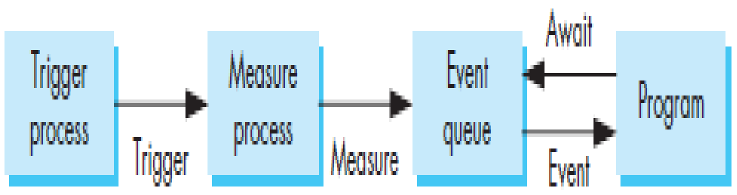
2. SAMPLE MODE

In sample mode, the user must have positioned the pointing device or entered data using the keyboard before the function call, because the measure is extracted immediately from the buffer.



3. 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.*



5.1.3 CLIENT AND SERVER

The computer graphics architecture is based on the client-server model.
*I.e., if computer graphics is to be useful for variety of real applications, it must function well in a world of distributed computing and network. In this architecture the building blocks are entities called as **"servers"*** perform the tasks requested by the ***"client"* .**Servers and clients can be distributed over a network or can be present within a single system. Today most of the computing is done in the form of distributed based and network based as shown below:

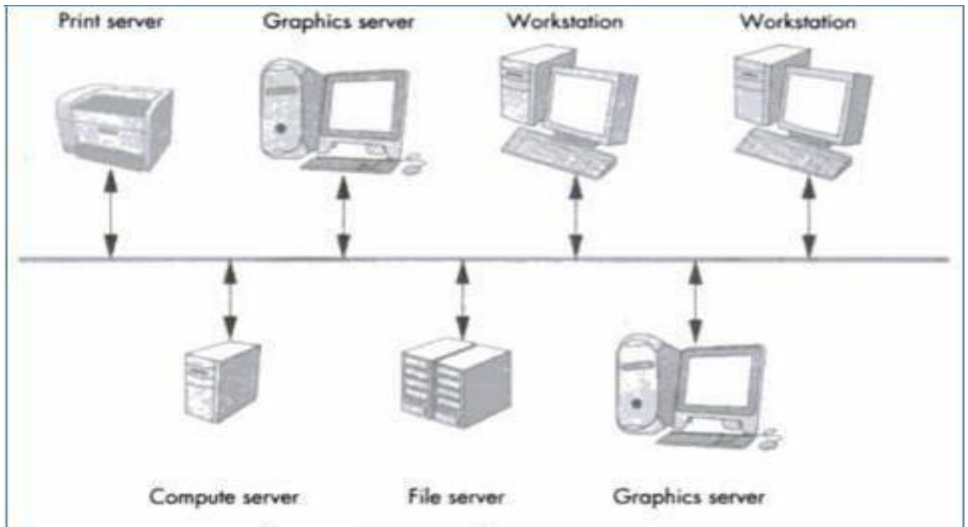


Figure: Network

Most popular examples of servers are print servers – which allow using high speed printer devices among multiple users. File servers – allow users to share files and programs. Users or clients will make use of these services with the help of user programs or client programs.

The OpenGL application programs are the client programs that use the services provided by the graphics server. Even if we have single user isolated system, the interaction would be configured as client-server model.

5.1.4 DISPLAY LISTS

The original architecture of a graphical system was based on a general-purpose computer connected to a display. The simple architecture is shown below.

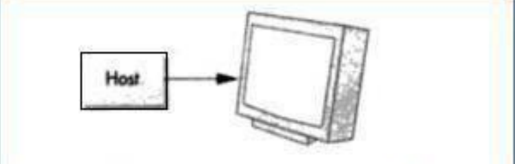
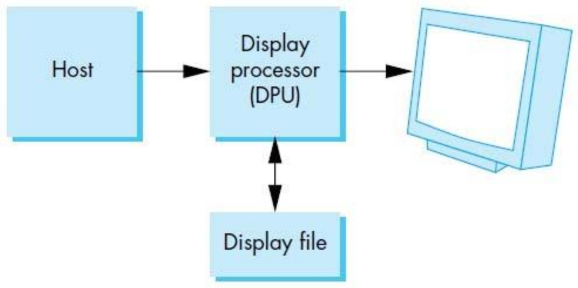


Figure: Simple graphics architecture

*At that time, the disadvantage is that system was slow and expensive. Therefore, a special purpose computer is build which is known as **"display processor"**.*
*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.



There are two modes in which objects can be drawn on the screen:

✓ ***IMMEDIATE MODE:*** *This mode sends the complete description of the object which needs to be drawn to the graphics server and no data can be retained. i.e., to redisplay the same object, the program must re-send the information. The information includes vertices, attributes, primitive types, viewing details.*

✓ ***RETAINEDMODE:*** *This mode is offered by the display lists. The object is defined once and its description is stored in a display list which is at the server side and redisplay of the object can be done by a simple function call issued by the client to the server.*

NOTE: *The main disadvantage of using display list is it requires memory at the server architecture and server efficiency decreases if the data is changing regularly.*

DEFINITION AND EXECUTION OF DISPLAY LISTS

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.

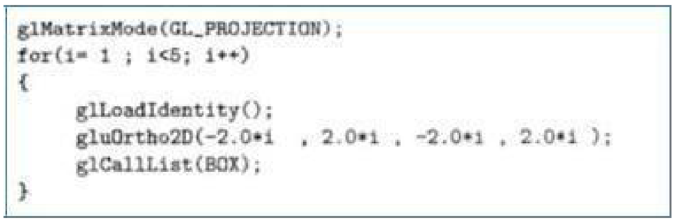
Each display list must have a unique identifier – an integer that is usually a macro defined in the C++ program by means of #define directive to an appropriate name for the object in the list.

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:

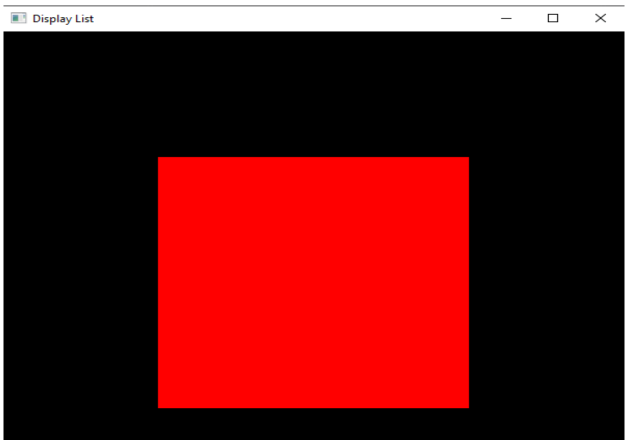
glCallList(Box)

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);
}
```

```
#include<stdio.h>
#include<gl/glut.h>
#define sq 10
void myinit()
{
		glMatrixMode(GL_PROJECTION_MATRIX);
		glLoadIdentity();
		gluOrtho2D(-100, 100, -100, 100);
		glMatrixMode(GL_MODELVIEW);
}
void display()
{
		glClearColor(0, 0, 0, 1);
		glClear(GL_COLOR_BUFFER_BIT);
		glColor3f(1, 0, 0);
		glNewList(sq, GL_COMPILE_AND_EXECUTE);
		glBegin(GL_POLYGON);
		glVertex2f(-50, -50);
		glVertex2f(-50, 50);
		glVertex2f(50, 50);
		glVertex2f(50, -50);
		glEnd();
		glEndList();
		glCallList(sq);
		glFlush();
}
void main()
{
		glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
		glutInitWindowSize(600, 600);
		glutInitWindowPosition(300, 150);
		glutCreateWindow("Display List");
		myinit();
		glutDisplayFunc(display);
		glutMainLoop();
}
```



We can save the present values of the attributes and the matrices by pushing them into the stack, usually the below function calls are placed at the beginning of the display list,

glPushAttrib(GL_ALL_ATTRIB_BITS);

glPushMatrix();

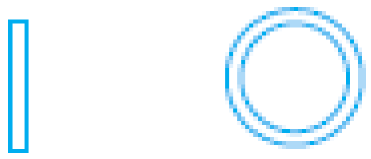
We can retrieve these values by popping them from the stack, usually the below function calls are placed at the end of the display list,

glPopAttrib();

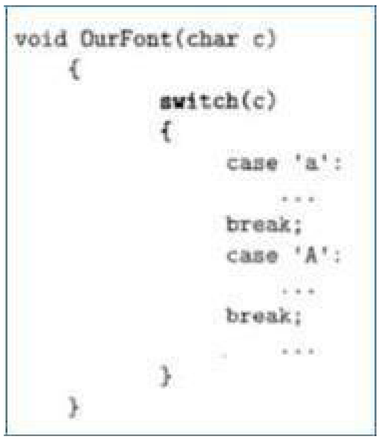
glPopMatrix();

TEXT AND DISPLAY LISTS

There are two types of text i.e., ***raster text and stroke text*** *which can be generated. For example, let us consider a raster text character is to be drawn of size 8x13 pattern of bits. It takes 13 bytes to store each character. If we define a stroke font using only line segments, each character requires a different number of lines.*

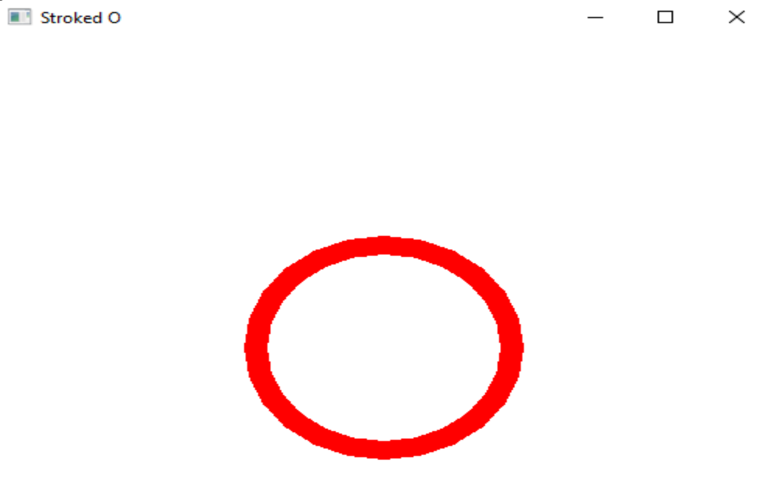


From the above figure we can observe to draw letter "I" we can use lines. Circle generation algorithm can be used for generating letter 'O'. Performance of the graphics system will be degraded for the applications that require large quantity of text if stroke text generation is used. A more efficient strategy is to define the font once, using a display list for each char and then store in the server. We define a function OurFont() which will draw any ASCII character stored in variable "c"



```
void OurFont(char c)
    {
            switch(c)
            {
                 case 'a':
                      ...
                 break;
                 case 'A':
                      ...
                 break;
                      ...
            }
    }
```

```
#include<stdio.h>
#include<math.h>
#include<gl/glut.h>

void myinit()
{
        glMatrixMode(GL_PROJECTION_MATRIX);
        glLoadIdentity();
        gluOrtho2D(-50, 50, -50, 50);
        glMatrixMode(GL_MODELVIEW);
}

void display()
{
        glClearColor(1, 1, 1, 1);
        glClear(GL_COLOR_BUFFER_BIT);

        int i;
        float x1, x2, y1, y2, r1 = 15, r2 = 18, t;

        glColor3f(1, 0, 0);
        glBegin(GL_QUAD_STRIP);
        for (i = 0; i <= 24; i++)
        {
                t = 3.142 / 12 * i;
                x1 = r1 * cos(t);
                y1 = r1 * sin(t);
                x2 = r2 * cos(t);
                y2 = r2 * sin(t);
                glVertex2f(x1, y1);
                glVertex2f(x2, y2);
        }
        glEnd();

        glFlush();
}

void main()
{
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(300, 150);

        glutCreateWindow("Stroked O");
        myinit();
        glutDisplayFunc(display);
        glutMainLoop();
}
```



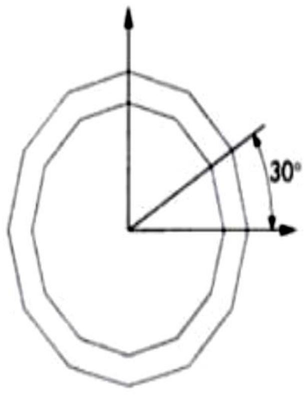
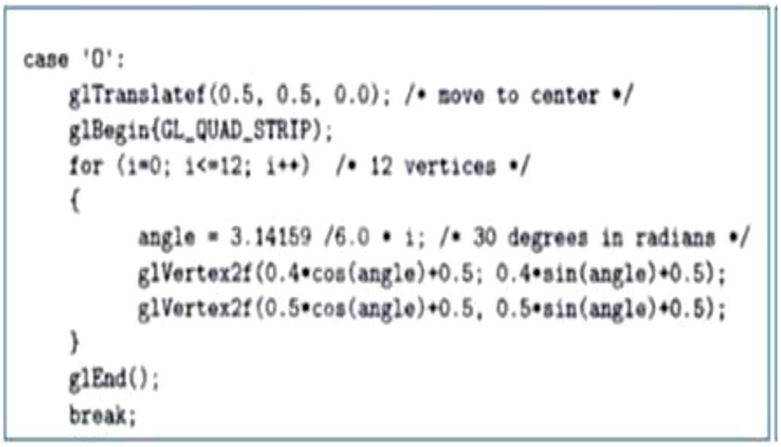


FIGURE Drawing of the letter "O."



```
case 'O':
    glTranslatef(0.5, 0.5, 0.0); /* move to center */
    glBegin(GL_QUAD_STRIP);
    for (i=0; i<=12; i++)  /* 12 vertices */
    {
        angle = 3.14159 /6.0 * i; /* 30 degrees in radians */
        glVertex2f(0.4*cos(angle)+0.5; 0.4*sin(angle)+0.5);
        glVertex2f(0.5*cos(angle)+0.5, 0.5*sin(angle)+0.5);
    }
    glEnd();
    break;
```

When we want to generate a 256-character set, the required code using OurFont() is as follows

```
base = glGenLists(256);
for(i=0;i<256;i++)
{
glNewList(base+i, GL_COMPILE);
  OurFont(i);
glEndList();

}
```

```
void OurFont(char c)
{
        switch(c)
        {
                case 'a':
                ... break;
                case 'A':
                ...
                break;
                ...
        }
}
```

When we wish to use these display lists to draw individual characters, rather than offsetting the identifier of the display lists by base each time, we can set an offset as follows:

glListBase(base);

*Finally, our drawing of a string is accomplished in the erver by the function call char *text_string;*

glCallLists((GLint) strlen(text_string), GL_BYTE, text_string);

which makes use of the standard C library function strlen to find the length of input string text_string. The first argument in the function glCallLists is the number of lists to be executed. The third is a pointer to an array of a type given by the second argument.

Fonts in Glut

In general, we prefer to use an existing font rather than to define our own. GLUT provides a few raster and stroke fonts.

glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)

We usually control the position of a character by using a translation before the character function is called.

glTranslatef(x, y, 0);

glutStrokeCharacter(GLUT_STROKE_ROMAN, 'a');

Raster and bitmap characters are produced in a similar manner. For example, a single 8 × 13 character is obtained using the following:

glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)

Positioning of bitmap characters is considerably simpler than the positioning of stroke characters is because bitmap characters are drawn directly in the frame buffer.

The position where the next raster primitive to be placed can be set using the glRasterPos(). glutBitmapCharacter function automatically advances the raster position, so typically we do not need to manipulate the raster position until we want to define a string of characters elsewhere on the display.*

DISPLAY LIST AND MODELING

Display list can call other display list. Therefore, they are powerful tools for building hierarchical models that can incorporate relationships among parts of a model. Consider a simple face modeling system that can produce images as follows.

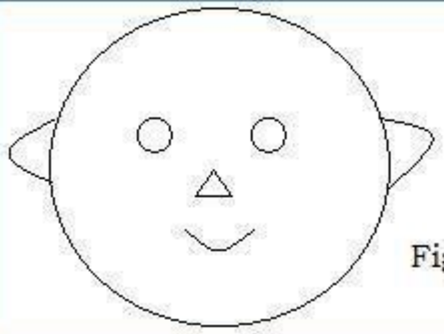
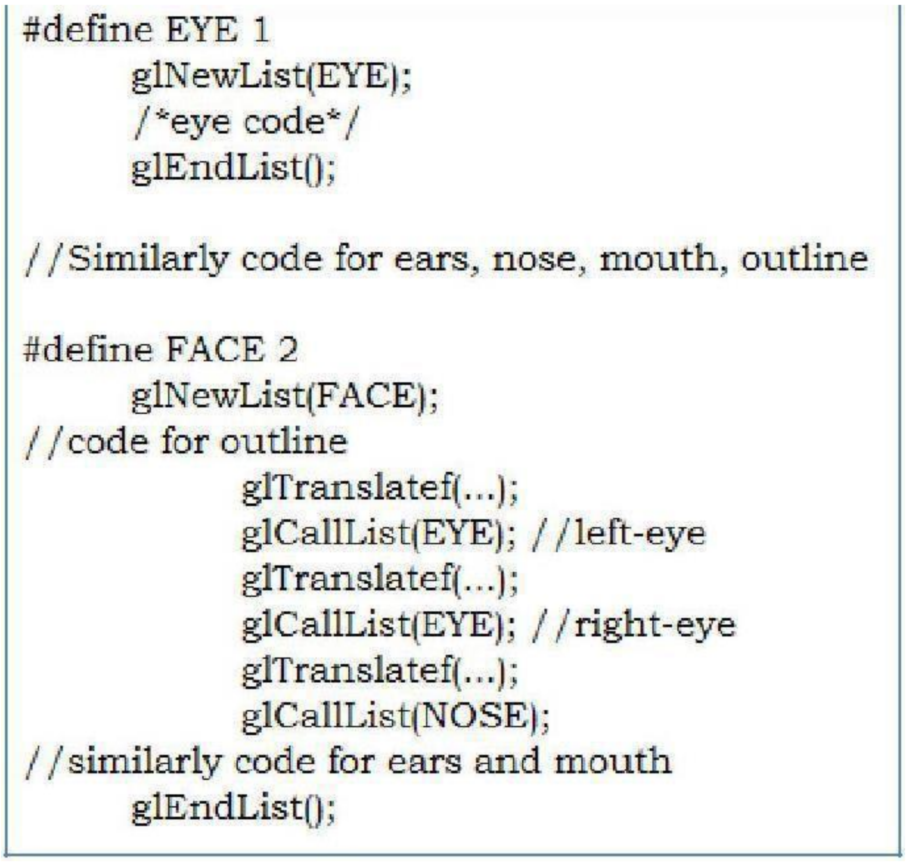


Figure: Simple animated face

Each face has two identical eyes, two identical ears, one nose, one mouth & an outline. We can specify these parts through display lists which is given below:



```
#define EYE 1
        glNewList(EYE);
        /*eye code*/
        glEndList();

//Similarly code for ears, nose, mouth, outline

#define FACE 2
        glNewList(FACE);
//code for outline
                glTranslatef(...);
                glCallList(EYE); //left-eye
                glTranslatef(...);
                glCallList(EYE); //right-eye
                glTranslatef(...);
                glCallList(NOSE);
//similarly code for ears and mouth
        glEndList();
```

What is the Necessity of Event driven input?

Computer Graphic Applications require input from various input devices and each device either works in request mode or sample mode and hence programming event driven input is required.

PROGRAMMING EVENT DRIVEN INPUT

In this we understand various events that are recognized by the window system and based on application, we write callback functions that indicate how the application program responds to these events.

Using Pointing Devices

Pointing devices like mouse, trackball, data tablet allow programmer to indicate a position on the display. There are two types of events associated with pointing device.

*A **move event** is generated when the mouse is moved with one of the buttons pressed. If the mouse is moved without a button being held down, this event is called a passive move event.*

After a move event, the position of the mouse—its measure—is made available to the application program.

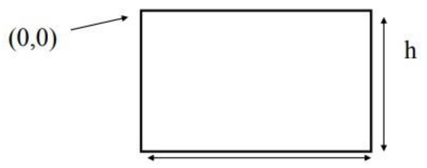
A mouse event occurs when one of the mouse buttons is either pressed or released. The information returned includes the button that generated the event(left button/right button/center button), the state of the button after the event (up or down), and the position of the cursor in window coordinates (with the origin in the upper-left corner of the window).We register the mouse callback function, usually in the main function as

glutMouseFunc(myMouse);

The mouse callback function must be defined in the form:

void myMouse(int button, int state, int x, int y)

When a user presses and releases mouse buttons in the window, each press and each release generate a mouse callback. The button parameter is one of GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, or GLUT_RIGHT_BUTTON. The state parameter is either GLUT_UP or GLUT_DOWN. The value of x & y is according to window, measured from topmost left corner as follows.

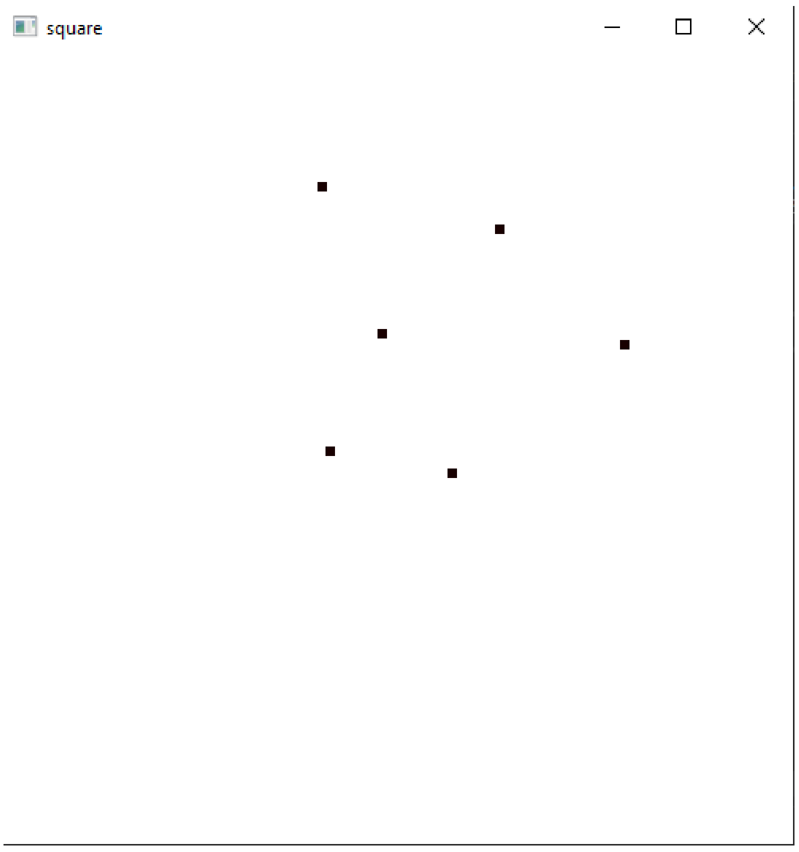


```
void myMouse(int button, int state, int x, int y)
{
if(button==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
exit(0);
}
```

The above code ensures whenever the left mouse button is pressed down, execution of the program gets terminated.

Write an OpenGL program to display square when a left button is pressed and to exit the program if right button is pressed.

```
#include<stdio.h>
#include<stdlib.h>
#include<GL/glut.h>
int wh = 500, ww = 500; float siz = 3;
void myinit()
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0, wh, 0, ww);// xmin, xmax, ymin, ymax
        glMatrixMode(GL_MODELVIEW);
}
void drawsq(int x, int y)
{
        y = wh - y;
        glColor3f(0.1, 0.0, 0.0);
        glBegin(GL_POLYGON);
        glVertex2f(x + siz, y + siz);
        glVertex2f(x - siz, y + siz);
        glVertex2f(x - siz, y - siz);
        glVertex2f(x + siz, y - siz);
        glEnd();
        glFlush();
}
void display()
{
        glClearColor(1, 1, 1, 1);
        glClear(GL_COLOR_BUFFER_BIT);
}
void myMouse(int button, int state, int x, int y)
{
        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
                drawsq(x, y);
        if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        {
            glClearColor(1, 1, 1, 1);
            glClear(GL_COLOR_BUFFER_BIT);
            glFlush();
        }//exit(0);
}
void main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
        glutInitWindowSize(wh, ww);
        glutCreateWindow("square");
        glutDisplayFunc(display);
        glutMouseFunc(myMouse);
        myinit();
        glutMainLoop();
}
```



Window Events

A window event is occurred when the corner of the window is dragged to new position or size of window is minimized or maximized by using mouse. The information returned to the program includes the height and width of newly resized window. The window callback function must be registered in the main function as

glutReshapeFunc(myReshape);

Window call back function must be defined as

```
void myReshape(GLsizei w, GLsizei h)
```

Returns width and height of new window.
Code segment of reshape function

```
 void myReshape(GLsizei w, GLsizei h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION_MATRIX);
 glLoadIdentity();
float t1 = (float)w/(float)h;
float t2 = (float)h/(float)w;
 if(w>h)
gluOrtho2D(-100*t1,100*t1,-100,100);
else
 gluOrtho2D(-100,100,-100*t2,100*t2);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}
```

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.

The GLUT function ***glutKeyboardFunc*** *is the callback for events generated by pressing a key, whereas* ***glutKeyboardUpFunc*** *is the callback for events generated by releasing a key.*

When a keyboard event occurs, the ASCII code for the key that generated the event and the location of the mouse are returned
glutKeyboardFunc(myKey);

The keyboard callback function must be defined in the form:
void mykey (unsigned char key, int x, int y)

key is the ASCII value of the char entered. The x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed. 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.

glutKeyboardFunc: registers callback handler for keyboard event.

*void glutKeyboardFunc (void (*func)(unsigned char key, int x, int y)*

// key is the char pressed, e.g., 'a' or 27 for ESC

// (x, y) is the mouse location in Windows' coordinates

glutSpecialFunc: registers callback handler for special key (such as arrow keys and function keys).

*void glutSpecialFunc (void (*func)(int specialKey, int x, int y)*

// specialKey: GLUT_KEY_* (for LEFT, RIGHT, UP, DOWN, HOME, END, PAGE_UP, PAGE_DOWN, F1,...F12).*

// (x, y) is the mouse location in Windows' coordinates

The Display and Idle Callbacks

*Display callback is specified by GLUT using glutDisplayFunc(myDisplay). It is invoked when GLUT determines that window should be redisplayed. Re-execution of the display function can be achieved by using **glutPostRedisplay().***

*The idle callback is invoked when there are no other events. It is specified by GLUT using **glutIdleFunc(myIdle).***

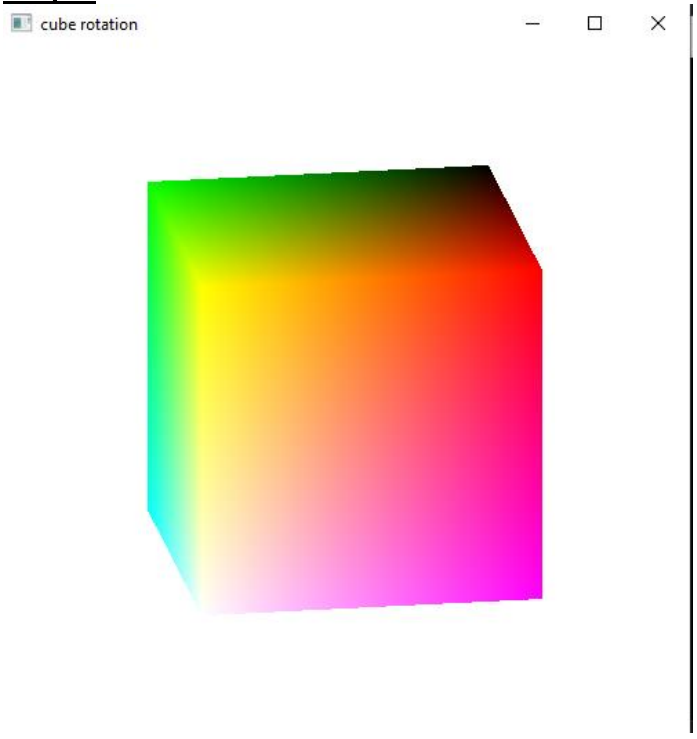
```
void spinCube()
{
/* Idle callback, spin cube 2 degrees about selected axis */
theta[axis] += 2.0;
if( theta[axis] > 360.0 )
   theta[axis] -= 360.0;
glutPostRedisplay();
}
void mouse(int btn, int state, int x, int y)
{
/* mouse callback, selects an axis about which to rotate */
if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN) axis = 0;
if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN) axis = 1;
if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN) axis = 2;
}

glutIdleFunc(spinCube);
glutMouseFunc(mouse);
```

Draw a color cube and spin it using OpenGL transformation matrices.

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

```
float v[8][3]={{-1,-1,1},{-1,1,1},{1,1,1},{1,-1,1},{-1,-1,-1},{-1,1,-1},{1,1,-1},{1,-1,-1}};
float p[8][3]={{0,0,1},{0,1,1},{1,1,1},{1,0,1},{0,0,0},{0,1,0},{1,1,0},{1,0,0}};
float theta[3]={0,0,0};
int flag=2;

void myinit()
{
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       glOrtho(-2,2,-2,2,-2,2);
       glMatrixMode(GL_MODELVIEW);
}
void idlefunc()
{
          theta[flag]++;
          if(theta[flag]>360)
          theta[flag]=0;
          glutPostRedisplay();
}
void mousefunc(int button,int status,int x,int y)
{
        if(button==GLUT_LEFT_BUTTON&&status==GLUT_DOWN)
            flag=2;
        if(button==GLUT_MIDDLE_BUTTON&&status==GLUT_DOWN)
            flag=1;
         if(button==GLUT_RIGHT_BUTTON&&status==GLUT_DOWN)
             flag=0;
}
void drawpoly(int a,int b,int c,int d)
{
       glBegin(GL_POLYGON);
           glColor3fv(p[a]);
           glVertex3fv(v[a]);
           glColor3fv(p[b]);
          glVertex3fv(v[b]);
          glColor3fv(p[c]);
          glVertex3fv(v[c]);
          glColor3fv(p[d]);
          glVertex3fv(v[d]);
     glEnd();
}
void colorcube()
{
        drawpoly(0,1,2,3);
        drawpoly(0,1,5,4);
        drawpoly(1,5,6,2);
        drawpoly(4,5,6,7);
        drawpoly(3,2,6,7);
        drawpoly(0,4,7,3);
}
void display()
{
       glClearColor(1,1,1,1);
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       glColor3f(1,0,0);
       glEnable(GL_DEPTH_TEST);
       glLoadIdentity();
       glRotatef(theta[0],1,0,0);//x
       glRotatef(theta[1],0,1,0);//y
      glRotatef(theta[2],0,0,1);//z
      colorcube();
      glFlush();
      glutSwapBuffers();
}
void main()
{
          glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE |GLUT_DEPTH);
          glutInitWindowPosition(100,100);
          glutInitWindowSize(500,500);
          glutCreateWindow("cube rotation");
          myinit();
          glutDisplayFunc(display);
          glutMouseFunc(mousefunc);
          glutIdleFunc(idlefunc);
          glutMainLoop();
}
```

Output



Window Management

GLUT also supports multiple windows and Sub windows of a given window. We can open a second top-level window

id=glutCreateWindow("second window");

The returned integer value allows us to select this window as the current window into which objects will be rendered as follows:
glutSetWindow(id);

Furthermore, each window can have its own set of callback functions because callback specifications refer to the present window.

MENUS

Menus are an important feature of any application program. OpenGL provides a feature called "Pop-up-menus" using which sophisticated interactive applications can be created. Menu creation involves the following steps:

1. Define the actions corresponding to each entry in the menu.
2. Link the menu to a corresponding mouse button.
3. Register a callback function for each entry in the menu.

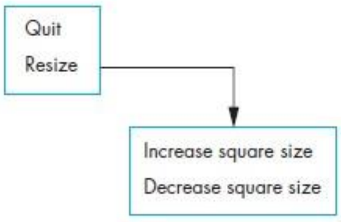
*For example we can demonstrate simple menus with the example of a pop-up menu that has three entries. The first selection allows us to exit our program. The second and third change the size of the squares in our **drawSquare** function. We name the menu callback demo_menu. The function calls to set up the menu and to link it to the right mouse button should be placed in our main function.*

```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit",1);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

The glutCreateMenu() registers the callback function demo_menu. The function glutAddMenuEntry() adds the entry in the menu whose name is passed in first argument and the second argument is the identifier passed to the callback when the entry is selected.

```
void demo_menu(int id)
{
switch(id)
{
case 1: exit(0);
break;
case 2: size = 2 * size;
break;
case 3: if(size > 1) size = size/2;
break;
}
glutPostRedisplay( );
}
```

GLUT also supports hierarchical menus, as shown in the below Figure.



Suppose that we want the main menu that we create to have only two entries. The firstentry still causes the program to terminate, but now the second causes a submenu topop up. The submenu contains the two entries for changing the size of the square inour square-drawing program.

```
  sub_menu = glutCreateMenu(size_menu);
  glutAddMenuEntry("Increase square size", 2);
   glutAddMenuEntry("Decrease square size", 3);
   glutCreateMenu(top_menu);
   glutAddMenuEntry("Quit",1);
   glutAddSubMenu("Resize", sub_menu);
   glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Picking

It is the logical input operation that allows the user to identify an object on the display.

The action of picking uses pointing device but the information returned to the application program is the identifier of an object not a position.

It is difficult to implement picking in modern system because of graphics pipeline architecture. Therefore, converting from location on the display to the corresponding primitive is not direct calculation.

There are at least three ways to deal with this difficulty

Selection: *It involves adjusting the clipping region and viewport such that we can keep track of which primitives lies in a small clipping region and are rendered into region near the cursor. These primitives are sent into a hit list that can be examined later by the user program.*

Bounding boxes or extents: *A simple approach is to use (axis-aligned) bounding boxes, or extents, for objects of interest. The extent of an object is the smallest rectangle, aligned with the coordinates axes, that contains the object.*

For two-dimensional applications, itis relatively easy to determine the rectangle in screen coordinates that corresponds toa rectangle point in object or world coordinates.

***Usage of back buffer and extra rendering**: When we use double buffering it has two color buffers: front and back buffers. The contents present in the front buffer is displayed, whereas contents in back buffer is not displayed so we can use back buffer for other than rendering the scene. Suppose that we render our objects into the back buffer, each in a distinct color. The application program will identify the object based on the color saved.*

Picking can be performed in four steps that are initiated by user defined pick function in the application:

- *We draw the objects into back buffer with the pick colors.*
- *We get the position of the mouse using the mouse callback.*
- *Use glReadPixels() to find the color at the position in the frame buffer corresponding to the mouse position.*
- *We search table of colors to find the object corresponds to the color read.*

PICKING AND SELECTION MODE

The difficult problem in implementing picking within the OpenGL pipeline is that we cannot go backward directly from the position of the mouse to primitives that were rendered close to that point on the screen.

OpenGL provides a somewhat complex process using a rendering mode called selection mode to do picking at the cost of an extra rendering each time that we do a pick.

Initially the application will be in GL_RENDER which is the default mode. When there is a click, we reduce the viewport to the region of click and change the mode to GL_SELECT. In GL_SELECT the objects primitives can be retrieved and It can be identified.

The function glRenderMode lets us select one of three modes: normal rendering to the color buffer (GL_RENDER),selection mode (GL_SELECT), or feedback mode (GL_FEEDBACK).

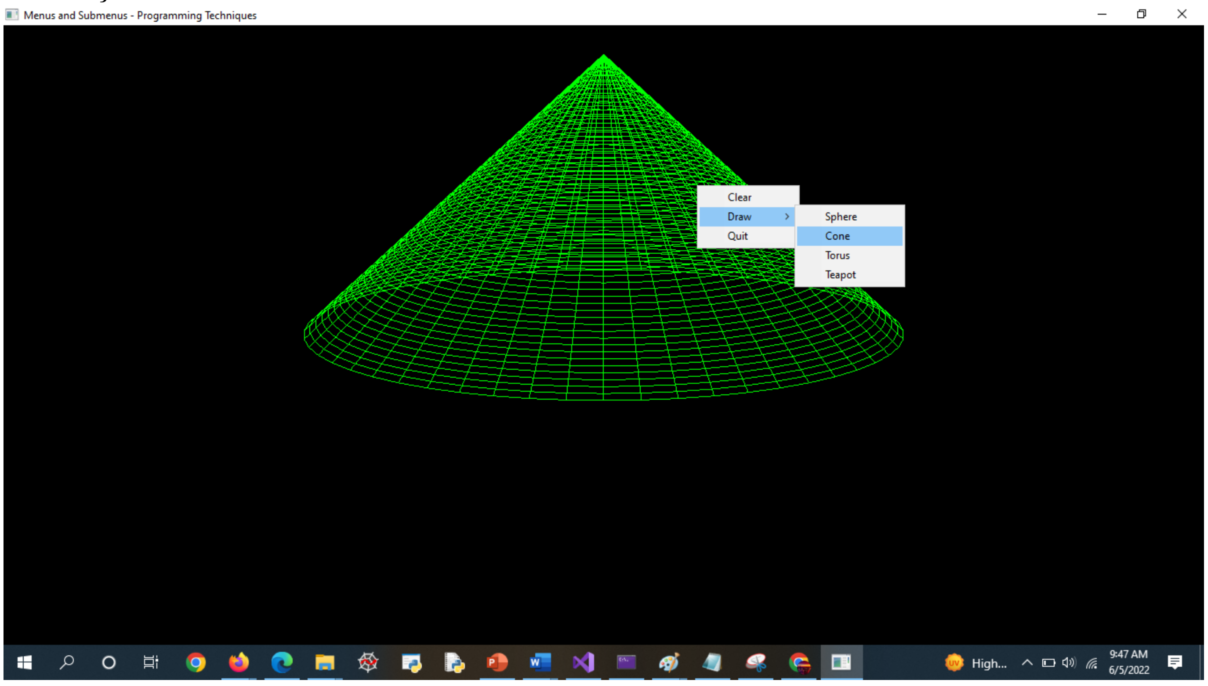
*When we enter selection mode and render a scene, each primitive within the clipping volume generates a message called a hit that is stored in a buffer called the **name stack**.*

The following functions are used in selection mode:

- *void glInitNames() // initializes the name stack.*
- *void glPushName(GLuint name) // pushes name on the name stack.*
  void glPopName() // pops the top name from the name stack.
- *void glLoadName(GLuint name) // replaces the top of the name stack with name.*
- *OpenGL allow us to set clipping volume for picking using gluPickMatrix() which is applied before gluOrtho2D.*
- *gluPickMatrix(x,y,w,h,*vp) : creates a projection matrix for picking that restricts drawing to a w * h area and centered at (x,y) in window coordinates within the viewport vp.*

Open GL Menu & Submenu

```
#include <windows.h>
#include <GL/glut.h>
static int window;
static int menu_id;
static int submenu_id;
static int value = 0;
void menu(int num) {
        if (num == 0) {
                glutDestroyWindow(window);
                exit(0);
        }
        else {
                value = num;
        }
        glutPostRedisplay();
}
void createMenu(void) {
        submenu_id = glutCreateMenu(menu);
        glutAddMenuEntry("Sphere", 2);
        glutAddMenuEntry("Cone", 3);
        glutAddMenuEntry("Torus", 4);
        glutAddMenuEntry("Teapot", 5);
          menu_id = glutCreateMenu(menu);
        glutAddMenuEntry("Clear", 1);
        glutAddSubMenu("Draw", submenu_id);
        glutAddMenuEntry("Quit", 0);
          glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        if (value == 1) {
                glutPostRedisplay();
        }
        else if (value == 2) {
                glPushMatrix();
                glColor3d(1.0, 0.0, 0.0);
                glutWireSphere(0.5, 50, 50);
                glPopMatrix();
        }
        else if (value == 3) {
                glPushMatrix();
                glColor3d(0.0, 1.0, 0.0);
                glRotated(65, -1.0, 0.0, 0.0);
                glutWireCone(0.5, 1.0, 50, 50);
                glPopMatrix();
        }
        else if (value == 4) {
                glPushMatrix();
                glColor3d(0.0, 0.0, 1.0);
                glutWireTorus(0.3, 0.6, 100, 100);
                glPopMatrix();
        }
        else if (value == 5) {
                glPushMatrix();
                glColor3d(1.0, 0.0, 1.0);
                glutSolidTeapot(0.5);
                glPopMatrix();
        }
        glFlush();

}
int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGBA | GLUT_SINGLE);
        glutInitWindowSize(500, 500);
        glutInitWindowPosition(100, 100);
        window = glutCreateWindow("Menus and Submenus");
        createMenu();
        glClearColor(0.0, 0.0, 0.0, 0.0);
        glutDisplayFunc(display);
        glutMainLoop();
        }
```



DESIGN OF INTERACTIVE PROGRAMS

The following are the features of most interactive program:

- *A smooth display, showing neither flicker nor any artifacts of the refresh process.*
- *A variety of interactive devices on the display*
- *A variety of methods for entering and displaying information*
- *An easy to use interface that does not require substantial effort to learn*
- *Feedback to the user.*
- *Tolerance for user errors*

- *A design that incorporates consideration of both the visual and motor properties of the human.*

The following two examples illustrate the limitations of geometric rendering.

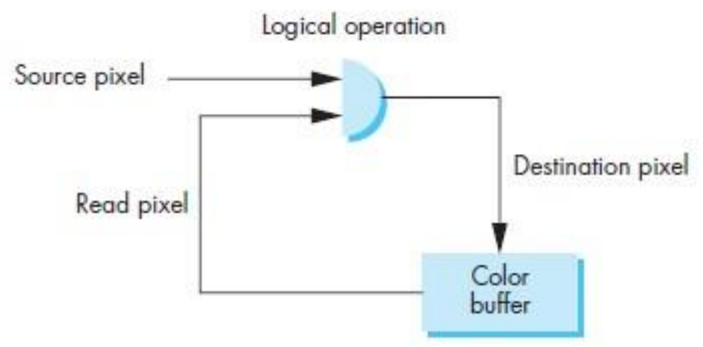
- ***Pop-up menus: When menu callback is invoked, the menu appears over whatever was on the display. After we make our selection, the menu disappears, and screen is restored to its previous state.***
- ***Rubberbanding: It is a technique used to define the elastic nature of pointing device to draw primitives***

LOGIC OPERATIONS

Rubberbanding, a technique used for displaying line segments (and other primitives) that change as they are manipulated interactively.

Two types of functions that define writing modes are: 1. Replacement mode 2. Exclusive OR (XOR)

When a program specifies a primitive that is visible, OpenGL renders it into a set of colored pixels that are written into the present drawing buffer. In the default mode of operation, these pixels simply replace the corresponding pixels that are already in the frame buffer, this mode, called copy, or replacement, mode, it does not matter what color the original pixels under the rectangle were before.



The pixel that we want to write is called the source pixel. The pixel in the drawing buffer that the source pixel will affect is called the destination pixel. In copy mode, the source pixel replaces the destination pixel.

The second mode is the exclusive OR or XOR mode in which corresponding bits in each pixel are combined using the exclusive or logical operation. If s and d are corresponding bits in the source and destination pixels, we can denote the new destination bit as d^i, and it is given by

d'= d xor s,

The most interesting property of the xor operation is that if we apply it twice, we return to the original state. That is,

d'= (d xor s) xor s.

Thus, if we draw something in xor mode, we can erase it by simply drawing it a second time.

To change mode, we must enable logic operation, glEnable(GL_COLOR_LOGIC_OP) and then it can change to XOR mode glLogicOp(GL_XOR). GL_COPY is the default operator.

Drawing Erasable Lines

We use the mouse to get the first endpoint and store this point in object coordinates as follows:

```
xm = x/500.;
ym = (500-y)/500.;
```

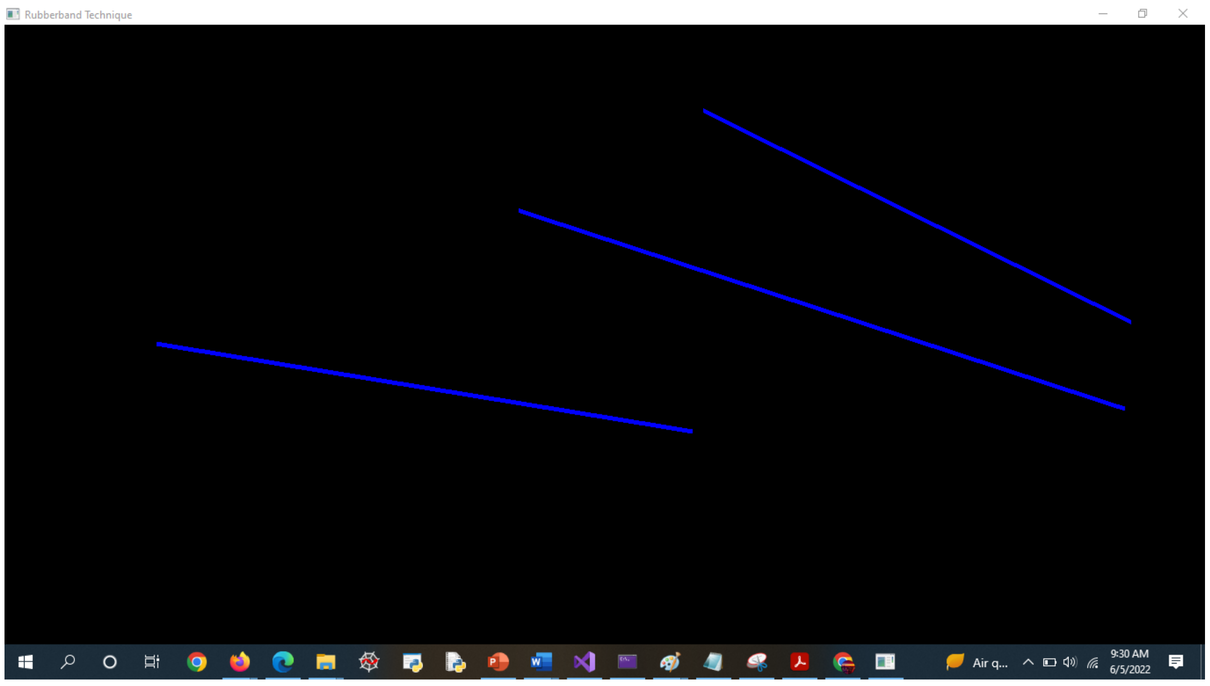
We then can get the second point and draw a line segment in xor mode:

```
xmm = x/500.;
ymm = (500-y)/500.;
glLogicOp(GL_XOR);
glBegin(GL_LINES);
glVertex2f(xm, ym);
 glVertex2f(xmm, ymm);
glLogicOp(GL_COPY);
 glEnd();
glFlush();
```

OpenGL -Rubberbanding

```
#include<stdio.h>
#include<gl/glut.h>
float xm, ym, xmm, ymm;
int first = 0, w = 600, h = 600;
void init()
{
	glMatrixMode(GL_PROJECTION_MATRIX);
	glLoadIdentity();
	gluOrtho2D(0, w, 0, h);
	glMatrixMode(GL_MODELVIEW);
}
void disp()
{
	glClearColor(0, 0, 0, 1);
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3f(0, 0, 1);
	glLineWidth(5);
	glFlush();
}
void mouse(int b, int s, int x, int y)
{
	glColor3f(0, 0, 1);
	y = h - y;
	if (b == GLUT_LEFT_BUTTON && s == GLUT_DOWN)
	{
		xm = x;
		ym = y;
		first = 0;
	}
	if (b == GLUT_LEFT_BUTTON && s == GLUT_UP)
	{
		glLogicOp(GL_XOR);
		glBegin(GL_LINES);
		glVertex2f(xm, ym);
		glVertex2f(xmm, ymm);
		glEnd();
		glFlush();
```

```
                    glLogicOp(GL_COPY);
                    glBegin(GL_LINES);
                    glVertex2f(xm, ym);
                    glVertex2f(xmm, ymm);
                    glEnd();
                    glFlush();
            }
            if (b == GLUT_RIGHT_BUTTON && s == GLUT_DOWN)
            {
                    glClearColor(0, 0, 0, 1);
                    glClear(GL_COLOR_BUFFER_BIT);
                    glFlush();
            }
            glFlush();
}
void move(int x, int y)
{
            y = h - y;
            if (first == 1)
            {
                    glLogicOp(GL_XOR);
                    glBegin(GL_LINES);
                    glVertex2f(xm, ym);
                    glVertex2f(xmm, ymm);
                    glEnd();
            }
            xmm = x;
            ymm = y;
            glLogicOp(GL_XOR);
            glBegin(GL_LINES);
            glVertex2f(xm, ym);
            glVertex2f(xmm, ymm);
            glEnd();
            glFlush();
            first = 1;
}
void main()
{
            glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
            glutInitWindowSize(600, 600);
            glutInitWindowPosition(300, 150);
            glutCreateWindow("Rubberband Technique");
            init();
            glEnable(GL_COLOR_LOGIC_OP);
            glutDisplayFunc(disp);
            glutMouseFunc(mouse);
            glutMotionFunc(move);
            glutMainLoop();
}
```



List the various Features that a good Interactive program should include

1. A smooth display, showing neither flicker nor any artifacts of the refresh process
2. A variety of interactive devices on the display
3. A variety of methods for entering and displaying information
4. An easy-to-use interface that does not require substantial effort to learn
5. Feedback to the user
6. Tolerance for user errors
7. A design that incorporates consideration of both the visual and motor properties of the human.

5.2 Curves

Curved surfaces
It is required to generate curved objects instead of polygons, for the curved objects the equation can be expressed either in parametric form or non-parametric form. The various objects that are often useful in graphics applications include quadric surfaces, super quadrics, polynomial and exponential functions, and spline surfaces.

Parametric form: *When the object description is given in terms of its dimensionality parameter, the description is termed as parametric representation. A curve in the plane has the form $C(t) = (x(t), y(t))$, and a curve in space has the form $C(t) = (x(t), y(t), z(t))$. The functions $x(t)$, $y(t)$ and $z(t)$ are called the coordinates functions. The image of $C(t)$ is called the trace of C, and $C(t)$ is called a parametrization of C. A parametric curve defined by polynomial coordinate function is called a polynomial curve. The degree of a polynomial curve is the highest power of the variable occurring in any coordinate function.*

***Non parametric form**: When the object descriptions are directly in terms of coordinates of reference frame, then the representation is termed as non-parametric. Example: a surface can be described in non-parametric form as: f1(x,y,z)=0 or z=f2(x,y). The coordinates (x, y) of points of a no parametric explicit planner curve satisfy y= f(x) or x = g(y). Such curve have the parametric form C(t) = (t, f(t)) or C(t) = (g(t), t)*

Quadric Surfaces

A frequently used class of objects are the quadric surfaces, which are described with second-degree equations. Quadric surfaces, particularly spheres and ellipsoids, are common elements of graphics scenes.

Sphere

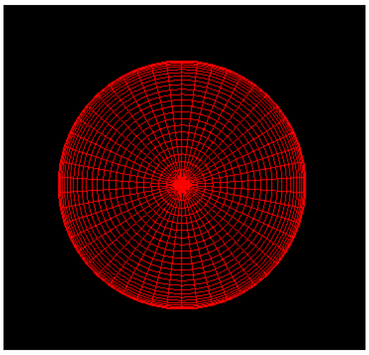
A spherical surface with radius r centered on the coordinate origin is defined as the set of points (x, y, z) that satisfy the equation

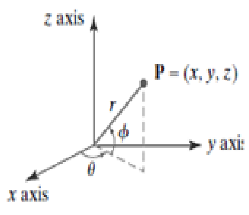
$$x^2 + y^2 + z^2 = r^2$$

We can also describe the spherical surface in parametric form, using latitude and longitude angles as shown in figure

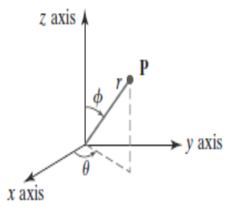
$$x = r \cos\varphi \cos\theta, \qquad -\pi/2 \le \varphi \le \pi/2$$
$$y = r \cos\varphi \sin\theta, \qquad -\pi \le \theta \le \pi$$
$$z = r \sin\varphi$$

Alternatively, we could write the parametric equations using standard spherical coordinates, where angle φ is specified as the colatitude. Then, φ is defined over the range $0 \le \varphi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$.





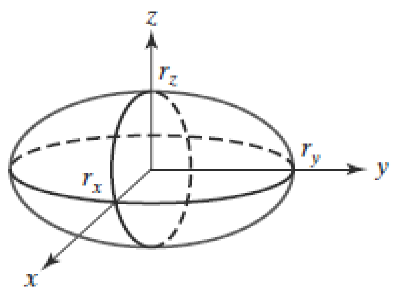
Parametric coordinate position (r, θ, φ) on the surface of a sphere with radius r



Spherical coordinate parameters (r, θ, φ), using colatitude for angle φ.

Ellipsoid

An ellipsoidal surface can be described as an extension of a spherical surface where the radii in three mutually perpendicular directions can have different values. The Cartesian representation for points over the surface of an ellipsoid centered on the origin is Torus. parametric representation for the ellipsoid in terms of the latitude angle φ and the longitude angle ϑ.



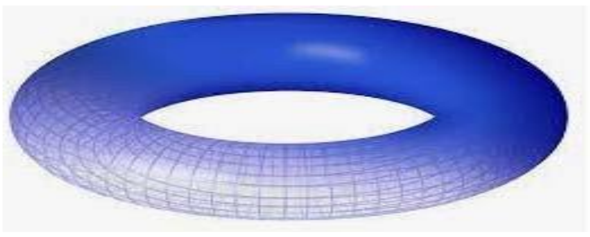
An ellipsoid with radii r_x, r_y, and r_z, centered on the coordinate origin.

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

$$x = r_x \cos\phi \cos\theta, \qquad -\pi/2 \le \phi \le \pi/2$$
$$y = r_y \cos\phi \sin\theta, \qquad -\pi \le \theta \le \pi$$
$$z = r_z \sin\phi$$

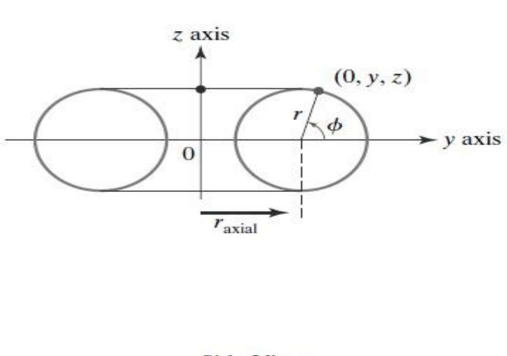
Torus

A doughnut-shaped object is called a torus or anchor ring. Most often it is described as the surface generated by rotating a circle or an ellipse about a coplanar axis line that is external to the conic. A torus generated by the rotation of a circle with radius r in the yz plane about the z axis is shown below.

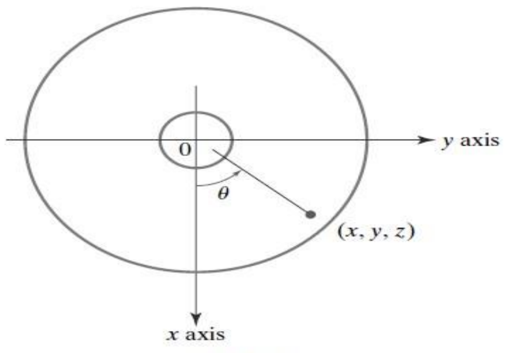


With the circle center on the y axis, the axial radius, raxial, of the resulting torus is equal to the distance along the y axis to the circle center from the z axis The equation for the cross-sectional circle shown in the side view of

$(y - r_{axial})^2 + z^2 = r^2$



Side View



Top View

Rotating this circle about the z axis produces the torus whose surface positions are described with the Cartesian equation. The corresponding parametric equations for the torus with a circular cross-section

$$\left(\sqrt{x^2+y^2}-r_{\text{axial}}\right)^2+z^2=r^2$$

$$\begin{aligned} x &= (r_{\text{axial}}+r\cos\phi)\cos\theta, & -\pi\le\phi\le\pi\\ y &= (r_{\text{axial}}+r\cos\phi)\sin\theta, & -\pi\le\theta\le\pi\\ z &= r\sin\phi \end{aligned}$$

We could also generate a torus by rotating an ellipse, instead of a circle, about the z axis. For an ellipse in the yz plane with semimajor and semi minor axes denoted as ry and rz, we can write the ellipse equation as

$$\left(\frac{y-r_{\text{axial}}}{r_y}\right)^2+\left(\frac{z}{r_z}\right)^2=1$$

where raxial is the distance along the y axis from the rotation z axis to the ellipse center. This generates a torus that can be described with the Cartesian equation

$$\left(\frac{\sqrt{x^2+y^2}-r_{\text{axial}}}{r_y}\right)+\left(\frac{z}{r_z}\right)^2=1$$

The corresponding parametric representation for the torus with an elliptical cross section is

$$\begin{aligned} x &= (r_{\text{axial}}+r_y\cos\phi)\cos\theta, & -\pi\le\phi\le\pi\\ y &= (r_{\text{axial}}+r_y\cos\phi)\sin\theta, & -\pi\le\theta\le\pi\\ z &= r_z\sin\phi \end{aligned}$$

OpenGL Quadric-Surface and Cubic-Surface Functions

Several other three-dimensional quadric-surface objects can be displayed using functions that are included in the OpenGL Utility Toolkit (GLUT) and in the OpenGL Utility (GLU). With the GLUT functions, we can display a sphere, cone, torus, or the teapot. With the GLU functions, we can display a sphere, cylinder, tapered cylinder, cone, flat circular ring (or hollow disk), and a section of a circular ring (or disk).

GLUT Quadric-Surface Functions

Sphere

We generate a GLUT sphere with either of these two functions:

glutWireSphere (r, nLongitudes, nLatitudes);

or
glutSolidSphere (r, nLongitudes, nLatitudes);

where the sphere radius is determined by the double-precision floating-point number assigned to parameter r. Parameters nLongitudes and nLatitudes are used to select the integer number of longitude and latitude lines that will be used to approximate the spherical surface as a quadrilateral mesh. Edges of the quadrilateral surface patches are straight-line approximations of the longitude and latitude lines. The sphere is defined in modeling coordinates, centered at the world-coordinate origin with its polar axis along the z axis.

Cone

A GLUT cone is obtained with either of these two functions:

glutWireCone (rBase, height, nLongitudes, nLatitudes);
or
glutSolidCone (rBase, height, nLongitudes, nLatitudes);

We set double-precision, floating-point values for the radius of the cone base and for the cone height using parameters rbase and height, respectively. As with a GLUT sphere, parameters nLongitudes and nLatitudes are assigned integer values that specify the number of orthogonal surface lines for the quadrilateral mesh approximation. A cone longitude line is a straight-line segment along the cone surface from the apex to the base that lies in a plane containing the cone axis. Each latitude line is displayed as a set of straight-line segments around the circumference of a circle on the cone surface that is parallel to the cone base and that lies in a plane perpendicular to the cone axis. The cone is described in modeling coordinates, with the center of the base at the world-coordinate origin and with the cone axis along the world z axis.

glutWireCone (0.7, 2.0, 7, 6);
Torus
glutWireTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);
or
glutSolidTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);

- *rCrossSection radius about the coplanar z axis*
- *rAxial is the distance of the circle center from the z axis*
- *nConcentrics specifies the number of concentric circles (with center on the z axis) to be used on the torus surface,*
- *nRadialSlices specifies the number of radial slices through the torus surface*

GLUT Cubic-Surface Teapot Function:

glutWireTeapot (size);

or

glutSolidTeapot (size);

The teapot surface is generated using OpenGL Bezier curve functions. Parameter size sets the double-precision floating-point value for the maximum radius of the teapot bowl. The teapot is centered on the world-coordinate origin coordinate origin with its vertical axis along the y axis.

Quadrics

Algebraic surface f(x,y,z) is the sum of polynomials in x, y, z

Quadric surface are Algebraic surfaces with a degree up to 2

Examples: x, y, xy, z^2, but not xy^2

GLU Quadric-Surface Functions

To generate a quadric surface using GLU functions
1. assign a name to the quadric
2.activate the GLU quadric renderer.
3.designate values for the surface parameters.

The following statements illustrate the basic sequence of calls for displaying a wire-frame sphere centered on the world-coordinate origin.

- *GLUquadricObj *sphere1;*
- *sphere1 = gluNewQuadric( );*
- *gluQuadricDrawStyle (sphere1, GLU_LINE);*
- *gluSphere (sphere1, r, nLongitudes, nLatitudes);*

Where,sphere1 is the name of the object the quadric renderer is activated with the gluNewQuadric function, and then the display mode GLU_LINE is selected for sphere1 with the gluQuadricDrawStyle command.

Parameter r is assigned a double-precision value for the sphere radius nLongitudes and nLatitudes. number of longitude lines and latitude lines Three other display modes are available for GLU quadric surfaces GLU_POINT: quadric surface is displayed as point plot ***GLU_SILHOUETTE:*** *quadric surface displayed will not contain shared edges between two coplanar polygon facets*

GLU_FILL: *quadric surface is displayed as patches of filled area. Three steps to create a cylinder:*
1. Create a GLU quadric object

*GLUquadricObj *p = gluNewQuadric();*

2. Set to wire frame mode

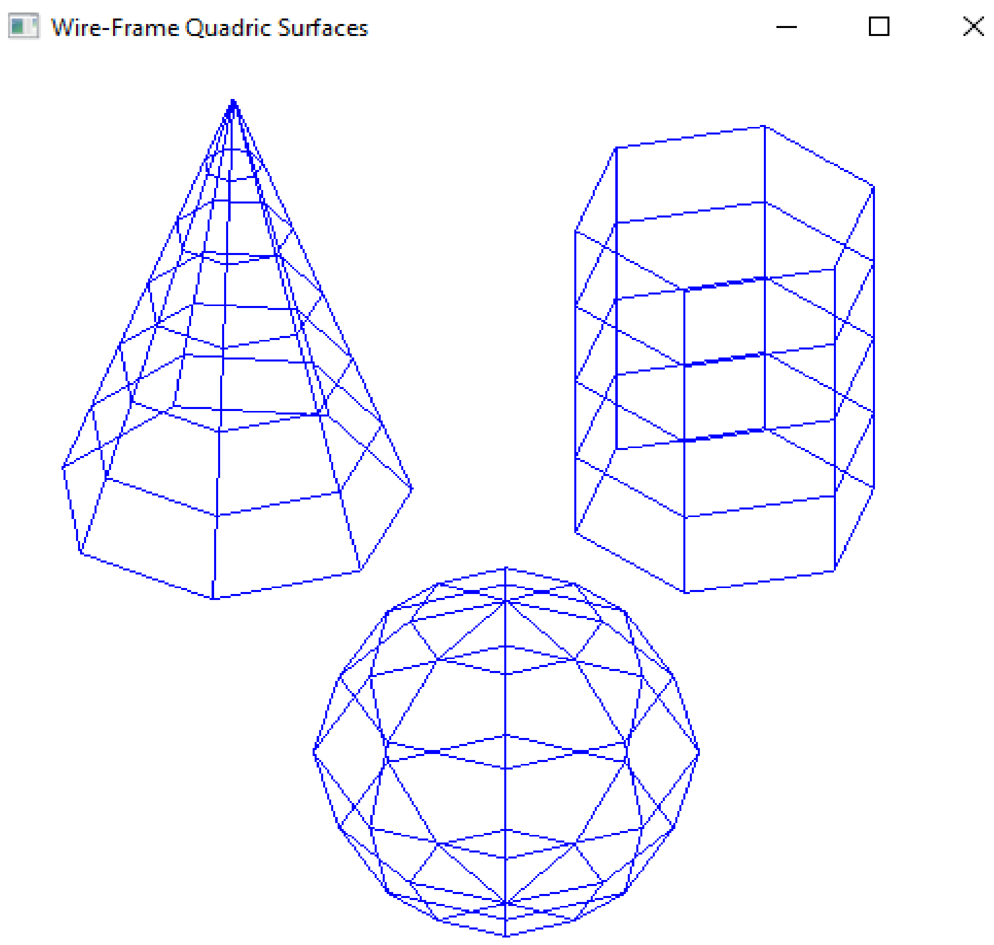
gluQuadricDrawStyle(GLU LINE);

3. Derive a cylinder object from p

gluCylinder(p, base, top, height, slice, stacks);

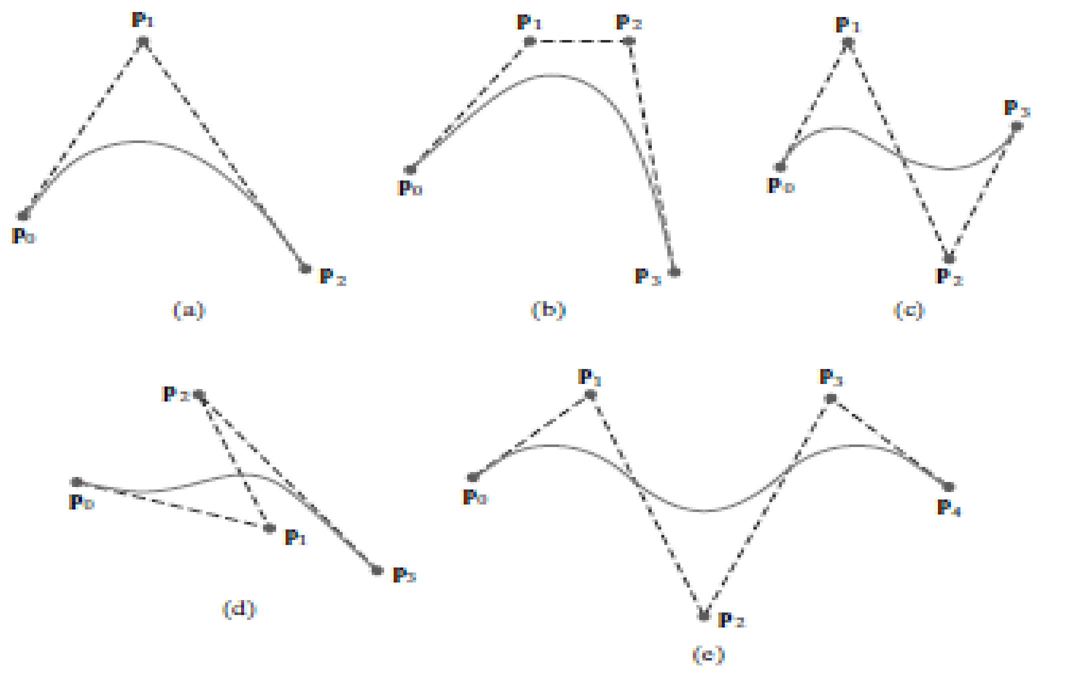
#include <GL/glut.h>

```
GLsizei winWidth = 500, winHeight = 500; // Initial display-window size.
void init(void)
{
        glClearColor(1.0, 1.0, 1.0, 0.0); // Set display-window color.
}
void wireQuadSurfs(void)
{
        glClear(GL_COLOR_BUFFER_BIT); // Clear display window.
        glColor3f(0.0, 0.0, 1.0); // Set line-color to blue.
        /* Set viewing parameters with world z axis as view-up direction. */
        gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
        /* Position and display GLUT wire-frame sphere. */
        glPushMatrix();
        glTranslatef(1.0, 1.0, 0.0);
        glutWireSphere(0.75, 8, 6);
        glPopMatrix();
        /* Position and display GLUT wire-frame cone. */
        glPushMatrix();
        glTranslatef(1.0, -0.5, 0.5);
        glutWireCone(0.7, 2.0, 7, 6);
        glPopMatrix();
        /* Position and display GLU wire-frame cylinder. */
        GLUquadricObj* cylinder; // Set name for GLU quadric object.
        glPushMatrix();
        glTranslatef(0.0, 1.2, 0.8);
        cylinder = gluNewQuadric();
        gluQuadricDrawStyle(cylinder, GLU_LINE);
        gluCylinder(cylinder, 0.6, 0.6, 1.5, 6, 4);
        glPopMatrix();
        glFlush();
}
void winReshapeFcn(GLint newWidth, GLint newHeight)
{
        glViewport(0, 0, newWidth, newHeight);
        glMatrixMode(GL_PROJECTION);
        glOrtho(-2.0, 2.0, -2.0, 2.0, 0.0, 5.0);
        glMatrixMode(GL_MODELVIEW);
        glClear(GL_COLOR_BUFFER_BIT);
}
void main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitWindowPosition(100, 100);
        glutInitWindowSize(winWidth, winHeight);
        glutCreateWindow("Wire-Frame Quadric Surfaces");
        init();
        glutDisplayFunc(wireQuadSurfs);
        glutReshapeFunc(winReshapeFcn);
        glutMainLoop();
}
```


Wire-Frame Quadric Surfaces

Summary of OpenGL Quadric-Surface and Cubic-Surface Functions

Function	Description
`glutWireSphere`	Displays a wire-frame GLUT sphere.
`glutSolidSphere`	Displays a surface-shaded GLUT sphere.
`glutWireCone`	Displays a wire-frame GLUT cone.
`glutSolidCone`	Displays a surface-shaded GLUT cone.
`glutWireTorus`	Displays a wire-frame GLUT torus with a circular cross-section.
`glutSolidTorus`	Displays a surface-shaded, circular cross-section GLUT torus.
`glutWireTeapot`	Displays a wire-frame GLUT teapot.
`glutSolidTeapot`	Displays a surface-shaded GLUT teapot.
`gluNewQuadric`	Activates the GLU quadric renderer for an object name that has been defined with the declaration: `GLUquadricObj *nameOfObject;`
`gluQuadricDrawStyle`	Selects a display mode for a predefined GLU object name.
`gluSphere`	Displays a GLU sphere.
`gluCylinder`	Displays a GLU cone, cylinder, or tapered cylinder.
`gluDisk`	Displays a GLU flat, circular ring or solid disk.
`gluPartialDisk`	Displays a section of a GLU flat, circular ring or solid disk.
`gluDeleteQuadric`	Eliminates a GLU quadric object.
`gluQuadricOrientation`	Defines inside and outside orientations for a GLU quadric object.
`gluQuadricNormals`	Specifies how surface-normal vectors should be generated for a GLU quadric object.
`gluQuadricCallback`	Specifies a callback error function for a GLU quadric object.



Examples of two-dimensional Bezier curves generated with three, four, and five control points. Dashed lines connect the control-point positions.

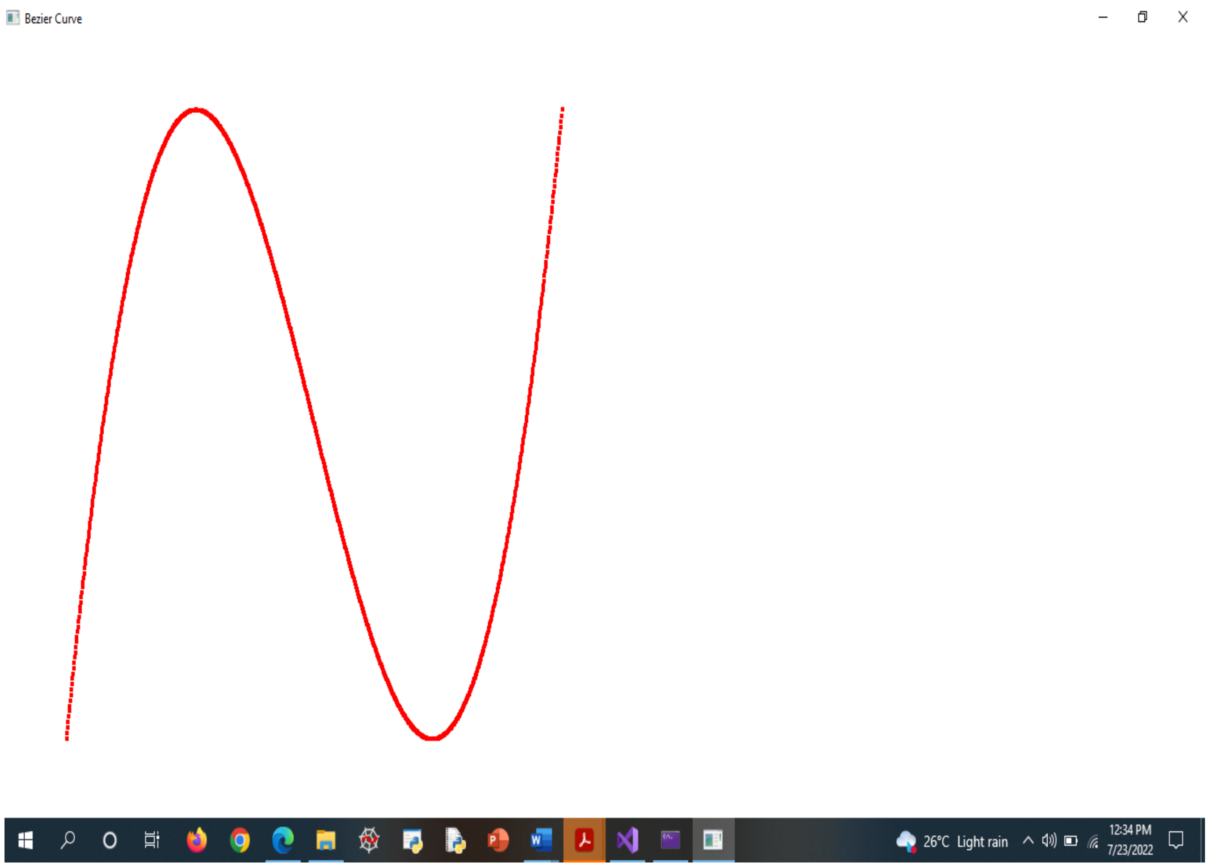
Bezier Curve

```
#include<GL/glut.h>
#include <stdlib.h>
#include <math.h>
/* Set initial size of the display window. */
GLsizei winWidth = 600, winHeight = 600;
/* Set size of world-coordinate clipping window.*/
GLfloat xwcMin = -50.0, xwcMax = 50.0;
GLfloat ywcMin = -50.0, ywcMax = 50.0;
class wcPt3D
{
public:
        GLfloat x, y, z;
};
void init(void)
{
        /* Set color of display window to white. */
        glClearColor(1.0, 1.0, 1.0, 0.0);
}
void plotPoint(wcPt3D bezCurvePt)
{
        glBegin(GL_POINTS);
        glVertex2f(bezCurvePt.x, bezCurvePt.y);
        glEnd();
}
void binomialCoeffs(GLint n, GLint* C)
{

        GLint k, j;
        for (k = 0; k <= n; k++) {
                /* Compute n!/(k!(n - k)!). */
                C[k] = 1;
                for (j = n; j >= k + 1; j--)
                        C[k] *= j;
                for (j = n - k; j >= 2; j--)
                        C[k] /= j;
        }
}
void computeBezPt(GLfloat u, wcPt3D* bezPt, GLint nCtrlPts, wcPt3D* ctrlPts, GLint* C)
{
        GLint k, n = nCtrlPts - 1;
        GLfloat bezBlendFcn;
        bezPt->x = bezPt->y = bezPt->z = 0.0;
        /* Compute blending functions and blend control points. */
        for (k = 0; k < nCtrlPts; k++) {
                bezBlendFcn = C[k] * pow(u, k) * pow(1 - u, n - k);
                bezPt->x += ctrlPts[k].x * bezBlendFcn;
                bezPt->y += ctrlPts[k].y * bezBlendFcn;
                bezPt->z += ctrlPts[k].z * bezBlendFcn;
        }
```

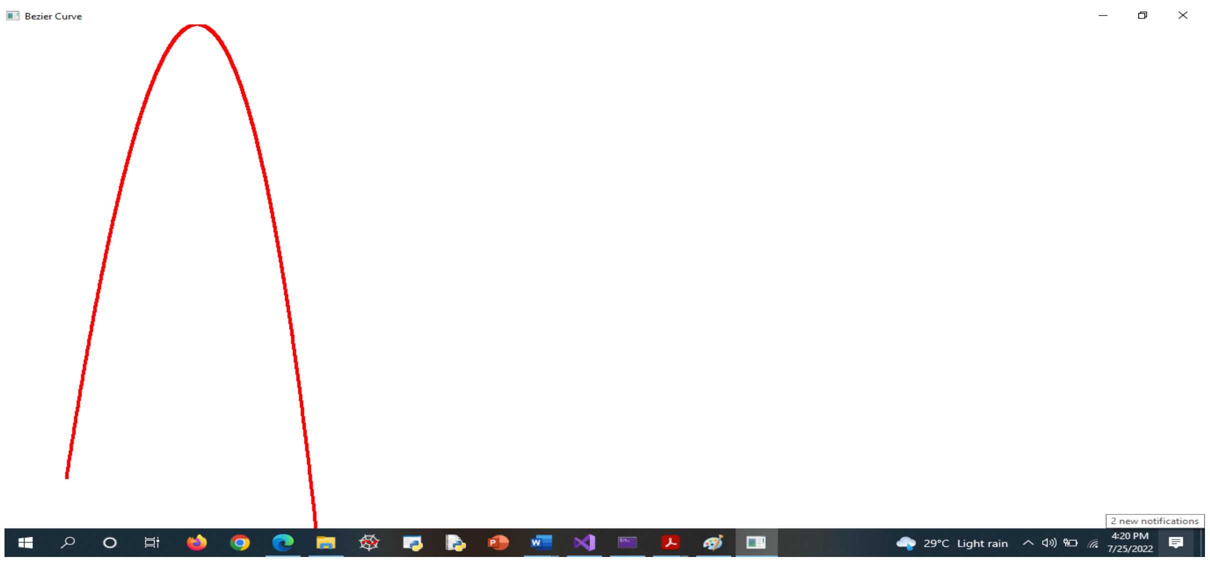
```
}
void bezier(wcPt3D* ctrlPts, GLint nCtrlPts, GLint nBezCurvePts)
{
        wcPt3D bezCurvePt;
        GLfloat u;
        GLint* C, k;
   C = new GLint[nCtrlPts];
        binomialCoeffs(nCtrlPts - 1, C);
        for (k = 0; k <= nBezCurvePts; k++)
        {
        u = GLfloat(k) / GLfloat(nBezCurvePts);
        computeBezPt(u, &bezCurvePt, nCtrlPts, ctrlPts,C);
        plotPoint(bezCurvePt);
   }
delete[ ] C;
}
void displayFcn(void)
{
/* Set example number of control points and number of curve positions to be plotted along
the Bezier curve. */
GLint nCtrlPts = 2, nBezCurvePts = 1000;
wcPt3D ctrlPts[4] = { {-40.0, -40.0, 0.0}, {-10.0, 200.0, 0.0},{10.0, -200.0, 0.0}, {40.0, 40.0,
0.0} };
glClear(GL_COLOR_BUFFER_BIT); // Clear display window.
glPointSize(4);
glColor3f(1.0, 0.0, 0.0); // Set point color to red.
bezier(ctrlPts, nCtrlPts, nBezCurvePts);
glFlush();
}
void winReshapeFcn(GLint newWidth, GLint newHeight)
{
        /* Maintain an aspect ratio of 1.0. */
        glViewport(0, 0, newHeight, newHeight);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(xwcMin, xwcMax, ywcMin, ywcMax);
        glClear(GL_COLOR_BUFFER_BIT);
}
void main(int argc, char** argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
        glutInitWindowPosition(50, 50);
        glutInitWindowSize(winWidth, winHeight);
        glutCreateWindow("Bezier Curve");
        init();
        glutDisplayFunc(displayFcn);
        glutReshapeFunc(winReshapeFcn);
        glutMainLoop();
```

}

Four Control Points



Three Control Points



Problem:

Given a Bezier curve with 4 control points

p0[1 0] ,p1[3 3] , p2[6 3] , p3[8 1.]Determine any 6 points lying on the curve. Also, draw a rough sketch of the curve.

Use Bezier Curve Equation

To get 5 points lying on the curve, assume any 6 values of u lying in the range 0 <= u<= 1.

Let 5 values of u are 0, 0.2, 0.4, 0.6, 0.8, 1

For u=0 we get

$P(u) = 1(1-0)^3 + 3*3*0(1-u)^2 + p_2 3*0(1-u) + p_3 0$

x(0)=1

$P(u) = 0(1-0)^3 + 3*3*0(1-u)^2 + p_2 3*0(1-u) + p_3 0$

y(0)=0

For u=0.2

$P(0.2) = 1(1-0.2)^3 + 3*3*0.2(1-0.2)^2 + 6*3*\ 0.2^2(1-0.2) + 8*0.2^3$

$P(0.2) = 0*(1-0.2)^3 + 3*3*0.2(1-0.2)^2 + 3*3*\ 0.2^2(1-0.2) + 1*0.2^3$

x=2.304

y=1.448

For u=0.4

$P(0.4) = 1(1-0.4)^3 + 3*3*0.4(1-0.4)^2 + 6*3*\ 0.4^2(1-0.4) + 8*0.4^3$

$P(0.4) = 0*(1-0.4)^3 + 3*3*0.4(1-0.4)^2 + 3*3*\ 0.4^2(1-0.4) + 1*0.4^3$

x=3.75

y=2.22

For u=0.6

$P(0.6) = 1(1-0.6)^3 + 3*3*0.6(1-0.6)^2 + 6*3*\ 0.6^2(1-0.6) + 8*0.6^3$

$P(0.6) = 0*(1-0.6)^3 + 3*3*0.6(1-0.6)^2 + 3*3*\ 0.6^2(1-0.6) + 1*0.6^3$

x=5.24

y=2.37

For u=0.8

$P(0.8) = 1(1-0.8)^3 + 3*3*0.8(1-0.8)^2 + 6*3*\ 0.8^2(1-0.8) + 8*0.8^3$

$P(0.8) = 0*(1-0.8)^3 + 3*3*0.8(1-0.8)^2 + 3*3*\ 0.8^2(1-0.8) + 1*0.8^3$

x=6.69

y= 1.95

For u=0.8

x=8

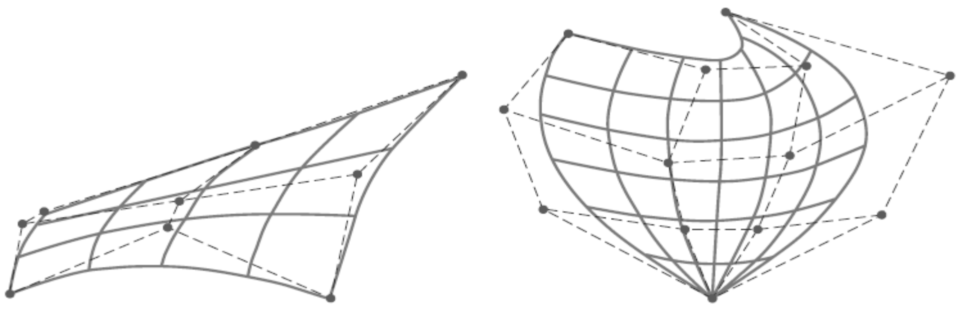
y= 1

Bezier Surfaces

Two sets of orthogonal Bezier curves can be used to design an object surface. The parametric vector function for the Bezier surface is formed as the tensor product of Bezier blending functions.

$$\mathbf{P}(u, v) = \sum_{j=0}^{m} \sum_{k=0}^{n} \mathbf{p}_{j,k}\, \mathrm{BEZ}_{j,m}(v)\, \mathrm{BEZ}_{k,n}(u)$$

with $p_{j,k}$ specifying the location of the (m + 1) by (n + 1) control points.



Bezier surfaces have the same properties as Bezier curves, and they provide a convenient method for interactive design applications. To specify the three dimensional coordinate positions for the control points, we could first construct rectangular grid in the xy "ground" plane. We then choose elevations above the ground plane at the grid intersections as the z-coordinate values for the control points.

B-Spline Curves

This spline category is the most widely used, and B-spline functions are commonly available in CAD systems and many graphics-programming packages. Like Bezier splines, B-splines are generated by approximating a set of control points. But B-splines have two advantages over Bézier splines:

1. *The degree of a B-spline polynomial can be set independently of the number of control points.*
2. *B-splines allow local control over the shape of a spline. The tradeoff is that B-splines are more complex than Bezier splines.*

We can write a general expression for the calculation of coordinate positions along a B-spline curve using a blending function formulation as:

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k B_{k,d}(u), \qquad u_{\min} \le u \le u_{\max}, \quad 2 \le d \le n+1$$

where pk is an input set of n + 1 control points. There are several differences between this B-spline formulation and the expression for a Bezier spline curve.

The range of parameter u now depends on how we choose the other B-spline parameters and the B-spline blending functions Bk,d are polynomials of degree d − 1, where d is the degree parameter.
The degree parameter d can be assigned any integer value in the range from 2 up to the number of control points (n + 1).
Blending functions for B-spline curves are defined by the Cox-deBoor recursion formulas.

$$B_{k,1}(u) = \begin{cases} 1 & \text{if } u_k \leq u \leq u_{k+1} \\ 0 & \text{otherwise} \end{cases}$$

$$B_{k,d}(u) = \frac{u - u_k}{u_{k+d-1} - u_k} B_{k,d-1}(u) + \frac{u_{k+d} - u}{u_{k+d} - u_{k+1}} B_{k+1,d-1}(u)$$

B-spline curves have the following properties.

• The polynomial curve has degree d −1 and Cd−2 continuity over the range of u.
• For n+1 control points, the curve is described with n+1 blending functions.
• Each blending function Bk,d is defined over d subintervals of the total range of u, starting at knot value uk .
• The range of parameter u is divided into n+d subintervals by the n+d +1 values specified in the knot vector.
With knot values labeled as {u0, u1, . . . ,un+d }, the resulting B-spline curve is defined only in the interval from knot value u_{d-1} up to knot value u_{n+1}.
• Each section of the spline curve (between two successive knot values) is influenced by d control points.
• Any one control point can affect the shape of at most d curve sections.

OpenGL Curve Functions

There are routines in the OpenGL Utility Toolkit (GLUT) that we can use to display some three-dimensional quadrics, such as spheres and cones, and some other shapes.
Another method we can use to generate a display of a simple curve is to approximate it using a polyline. We just need to locate a set of points along the curve path and connect the points with straight-line segments.

We specify parameters and activate the routines for Bezier-curve display with the OpenGL functions
glMap1 (GL_MAP1_VERTEX_3, uMin, uMax, stride, nPts, *ctrlPts);*
glEnable (GL_MAP1_VERTEX_3);
We deactivate the routines with
glDisable (GL_MAP1_VERTEX_3);
where,

- *A suffix code of f or d is used with glMap1 to indicate either floating-point or double precision for the data values.*

- *Minimum and maximum values for the curve parameter u are specified in uMin and uMax, although these values for a Bézier curve are typically set to 0 and 1.0, respectively.*
- *Bézier control points are listed in array ctrlPts number of elements in this array is given as a positive integer using parameter nPts.*
- *stride is assigned an integer offset that indicates the number of data values between the beginning of one coordinate position in array ctrlPts and the beginning of the next coordinate position.*
- *A coordinate position along the curve path is calculated with*
  glEvalCoord1 (uValue);*
  Where,parameter uValue is assigned some value in the interval from uMin to uMax.Function. glEvalCoord1 calculates a coordinate position using equation with the parameter value.

```
GLfloat ctrlpoints[4][3] = {
{ -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
{2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};
void init(void)
{
glClearColor(0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT);
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
glEnable(GL_MAP1_VERTEX_3);
}
void display(void)
{
int i;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_LINE_STRIP);
for (i = 0; i<= 30; i++)
glEvalCoord1f((GLfloat) i/30.0);
glEnd();
glFlush();
}
```

Bezier Surfaces in OpenGL

To handle surfaces, we just convert the OpenGL functions from the earlier section above to 2D. The basic functions are:

```
glMap2f(type, u_min, u_max, u_stride, u_order, v_min, v_max, v_stride,
v_order, point_array);
glEnable(type);
glEvalCoord2f(u,v);
then render with:
glMapGrid2f(8, 0.0, 1.0, 16, 0.0, 1.0);
glEvalMesh2(GL_LINE,0,NUM_S_STEPS,0,NUM_T_STEPS);
```

5.3 Animation

Computer animation *generally refers to any time sequence of visual changes in a picture. In addition to changing object positions using translations or rotations, a computer-generated animation could display time variations in object size, color, transparency, or surface texture. Animation adds to graphics the dimension of time which vastly increases the amount of information which can be transmitted.*

Two basic methods for constructing a motion sequence are:

- ***real-time animation***
  In a real-time computer-animation, each stage of the sequence is viewed as it is created. Thus the animation must be generated at a rate that is compatible with the constraints of the refresh rate.
- ***frame-by-frame animation***
  *In frame-by-frame animation, each frame of the motion is separately generated and stored. Later, the frames can be recorded on film, or they can be displayed consecutively on a vid**eo monitor in "real**-time **playback" mode.***

5.3.1 Raster Methods for Computer Animation

- *We can create simple animation sequences in our programs using real-time methods.*
- *We can produce an animation sequence on a raster-scan system one frame at a time, so that each completed frame could be saved in a file for later viewing.*
- *The animation can then be viewed by cycling through the completed frame sequence, or the frames could be transferred to film.*
- *If we want to generate an animation in real time, however, we need to produce the motion frames quickly enough so that a continuous motion sequence is displayed.*
- *The screen display is generated from successively modified pixel values in the refresh buffer, we can take advantage of some of the characteristics of the raster screen refresh process to produce motion sequences quickly.*

Double Buffering

- *One method for producing a real-time animation with a raster system to employ two refresh buffers. Then, while the screen is being refreshed from that buffer, we construct the next frame in the other buffer.*

- *When that frame is complete, we switch the roles of the two buffers so that the refresh routines use the second buffer during the process of creating the next frame in the first buffer.*
- *When a call is made to switch two refresh buffers, the interchange could be performed at various times.*
- *The most straight forward implementation is to switch the two buffers at the end of the current refresh cycle, during the vertical retrace of the electron beam.*
- *If a program can complete the construction of a frame within the time of a refresh cycle, say 1/60 of a second, each motion sequence is displayed in synchronization with the screen refresh rate.*
- *If the time to construct a frame is longer than the refresh time, the current frame is displayed for two or more refresh cycles while the next animation frame is being generated.*
- *Similarly, if the frame construction time is 1/25 of a second, the animation framerate is reducedto20frames per second because each frame is displayed three times.*
- *Irregular animation frame rates can occur with double buffering when the frame construction time is very nearly equal to an integer multiple of the screen refresh time the animation frame rate can change abruptly and erratically.*
- *One way to compensate for this effect is to add a small time delay to the program. Another possibility is to alter the motion or scene description to shorten the frame construction time.*

Generating Animations Using Raster Operations

- *We can also generate real-time raster animations for limited applications using block transfers of a rectangular array of pixel values.*
- *A simple method for translating an object from one location to another in the xy plane is to transfer the group of pixel values that define the shape of the object to the new location*
- *Sequences of raster operations can be executed to produce Realtime animation for either two-dimensional or three-dimensional objects.*
- *As long as we restrict the animation to motions in the projection plane then no viewing or visible-surface algorithms need be invoked.*
- *We can also animate objects along two-dimensional motion paths using* ***color table transformations.***
- *We predefine the object at successive positions along the motion path and set the successive blocks of pixel values to color-table entries.*

- *The pixels at the first position of the object are set to a foreground color, and the pixels at the other object positions are set to the background color .*
- *Then the animation is then accomplished by changing the color-table values so that the object color at successive positions along the animation path becomes the foreground color as the preceding position is set to the background color.*

5.3.2 Design of Animation Sequences

Animation sequence in general is designed in the following steps

- *Storyboard layout*
- *Object definitions*
- *Key-frame specifications*
- *Generation of in-between frames*

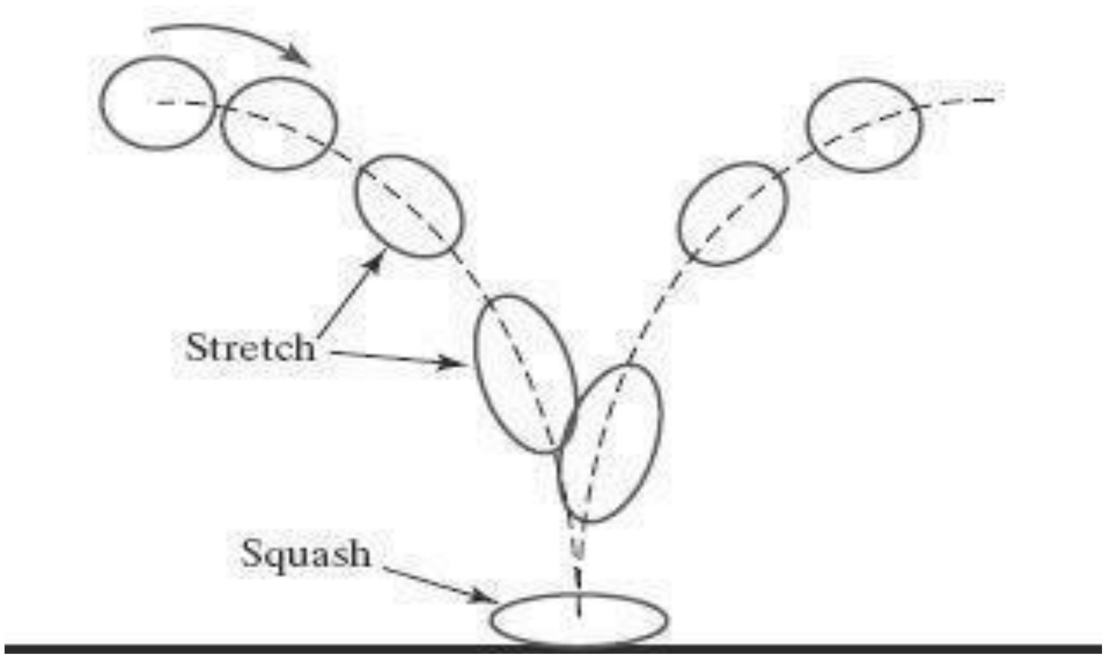
This approach of carrying out animations is applied to any other applications as well-thought-of some applications are exceptional cases and do not follow this sequence.

- *For frame-by-frame animation, every frame of the display or scene is generated separately and stored. Later, the frame recording can be done, and they might be displayed consecutively in terms of movie.*
- *The outline of the action is storyboard. This explains the motion sequence. The storyboard consists of a set of rough structures, or it could be a list of the basic ideas for the motion.*
- *For each participant in the action, an object definition is given. Objects are described in terms of basic shapes the examples of which are splines or polygons. The related movement associated with the objects are specified along with the shapes.*
- *A key frame in animation can be defined as a detailed drawing of the scene at a certain time in the animation sequence. Each object is positioned according to the time for that frame, within each key frame.*
- *Some key frames are selected at extreme positions and the others are placed so that the time interval between two consecutive key frames is not large. Greater number of key frames are specified for smooth motions than for slow and varying motion. The intermediate frames between the key frames are In-betweens.*
- *Media that we use determines the number of In-betweens which are required to display the animation. A Film needs 24 frames per second, and graphics terminals are refreshed at the rate of 30 to 60 frames per second.*

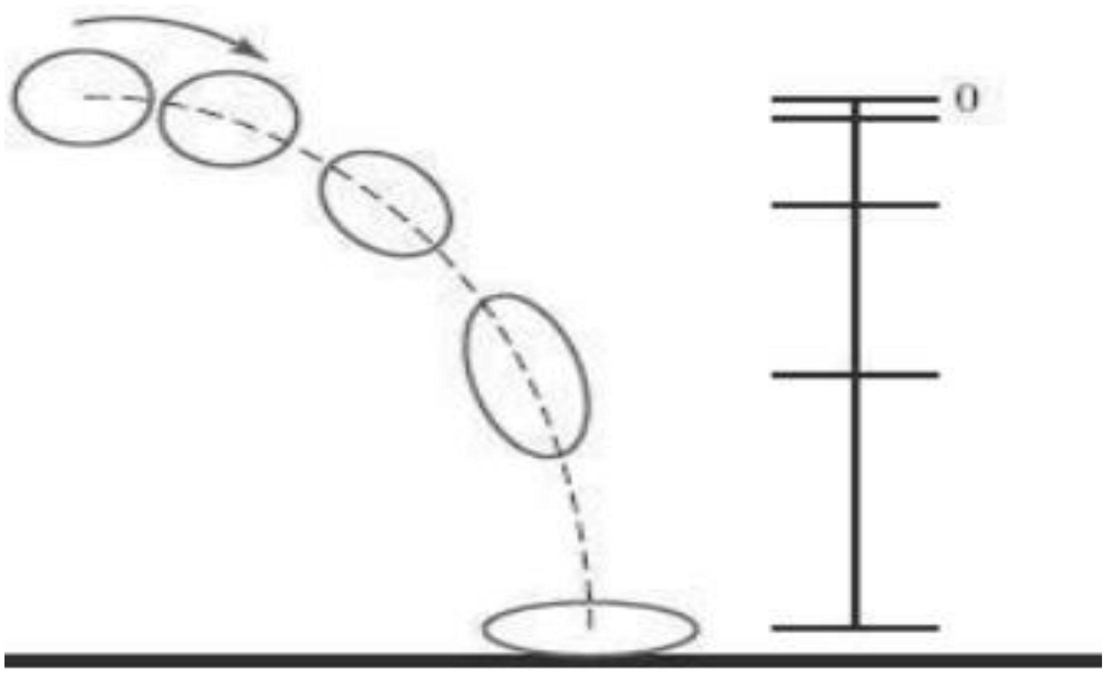
- *Depending on the speed specified for the motion, some key frames are duplicated. Fora one minutes film sequence with no duplication, we would require 288 key frames. We place the key frames a bit distant if the motion is not too complicated.*
- *Several other tasks may be carried out depending upon the application requirement for example synchronization of a soundtrack.*

5.3.3 Traditional Animation Techniques

Film animators use a variety of methods for depicting and emphasizing motion sequences. These include object deformations, spacing between animation frames, motion anticipation and follow through, and action focusing. One of the most important techniques for simulating acceleration effects, particularly for non-rigid objects, is ***squash and stretch.*** *Figure shows how this technique is used to emphasize the acceleration and deceleration of a bouncing ball. As the ball accelerates, it begins to stretch. When the ball hits the floor and stops, it is first compressed (squashed) and then stretched again as it accelerates and bounces upwards.*



Another technique used by film animators is timing, which refers to the spacing between motion frames. A slower moving object is represented with more closely spaced frames, and a faster moving object is displayed with fewer frames over the path of the motion.



General Computer-Animation Functions

Typical animation functions include managing object motions, generating views of objects, producing camera motions, and the generation of in-between frames.

- *Some animation packages, such as Wavefront for example, provide special functions for both the overall animation design and the processing of individual objects.*
- *Others are special-purpose packages for particular features of an animation, such as a system for generating in-between frames or a system for figure animation.*
- *A set of routines is often provided in a general animation package for storing and managing the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for generating the object motion and those for rendering the object surfaces.*
- *Another typical function set simulates camera movements. Standard camera motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweens can be generated automatically.*

OpenGL Animation Procedures

Double-buffering operations are activated using the following GLUT command:

glutInitDisplayMode (GLUT_DOUBLE);

This provides two buffers, called the front buffer and the back buffer, that we can use alternately to refresh the screen display.

We specify when the roles of the two buffers are to be interchanged using

glutSwapBuffers ( );

To determine whether double-buffer operations are available on a system, we can issue the following query:

glGetBooleanv (GL_DOUBLEBUFFER, status);

A value of GL_TRUE is returned to array parameter status if both front and back buffers are available on a system. Otherwise, the returned value is GL_FALSE.

For a continuous animation, we can also use

glutIdleFunc (animationFcn);

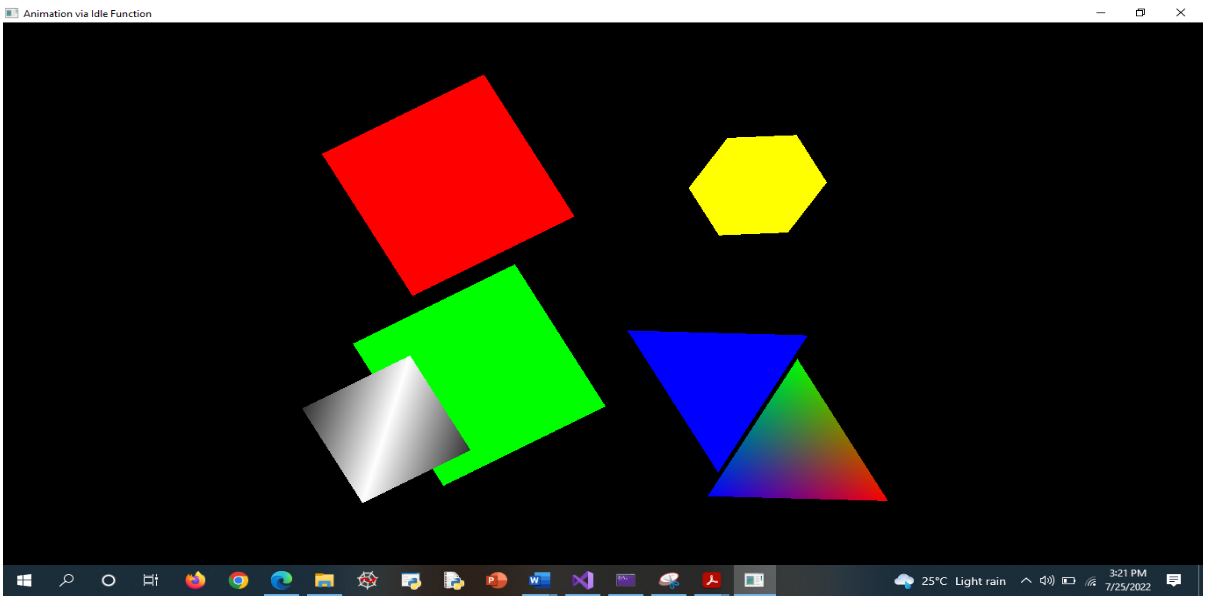
This procedure is continuously executed whenever there are no display-window events that must be processed. To disable the glutIdleFunc, we set its argument to the value NULL or the value 0.

Simple Animation using IDLE & Swap buffers

```
#include <windows.h>  // for MS Windows
#include <GL/glut.h>  // GLUT, include glu.h and gl.h

 // Global variable
GLfloat angle = 0.0f;  // Current rotational angle of the shapes

/* Initialize OpenGL Graphics */
void initGL() {
   // Set "clearing" or background color
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Black and opaque
}

/* Called back when there is no other event to be handled */
void idle() {
   glutPostRedisplay();   // Post a re-paint request to activate display()
}

/* Handler for window-repaint event. Call back when the window first appears and
  whenever the window needs to be re-painted. */
void display() {
   glClear(GL_COLOR_BUFFER_BIT);   // Clear the color buffer
   glMatrixMode(GL_MODELVIEW);     // To operate on Model-View matrix
   glLoadIdentity();               // Reset the model-view matrix

   glPushMatrix();                     // Save model-view matrix setting
   glTranslatef(-0.5f, 0.4f, 0.0f);    // Translate
   glRotatef(angle, 0.0f, 0.0f, 1.0f); // rotate by angle in degrees
   glBegin(GL_QUADS);                  // Each set of 4 vertices form a quad
   glColor3f(1.0f, 0.0f, 0.0f);     // Red
   glVertex2f(-0.3f, -0.3f);
   glVertex2f(0.3f, -0.3f);
```

```
glVertex2f(0.3f, 0.3f);
glVertex2f(-0.3f, 0.3f);
glEnd();
glPopMatrix();                  // Restore the model-view matrix

glPushMatrix();                 // Save model-view matrix setting
glTranslatef(-0.4f, -0.3f, 0.0f);   // Translate
glRotatef(angle, 0.0f, 0.0f, 1.0f); // rotate by angle in degrees
glBegin(GL_QUADS);
glColor3f(0.0f, 1.0f, 0.0f); // Green
glVertex2f(-0.3f, -0.3f);
glVertex2f(0.3f, -0.3f);
glVertex2f(0.3f, 0.3f);
glVertex2f(-0.3f, 0.3f);
glEnd();
glPopMatrix();                  // Restore the model-view matrix

glPushMatrix();                 // Save model-view matrix setting
glTranslatef(-0.7f, -0.5f, 0.0f);   // Translate
glRotatef(angle, 0.0f, 0.0f, 1.0f); // rotate by angle in degrees
glBegin(GL_QUADS);
glColor3f(0.2f, 0.2f, 0.2f); // Dark Gray
glVertex2f(-0.2f, -0.2f);
glColor3f(1.0f, 1.0f, 1.0f); // White
glVertex2f(0.2f, -0.2f);
glColor3f(0.2f, 0.2f, 0.2f); // Dark Gray
glVertex2f(0.2f, 0.2f);
glColor3f(1.0f, 1.0f, 1.0f); // White
glVertex2f(-0.2f, 0.2f);
glEnd();
glPopMatrix();                  // Restore the model-view matrix

glPushMatrix();                 // Save model-view matrix setting
glTranslatef(0.4f, -0.3f, 0.0f);    // Translate
glRotatef(angle, 0.0f, 0.0f, 1.0f); // rotate by angle in degrees
glBegin(GL_TRIANGLES);
glColor3f(0.0f, 0.0f, 1.0f); // Blue
glVertex2f(-0.3f, -0.2f);
glVertex2f(0.3f, -0.2f);
glVertex2f(0.0f, 0.3f);
glEnd();
glPopMatrix();                  // Restore the model-view matrix

glPushMatrix();                 // Save model-view matrix setting
glTranslatef(0.6f, -0.6f, 0.0f);    // Translate
glRotatef(180.0f + angle, 0.0f, 0.0f, 1.0f); // Rotate 180+angle degree
glBegin(GL_TRIANGLES);
glColor3f(1.0f, 0.0f, 0.0f); // Red
glVertex2f(-0.3f, -0.2f);
```

```
   glColor3f(0.0f, 1.0f, 0.0f); // Green
   glVertex2f(0.3f, -0.2f);
   glColor3f(0.0f, 0.0f, 1.0f); // Blue
   glVertex2f(0.0f, 0.3f);
   glEnd();
   glPopMatrix();                    // Restore the model-view matrix

   glPushMatrix();                   // Save model-view matrix setting
   glTranslatef(0.5f, 0.4f, 0.0f);     // Translate
   glRotatef(angle, 0.0f, 0.0f, 1.0f); // rotate by angle in degrees
   glBegin(GL_POLYGON);
   glColor3f(1.0f, 1.0f, 0.0f); // Yellow
   glVertex2f(-0.1f, -0.2f);
   glVertex2f(0.1f, -0.2f);
   glVertex2f(0.2f, 0.0f);
   glVertex2f(0.1f, 0.2f);
   glVertex2f(-0.1f, 0.2f);
   glVertex2f(-0.2f, 0.0f);
   glEnd();
   glPopMatrix();                    // Restore the model-view matrix
   glutSwapBuffers();   // Double buffered - swap the front and back buffers

   // Change the rotational angle after each display()
   angle += 0.2f;
}

/* Handler for window re-size event. Called back when the window first appears and
whenever the window is re-sized with its new width and height */
void reshape(GLsizei width, GLsizei height) {  // GLsizei for non-negative integer
  // Compute aspect ratio of the new window
   if (height == 0) height = 1;                // To prevent divide by 0
   GLfloat aspect = (GLfloat)width / (GLfloat)height;

   // Set the viewport to cover the new window
   glViewport(0, 0, width, height);

   // Set the aspect ratio of the clipping area to match the viewport
   glMatrixMode(GL_PROJECTION);  // To operate on the Projection matrix
   glLoadIdentity();
   if (width >= height) {
      // aspect >= 1, set the height from -1 to 1, with larger width
      gluOrtho2D(-1.0 * aspect, 1.0 * aspect, -1.0, 1.0);
   }
   else {
      // aspect < 1, set the width to -1 to 1, with larger height
      gluOrtho2D(-1.0, 1.0, -1.0 / aspect, 1.0 / aspect);
   }
}
int main(int argc, char** argv) {
   glutInit(&argc, argv);          // Initialize GLUT
   glutInitDisplayMode(GLUT_DOUBLE);  // Enable double buffered mode
   glutInitWindowSize(640, 480);   // Set the window's initial width & height - non-square
   glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
   glutCreateWindow("Animation via Idle Function");  // Create window with the given title
   glutDisplayFunc(display);       // Register callback handler for window re-paint event
   glutReshapeFunc(reshape);       // Register callback handler for window re-size event
   glutIdleFunc(idle);             // Register callback handler if no other event
   initGL();                       // Our own OpenGL initialization
   glutMainLoop();                 // Enter the infinite event-processing loop
   return 0;
}
```



Summary of OpenGL Animation Functions

Function	Description
`glutInitDisplayMode (GLUT_DOUBLE)`	Activates double-buffering operations.
`glutSwapBuffers`	Interchanges front and back refresh buffers.
`glGetBooleanv (GL_DOUBLEBUFFER, status)`	Queries a system to determine whether double buffering is available.
`glutIdleFunc`	Specifies a function for incrementing animation parameters.