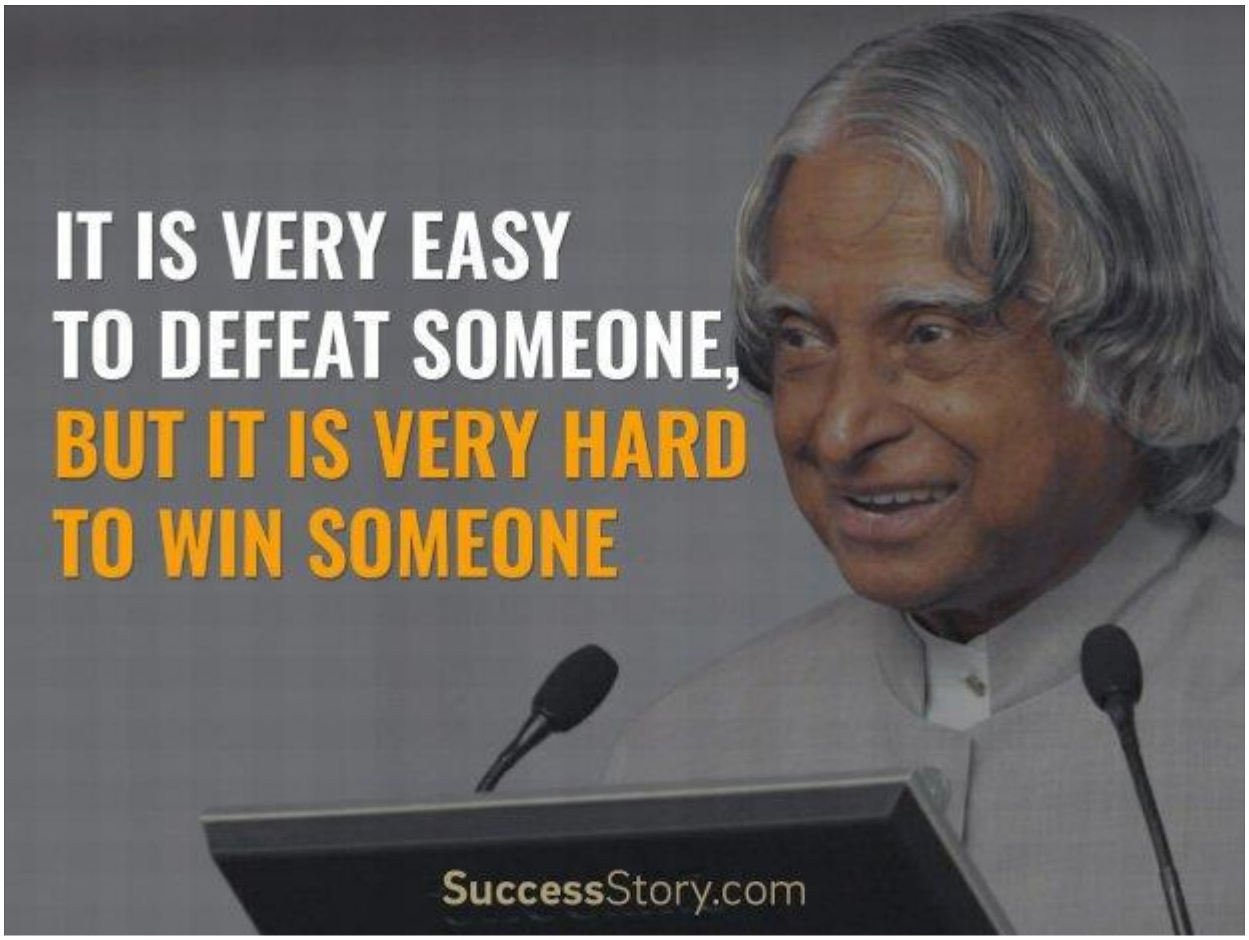
Acknowledgements to

Donald Hearn & Pauline Baker: Computer Graphics with OpenGL Version,3rd / 4th Edition, Pearson Education,2011

Edward Angel: Interactive Computer Graphics- A Top Down approach with OpenGL, 5th edition. Pearson Education, 2008

M M Raiker, Computer Graphics using OpenGL, Filip learning/Elsevier



5.1 INPUT AND INTERACTION

Interaction; Input devices;

Clients and servers; Display lists;

Display lists and modeling; Programming

event-driven input; Menus; Picking;

A simple CAD program;

Building interactive models;

Animating interactive programs;

Design of interactive programs;

Logic operations.

5.1.1INTERACTION

- 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.
- For e.g. Image varying in response to the input from the user.
- **OpenGL doesn't directl**y 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:

1. Physical input devices
2. 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:
- ***Key board devices*** like standard keyboard, flexible keyboard, handheld keyboard etc. These are used to provide character input like letters, numbers, symbols etc.
- ***Pointing devices*** like mouse, track ball, light pen etc. These are used to specify the position on the computer screen.

1. 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.

Figure: Computer Keyboard

2. MOUSE AND TRACKBALL: These are pointing devices used to specify the position. Mouse and trackball interacts 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.

Figure: Mouse

Figure: TrackBall

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:

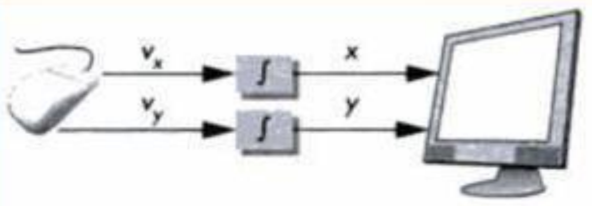


FIGURE Cursor positioning.

These devices are ***relative positioning devices*** because changes in the position of the ball yield a position in the user program.

3. <u>DATA TABLETS:</u> 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.

FIGURE Data tablet.

4. <u>LIGHT PEN:</u> 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.

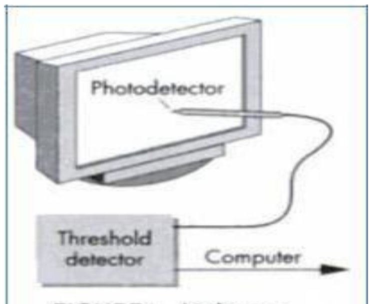


FIGURE : Lightpen.

The major disadvantage is that it has the difficulty in obtaining a position that corresponds to a dark area of the screen

5. <u>JOYSTICK:</u> The motion of the stick in two orthogonal directions is encoded, interpreted as two velocities and integrated to indentify 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. Thus, joystick is *variable sensitivity device*.

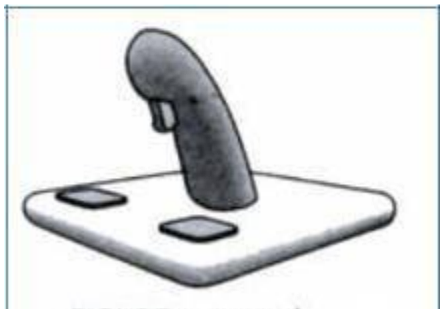


FIGURE : Joystick.

The advantage is that it is designed using mechanical elements such as springs and dampers which offer resistance to the user while pushing it. Such mechanical feel is suitable for application such as the flight simulators, game controllers etc.

6. <u>SPACE BALL:</u> It is a 3-Dimensional input device which looks like a joystick with a ball on the end of the stick.

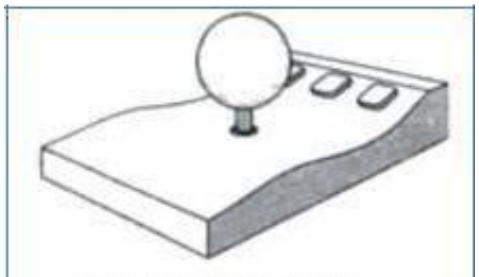


FIGURE Spaceball.

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.

Other 3-Dimensional devices such as laser scanners, measure 3-D positions directly. Numerous tracking systems used in virtual reality applications sense the position of the user and so on.

<u>LOGICAL INPUT DEVICES</u>

➔ These are characterized by its high-level interface with the application program rather than by its physical characteristics.

➔ 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. *The two major characteristics describe the logical behavior of input devices are as follows:*

- ***The measurements that the device returns to the user program***
- ***The time when the device returns those measurements***

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

1. **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.

2. **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.

3. **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.

4. **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.**

5. **VALUATORS:** They provide analog input to the user program on some graphical systems; there are boxes or dials to provide value.

6. **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.

INPUT MODES

Input devices can provide input to an application program in terms of two entities:

1. ***Measure*** of a device is what the device returns to the application program.
2. ***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.

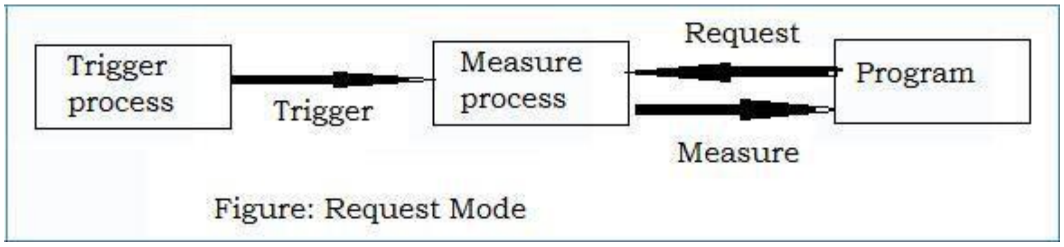


Figure: Request Mode

2. 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.

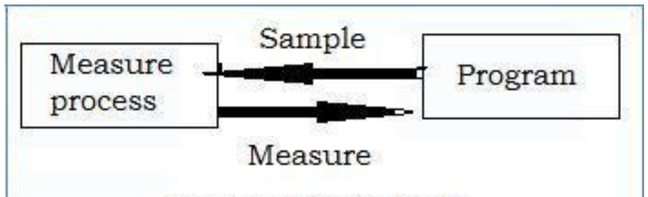


Figure: Sample Mode

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.

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.

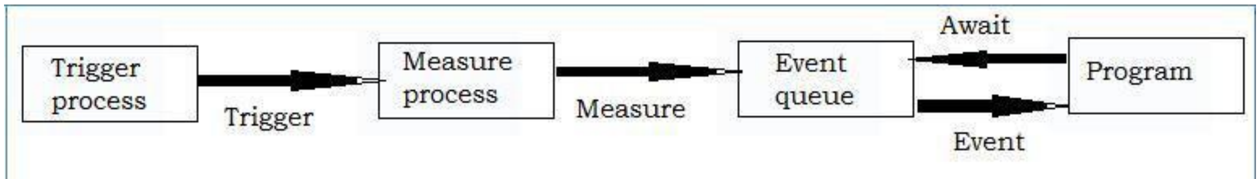


Figure: Event-mode model

Another approach is to associate a function when an event occurs, which is called as "call back."

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:

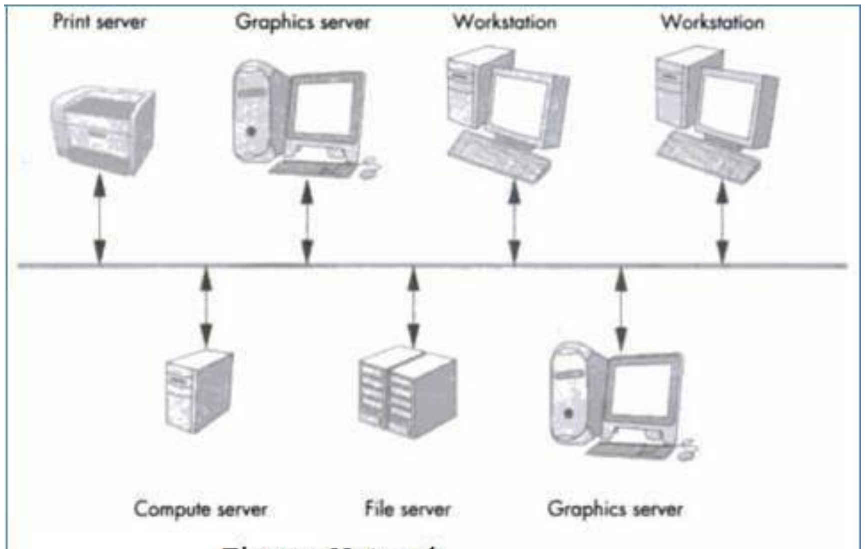


Figure: Network

- 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 architecture is shown in the next page.

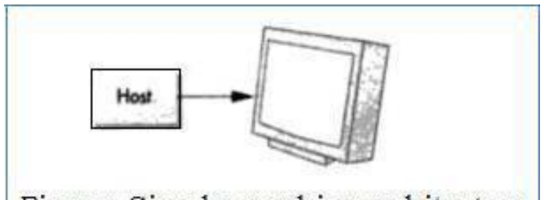


Figure: Simple graphics architecture

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"***.

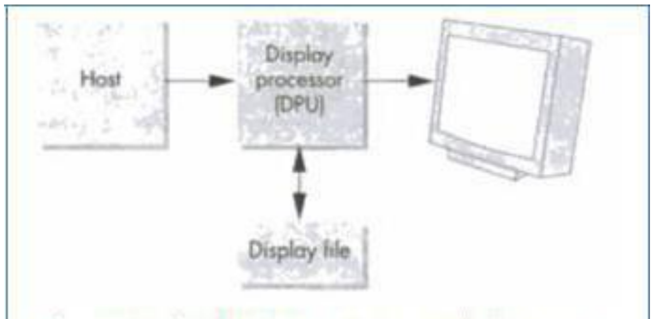


Figure: Display processor architecture

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.

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

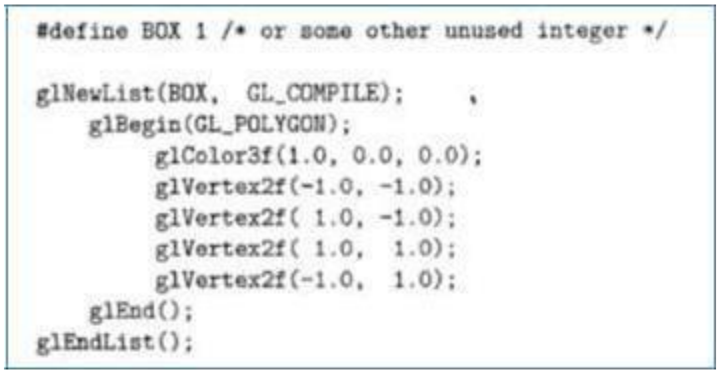
1. **<u>IMMEDIATE MODE:</u>** This mode sends the complete description of the object which needs to be drawn to the graphics server and no data can be retained. i.e., to redisplay the same object, the program must re-send the information. The information includes vertices, attributes, primitive types,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

2. <u>RETAINED MODE:</u> 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.

<u>DEFINITION AND EXECUTION OF DISPLAY LISTS</u>

- ➔ 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:*



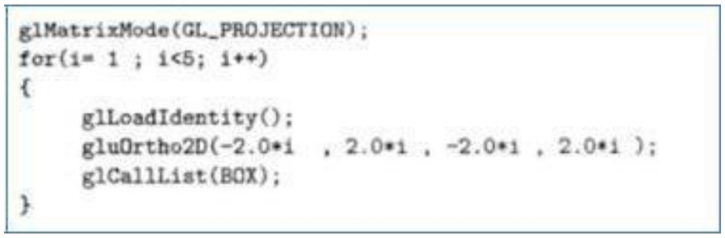
```
#define BOX 1 /* or some other unused integer */

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);
}
```

➔ OpenGL provides an API to retain the information by using ***stack* –** It is a data structure in which the item placed most recently is removed first [LIFO].

➔ *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();

➢ We can create multiple lists with consecutive identifiers more easily using:

glGenLists (number)

➢ We can display multiple display lists by using single funciton call:

glCallLists()

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.

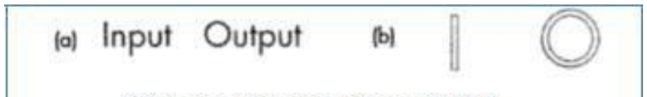
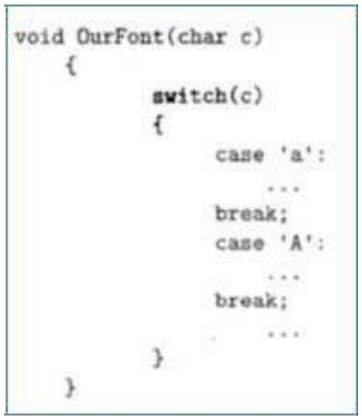


Figure: Stroke characters
(a) Filled strings (b) Magnified outlines

- From the above figure we can observe to draw letter 'I' is fairly simple, whereas drawing 'O' requires many line segments to get sufficiently smooth.
- So, on the average we need more than 13 bytes per character to represent stroke font. The performance of the graphics system will be degraded for the applications that require large quantity of text.
- 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'.
- The function may have the form



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

- For the character 'O' the code sequence is of the form as shown below:

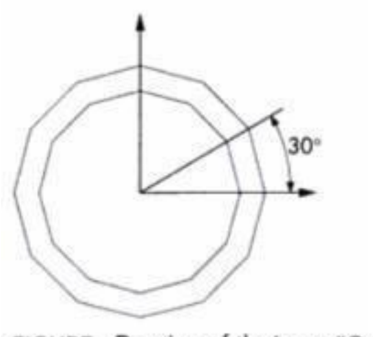
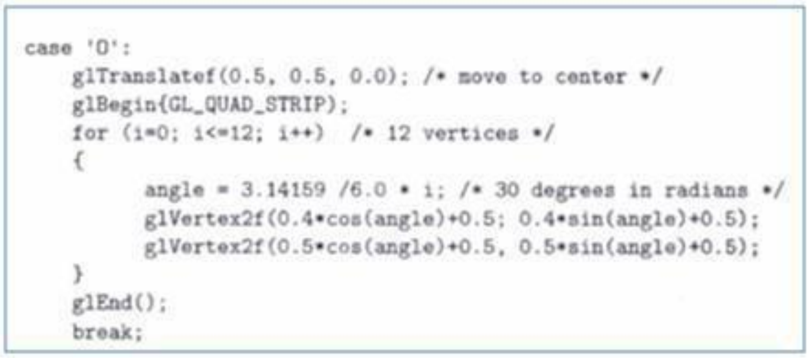


FIGURE Drawing of the letter "O."



```
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;
```

- The above code approximates the circle with 12 quadrilaterals.
- 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();
}
```

- To display char from the list, offset is set by using glListBase(base) function. The drawing of a string is accomplished in the server by the following function, char *text_string;

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

- The glCallLists has three arguments: (1) indicates number of lists to be executed (2) indicates the type (3) is a pointer to an array of a type given by second argument.

FONTS IN GLUT

- GLUT provides raster and stroke fonts; they do not make use of display lists.

- ***glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character);*** provides proportionally space characters. Position of a character is done by using a translation before the character function is called.
- ***glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character);*** produces the bitmap characters of size 8x13.

5.1.5 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:

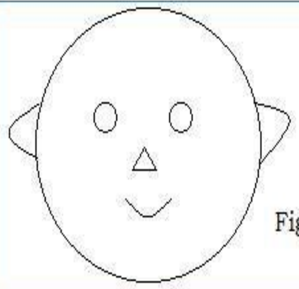
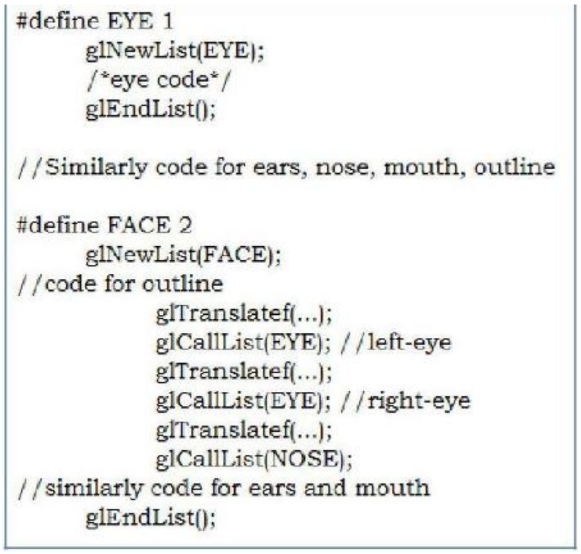


Figure: Simple animated face

- 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();
```

5.1.6 PROGRAMMING EVENT DRIVEN INPUT

- The various events can be recognized by the window system and call back function can be called for each of 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 event associated with pointing device, which is conventionally assumed to be mouse but could be trackball or data tablet also.

1. **MOVE EVENT –** is generated when the mouse is move with one of the button being pressed. If the mouse is moved without a button being pressed, this event is called as **"*passive move event*"**.
2. **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:

1. 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.

2. Register the defined mouse callback function in the main function, by means of GLUT function:
   glutMouseFunc(myMouse);

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()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glViewPort(0,0,w,h)
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0,(GLdouble) ww, 0, (GLdouble) wh);
        glMatrixMode(GL_MODELVIEW);
        glColor3f(1,0,0);
}

void drawsq (int x, int y)
{
        y=wh-y;
        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()
 {
        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)
                                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();
}
```

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:

1. 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.

2. The keyboard callback function must be registered in the main functionby means of GLUT function:
   glutKeyboardFunc(mykey);

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. Programming the window event involves two steps:

1. Window call back function must be defined in the form:

- ➔ ***void myReshape(GLsizei w, GLsizei h)*** is written by the application programmer.
- ➔ Let us consider drawing square as an example, the square of same size must be drawn regardless of window size.

```
void myReshape(GLsizei w, GLsizei h)
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0,(GLdouble) w, 0, (GLdouble) h);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glViewPort(0,0,w,h)
        /*save new window size in global  variables*/
         ww=w;
        wh=h;
}
```

2. The window callback function must be registered in the main function,

glutReshapeFunc(myReshape);

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)***.

WINDOW MANAGEMENT

- GLUT also supports multiple windows of a given window. We can create a second top-level window as follows:

  id = glutCreateWindow("second window");
- The returned integer value allows us to select this window as the current window.

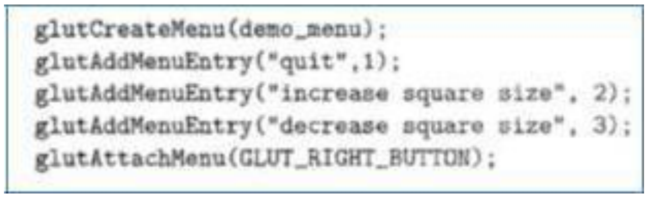
  i.e., glutSetWindow(id);

NOTE: The second window can have different properties from other window by invoking the glutInitDisplayMode before glutCreateWindow.

5.1.7 MENUS

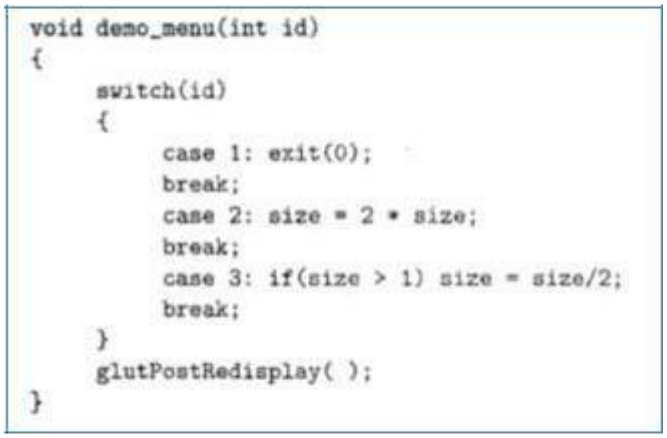
- Menus are an important feature of any application program. OpenGL provides a feature called ***"Pop-up-menus"*** using which sophisticated interactive applications can be created.
- Menu creation involves the following steps:

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.*



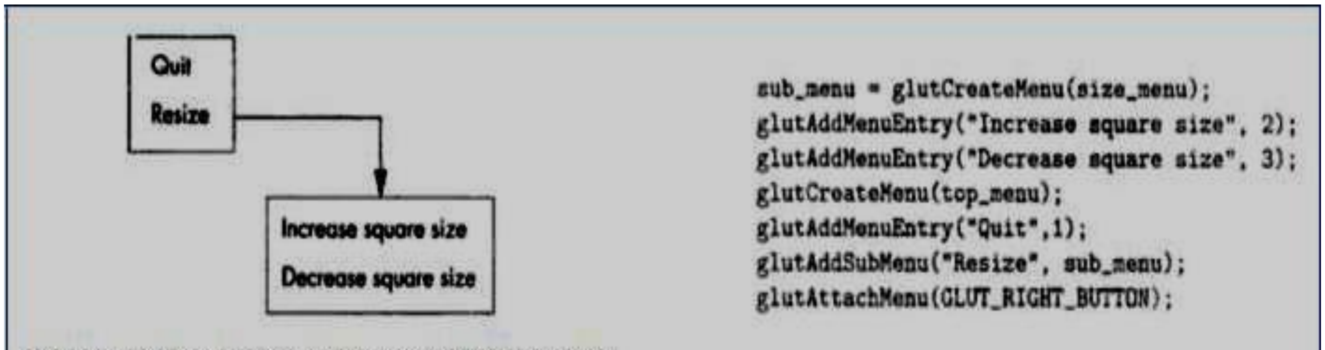
```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit",1);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

- The glutCreateMenu() registers the callback function demo_menu. The function glutAddMenuEntry() adds the entry in the menu whose name is pased 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 the creation of hierarchical menus which is given below:



```
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);
```

Figure: Structure of hierarchical menus

5.1.8 PICKING

- *Picking 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:***
        - In this approach, the extent of an object is the smallest rectangle aligned with the coordinate axis that contain object.
        - For 2D, it is relatively easy to determine the rectangle in screen coordinates that corresponds to rectangle point in object coordinates.
        - For 3D, bounding box is right parallelepiped. If application program maintains simple data structure to relate objects and bounding boxes, approximate picking can be done within application program.
    - ***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
- 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 difficulty in implementing the picking is we cannot go backward directly from the position of the mouse to the primitives.
- OpenGL provides **"**selection mode**"** to do picking. The glRenderMode() is used to choose select mode by passing GL_SELECT value.
- When we enter selection mode and render a scene, each primitive within the clipping volume generates a message called **"*hit*"** that is stored in a buffer called **"*name stack*"**.
- The following functions are used in selection mode:
  - ***void glSelectBuffer(GLsizei n, GLuint *buff) :*** specifies array **buffer of size 'n' in which to place selection data.**
  - ***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 x h area and centered at (x,y) in window coordinates within the viewport vp.

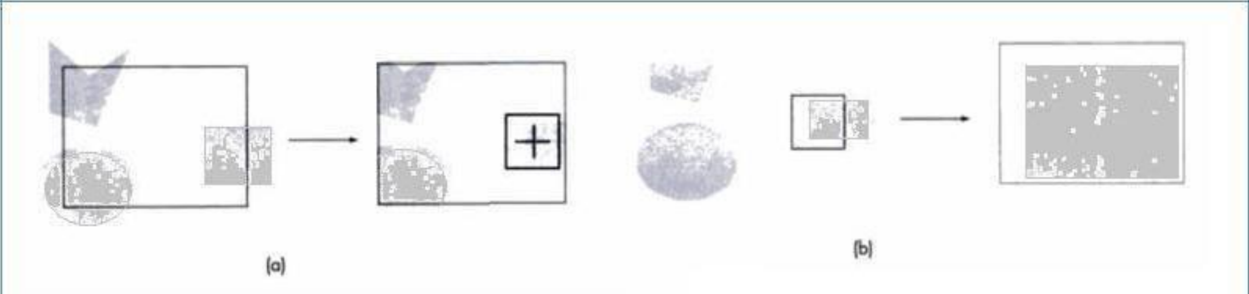


FIGURE Two-dimensional picking example. (a) Normal window and display. (b) Window and display after applying pick matrix.

(a) There is a normal window and image on the display. We also see the cursor with small box around it indicating the area in which primitive is rendered.

(b) It shows window and display after the window has been changed by gluPickMatrix.

The following code provides the implementation of picking process:

#include<glut.h>

```
void display( )
{
    glClear( GL_COLOR_BUFFER_BIT);
    draw_objects(GL_RENDER);
    glFlush();
}

 void drawObjects(GLenum mode)
 {
     if(mode == GL_SELECT) glLoadName(1);
     glColor3f(1.0, 0.0, 0.0);
     glRectf(-0.5, -0.5, 1.0, 1.0);

     if(mode == GL_SELECT) glLoadName(2);
     glColor3f(0.0, 0.0, 1.0);
     glRectf(-1.0, -1.0, 0.5, 0.5);
 }
```

```
#define N 2 /* N x N pixels around cursor for pick area */

 void mouse(int button, int state, int x, int y )
{
    GLuint nameBuffer[SIZE]; /* define SIZE elsewhere */
    GLint hits;
    GLint viewport[4];
    If( button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        /* initialize the name stack */

        glInitNames( );
        glPushName( 0 );
        glSelectBuffer( SIZE, nameBuffer);
       /* set up viewing for selection mode */

       glGetIntegerv(GL_VIEWPORT, viewport);
       glMatrixMode(GL_PROJECTION);

       /* save original viewing matrix */

       glPushMatrix();
       glLoadIdentity( );

       /* N x N pick area around cursor */
       /* must invert mouse y to get in world coords */
       gluPickMatrix( (GLdouble) x, (GLdouble)
           ( viewport[3] - y), N, N, viewport);

      /* same clipping window as in reshape callback */

      gluOrtho2D (xmin, xmax, ymin, ymax);

      draw_objects(GL_SELECT);
      glMatrixMode(GL_PROJECTION);

      /* restore viewing matrix */

      glPopMatrix();
      glFlush();

      /* return to normal render mode */

      hits = glRenderMode(GL_RENDER);

      /* process hits from selection mode rendering */

      processHits(hits, nameBuff);

      /* normal render */

      glutPostRedisplay();
   }
}
```

```
void myReshape()
{
        glViewPort(0,0,w,h)
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0,(GLdouble) w, 0, (GLdouble) h);
        glMatrixMode(GL_MODELVIEW);
}
```

```
void processHits (GLint hits, GLuint buffer[])
{
   unsigned int i, j;
   GLuint names, *ptr;

   printf ("hits = %d\n", hits);
   ptr = (GLuint *) buffer;

    /* loop over number of hits */

   for (i = 0; i < hits; i++)
    {
        names = *ptr;

        /* skip over number of names and depths */

        ptr += 3;

        /* check each name in record */

      for (j = 0; j < names; j++)
        {
         if(*ptr==1) printf ("red rectangle\n");
         else printf ("blue rectangle\n");

        /* go to next hit record */

         ptr++;
      }
   }
}
```

```
void main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE |GLUT_RGB);
```

```
        glutInitWindowSize(500,500);
        glutInitWindowPosition(100,100);
        glutCreateWindow("picking");
        glutReshapeFunc(myReshape);
        glutDisplayFunc(display);
        glutMouseFunc(Mouse);
        glClearColor(0.0,0.0,0.0,0.0);
        glutMainLoop();
    }
```

5.1.9 A SIMPLE CAD PROGRAM

Applications like interactive painting, design of mechanical parts and creating characters for a game are all examples of computer-aided design (CAD). CAD programs allow –

- The use of multiple windows and viewports to display a variety of information.
- The ability to create, delete and save user-defined objects.
- Multiple modes of operations employing menus, keyboard and mouse.

For example, consider the polygon-modeling CAD program which supports following operations:

1. Creation of polygons
2. Deletion of polygons
3. Selection and movement of polygons

Refer appendix A.5 polygon modeling program for the entire code from the prescribed text (Interactive Computer Graphics by Edward Angel 5th edition)

5.1.10 BUILDING INTERACTIVE MODELS

- Using OpenGL, we can develop a program where we can do insertion, manipulation, deletion etc and we can also build a program which is quite interactive by using the concept of instancing and display lists.
  - Consider an interior design application which has items like chairs, tables and other house hold items. These items are called the basic building blocks of the application. Each occurrence of these basic items is referred to as **"*instance*"**.
- Whenever the instances of building blocks are created by the user using the application program, the object (instance) is stored into an array called as **"*instance table*"**. We reserve the type 0 to specify that the object no longer exists (i.e., for deletion purpose)
- *Now suppose that the user has indicated through a menu that he wishes to eliminate an object and use the mouse to locate the object.*
  - o The program can now search the instance table till it finds the object as specified in the bounding box and then set its type to 0.
  - o Hence, next time when the display process goes through the instance table, the object would not be displayed and thereby it appears that object has been deleted.
- Although the above strategy works fine, a better data structure to implement the instance table is using linked lists instead of arrays.

5.1.11 ANIMATING INTERACTIVE PROGRAMS

- ➢ Using OpenGl, the programmer can design interactive programs. Programs in which objects are not static rather they appear to be moving or changing is considered as **"*Interactive programs*"**.
- ➢ Consider the following diagram:

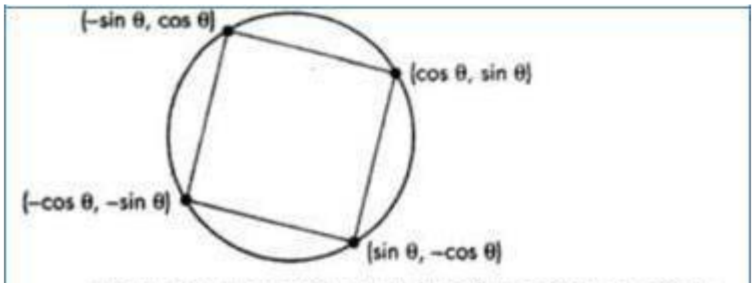


FIGURE Square constructed from four points on a circle.

- Consider a 2D point p(x,y) such that ***x = cos θ, y= sin θ***. This point would lie on a unit circle regardless of the value of θ. Thus, if we connect the above given four points we get a square which has its center as the origin. The above square can be displayed as shown below:

```
void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
    thetar = theta/(3.14159/180.0); /* convert degrees to radians */
    glVertex2f(cos(thetar), sin(thetar));
    glVertex2f(-sin(thetar), cos(thetar));
    glVertex2f(-cos(thetar), -sin(thetar));
    glVertex2f(sin(thetar), -cos(thetar));
  glEnd();
}
```

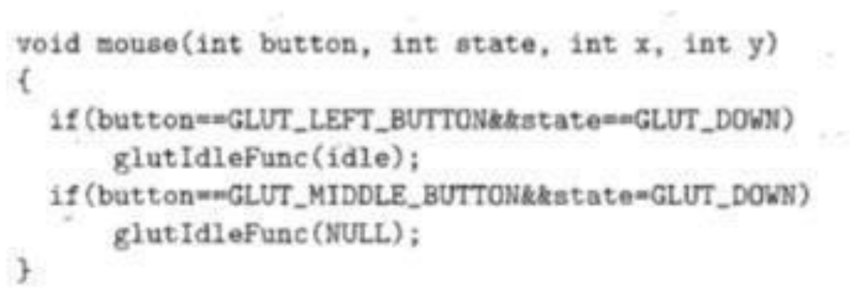
- Suppose that we change the value of θ as the program is running, the square appears to rotating about its origin. If the value of θ is to be changed by a fixed amount whenever nothing else is happening then an idle callback function must be designed as shown below:

```
void idle()
{
  theta+=2; /* or some other amount */
  if(theta >= 360.0 theta-=360.0;
  glutPostRedisplay();
}
```

- The above idle callback function must be registered in the main function:

  glutIdleFunc(idle);

- Suppose that we want to turn off and turn on the rotation feature then we can write a mouse callback function as shown below:



```
void mouse(int button, int state, int x, int y)
{
  if(button==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
      glutIdleFunc(idle);
  if(button==GLUT_MIDDLE_BUTTON&&state=GLUT_DOWN)
      glutIdleFunc(NULL);
}
```

- The above mouse callback function starts the rotation of the cube when the left mouse button and when the middle button is pressed it will halt the rotation.
- The above mouse callback function must be registered in the main function as follow:

  glutMouseFunc(mouse);

- However, when the above program is executed using single buffering scheme then flickering effect would be noticed on the display. This problem can be overcome using the concept of double buffering.

<u>DOUBLE BUFFERING:</u>

- Double buffering is a must in such animations where the primitives, attributes and viewing conditions are changing continuously.
- Double buffer consists of two buffers: front buffers and back buffers. Double buffering mode can be initialized:

  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);

- Further in the display function, we have to include: ***glutSwapBuffers()*** to exchange the contents of front and the back buffer.
- Using this approach, the problems associated with flicker can be eliminated.

USING TIMER:

- To understand the usage of timer, consider cube rotation program and its execution is done by using fast GPU (modern GPUs can render tens of millions of primitives per second) then cube will be rendered thousands of time per second and we will see the blur on the display.
- Therefore, GLUT provides the following timer function:

glutTimerFunc(int delay, void(*timer_func (int), int value)

- Execution of this function starts timer in the event loop that delays for *delay* milliseconds. When *timer* has counted down, *timer_func* is executed the *value* parameter allow user to pass variable into the timer call back.

5.1.12 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.

TOOLKITS, WIDGETS AND FRAME BUFFER:

The following two examples illustrate the limitations of geometric rendering.

1. ***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.

2. ***Rubberbanding:*** It is a technique used to define the elastic nature of pointing device to draw primitives.

➔ Consider paint application, if we want to draw a line, we indicate only two end points of our desired line segment. i.e., after locating first point, as we move the mouse, a line segment is drawn automatically [is updated on each refresh] from first location to present position of mouse.

➔ ***Rubberbanding*** begin when mouse button is pressed and continue until button is released at that time final line segment is drawn.

➔ We cannot implement this sequence of operations using only what we have presented so for. We will explore it in next chapters.

5.1.13 LOGIC OPERATIONS

Two types of functions that define writing modes are:

1. Replacement mode 2. Exclusive OR (XOR)

- When program specifies about visible primitive then OpenGL renders it into set of color pixels and stores it in the present drawing buffer.
  - In case of default mode, consider we start with a color buffer then has been cleared to black. Later we draw a blue color rectangle of size 10 x10 pixels then 100 blue pixels are copied into the color buffer, replacing 100 black pixels. Therefore, this mode is called as **"*copy or replacement mode*"**.
- Consider the below model, where we are writing single pixel into color buffer.

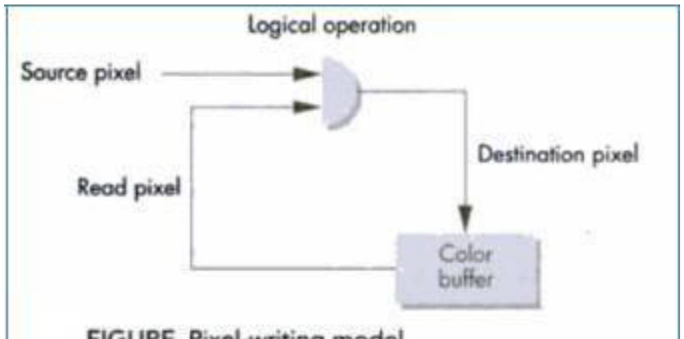


FIGURE Pixel-writing model.

- The pixel that we want to write is called as **“*source pixel*”**.
- The pixel in the drawing buffer which gets replaced by source pixel is called as **“*destination pixel*”**.
- In ***Exclusive-OR or (XOR) mode***, corresponding bits in each pixel are combing using XOR logical operation.
- If s and d are corresponding bits in the source and destination pixels, we **can denote the new destination bit as d’. d’ = d** $\oplus$ s
- One special property of XOR operation is if we apply it twice, it returns to the original state, it returns to the original state. So, if we draw some thing in XOR mode, we can erase it by drawing it again.

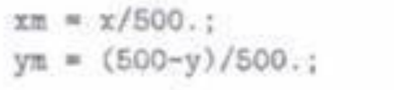
$$\mathbf{d = (d \oplus s) \oplus s}$$

- OpenGL supports all 16 logic modes, copy mode (GL_COPY) is the default. To change mode, we must enable logic operation, glEnable(GL_COLOR_LOGIC_OP) and then it can change to XOR mode glLogicOp(GL_XOR)

<u>DRAWING ERASABLE LINES</u>

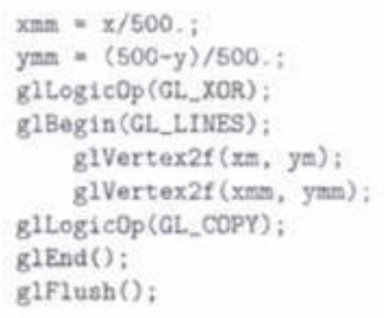
One way to draw erasable lines is given below:

- Mouse is used to get first end point and store this in object coordinates.



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

- Again mouse is used to get 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();
```

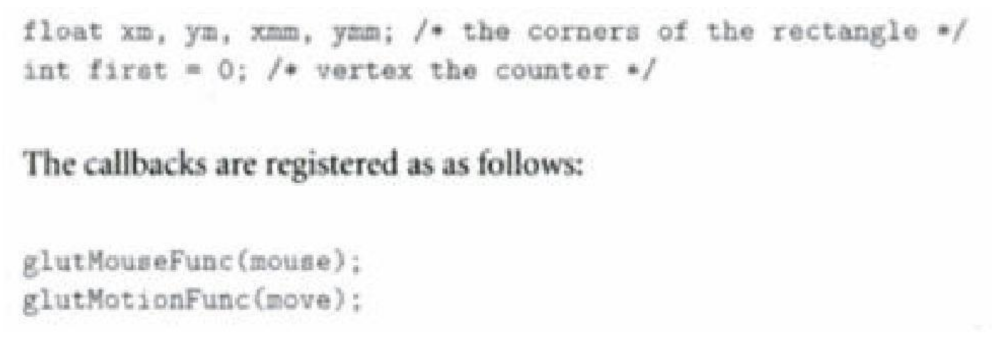
- Here in the above code, copy mode is used to switch back in order to draw other objects in normal mode.
    - If we enter another point with mouse, we first draw line in XOR mode from 1st point to 2nd point and draw second line using 1st point to current point is as follows:

```
glLogicOp(GL_XOR);
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glEnd();
glFlush();
xmm = x/500.0;
ymm = (500-y)/500.0;
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glEnd();
glLogicOp(GL_COPY);
glFlush();
```

Final form of code can be written as shown below:

```
glLogicOp(GL_COPY);
glBegin(GL_LINES);
    glVertex2f(xm, ym);
    glVertex2f(xmm, ymm);
glEnd();
glFlush();
glLogicOp(GL_XOR);
```

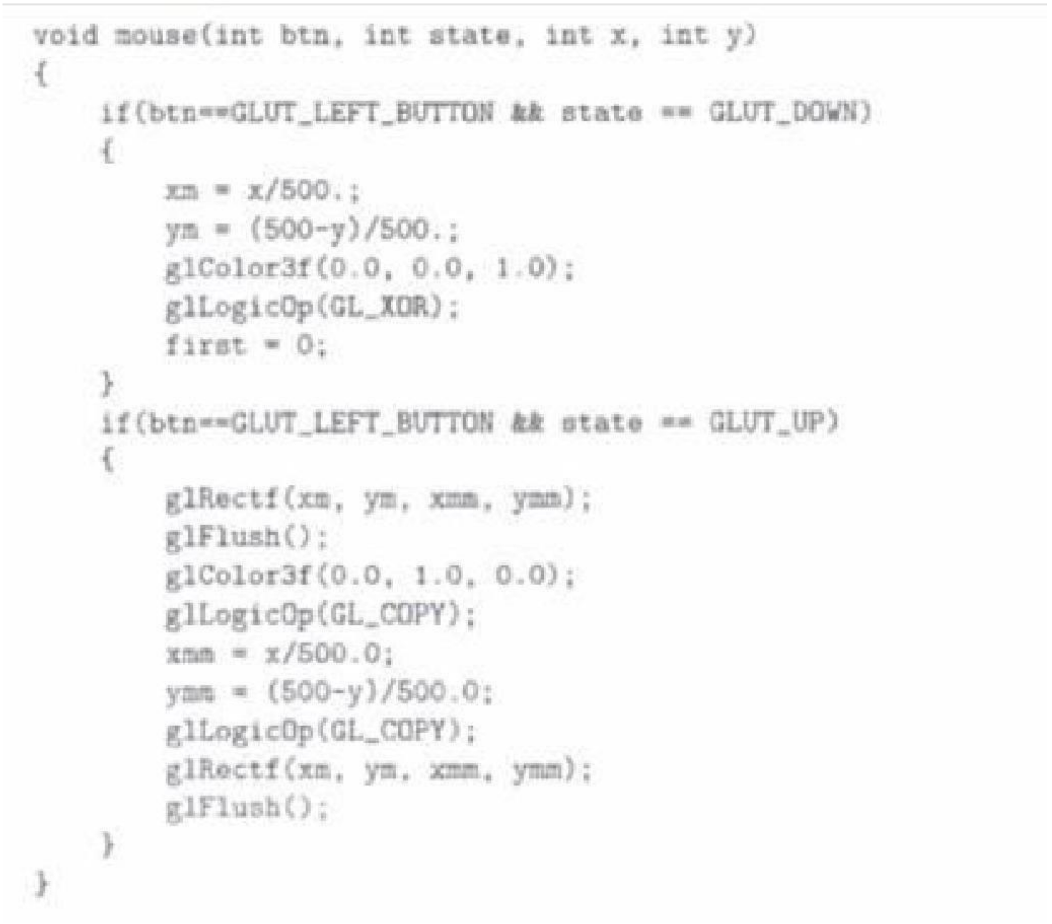
In this example, we draw rectangle using same concept and the code for callback function are given below:



```
float xm, ym, xmm, ymm; /* the corners of the rectangle */
int first = 0; /* vertex the counter */
```

The callbacks are registered as as follows:

```
glutMouseFunc(mouse);
glutMotionFunc(move);
```



```
void mouse(int btn, int state, int x, int y)
{
     if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
     {
          xm = x/500.;
          ym = (500-y)/500.;
          glColor3f(0.0, 0.0, 1.0);
          glLogicOp(GL_XOR);
          first = 0;
     }
     if(btn==GLUT_LEFT_BUTTON && state == GLUT_UP)
     {
          glRectf(xm, ym, xmm, ymm);
          glFlush();
          glColor3f(0.0, 1.0, 0.0);
          glLogicOp(GL_COPY);
          xmm = x/500.0;
          ymm = (500-y)/500.0;
          glLogicOp(GL_COPY);
          glRectf(xm, ym, xmm, ymm);
          glFlush();
     }
}
```

```
void move(int x, int y)
{
    if( first == 1)
    {
        glRectf(xm, ym, xmm, ymm);
        glFlush();
    }
    xmm = x/500.0;
    ymm = (500-y)/500.0;
    glRectf(xm, ym, xmm, ymm);
    glFlush();
    first = 1;
}
```

- For the first time, we draw a single rectangle in XOR mode.
- After that each time that we get vertex, we first erase the existing rectangle by redrawing new rectangle using new vertex.
- Finally, when mouse button is released the mouse callback is executed again which performs final erase and draw and go to replacement mode.

<u>XOR AND COLOR</u>

- ✓ Consider we would like to draw blue color line where 24 bit RGB values (00000000, 00000000, 11111111).
- ✓ Suppose the screen is clear to write (11111111, 11111111, 11111111) then when we draw blue line using XOR mode, then the resultant line would appear in yellow color (11111111, 11111111, 00000000) because XOR operation is applied bit-wise.
- ✓ This leads to form annoying visual effects.
- ✓ Therefore, we should use copy mode while drawing final output to get it in required color.

<u>CURSORS AND OVERLAY PLANES</u>

- Rubberbanding and cursors can place a significant burden on graphics system as they require the display to be updated constantly.
- Although XOR mode simplifies the process, it requires the system to read present destination pixels before computing new destination pixels.

- Alternative is to provide hardware support by providing extra bits in the color buffer by adding ***"overlay planes"***.

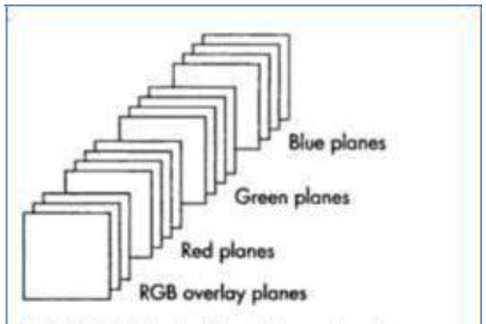


FIGURE Color buffer with overlay planes.

Therefore, typical color buffer may have 8 bits for each Red, green and blue and one red, one green and one blue overlay plane. i.e., each color will be having its own overlay plane then those values will be updated to color buffer.

FREQUENTLY ASKED QUESTIONS:

1. Explain all the available logical input devices in detail. (07M)
2. What is meant by measure and trigger of a device? Explain with the neat diagram, the various input mode models. (10M)
3. Explain the following: (i) Request Mode (ii)Sample Mode (iii)Event Mode (12M)
4. Differentiate event mode with request mode. (04M)
5. Describe logical input operation of picking in selection mode. (06M)
6. Write an OpenGL program to display square when a left button is pressed and to exit the program if right button is pressed. (10M)
7. What is display list? Give the OpenGL code segment that generates a display list defining a red triangle with vertices at (50,50) (150,50) and (100,150). (7M)
8. Using OpenGL functions, explain the structure of hierarchical menus. (06M)
9. List out the characteristics of good interactive program. Explain in detail. (08M)
10. What is double buffering? How OpenGL implements double buffering? (04M)
11. ***Write an OpenGL program on rotating or spinning a cube. (10M)***

<u>NOTE:</u> All the above questions are from previous year question papers. Do study the questions from other concepts also.

5.2 Curves:

5.2.1 Curved surfaces

- ➢ Sometimes 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. Curves and surfaces can be described by parameters
- ➢ **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:

    $f_1(x,y,z)=0$ or $z=f_2(x,y)$

- ✓ 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).

5.2.2 Quadric surfaces

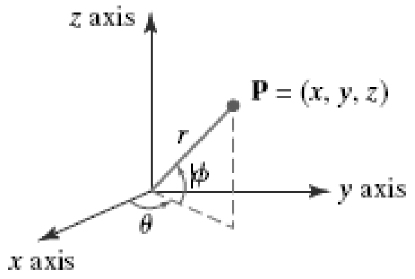
- ✓ A frequently used class of objects is the quadric surfaces, which are described with second - degree equations (quadratics).
- ✓ They include spheres, ellipsoids, tori, paraboloids, and hyperboloids.

1. <u>Sphere</u>

- ❖ 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

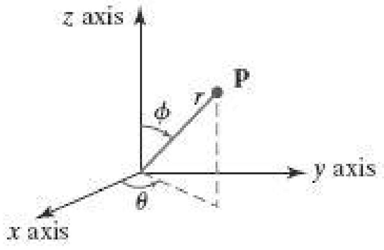
$$x^2 + y^2 + z^2 = r^2$$

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



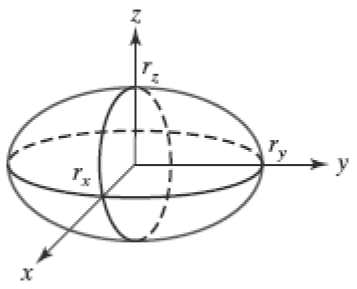
$$x = r\cos\varphi\cos\theta,\ -\pi/2 \le \varphi \le \pi/2$$
$$y = r\cos\varphi\sin\theta,\ -\pi \le \theta \le \pi$$
$$z = r\sin\varphi$$

- ❖ Alternatively, we could write the parametric equations using standard spherical coordinates, where angle φ is specified as the colatitudes as shown in figure



2. 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

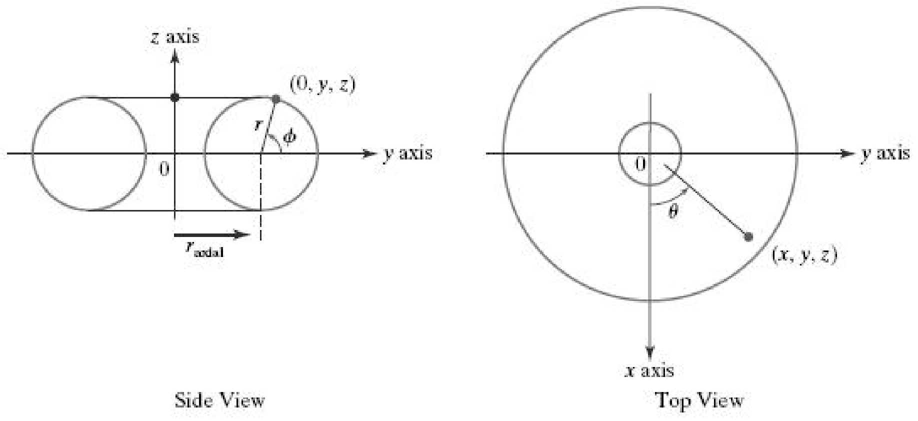
$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$

❖ And a parametric representation for the ellipsoid in terms of the latitude angle φ and the longitude angle θ

$$x = rx \cos\varphi \cos\theta, \; -\pi/2 \le \varphi \le \pi/2$$
$$y = ry \cos\varphi \sin\theta, \; -\pi \le \theta \le \pi$$
$$z = rz \sin\varphi$$

3. Torus

❖ A torus is a doughnut-shaped object, as shown in fig. below.



❖ It can be generated by rotating a circle or other conic about a specified axis.

❖ The equation for the cross-sectional circle shown in the side view is given by

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

❖ Rotating this circle about the *z* axis produces the torus whose surface positions are described with the Cartesian equation

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

❖ The corresponding parametric equations for the torus with a circular cross-section are

$$x=(r\text{axial}+r\cos\varphi)\cos\theta,\ -\pi\le\varphi\le\pi$$
$$y=(r\text{axial}+r\cos\varphi)\sin\theta,\ -\pi\le\theta\le\pi$$
$$z=r\sin\varphi$$

❖ We could also generate a torus by rotating an ellipse, instead of a circle, about the *z* axis. 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 r_{axial} 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 crosssection is

$$x=(r\text{axial}+ry\cos\varphi)\cos\theta,\ -\pi\le\varphi\le\pi$$
$$y=(r\text{axial}+ry\cos\varphi)\sin\theta,\ -\pi\le\theta\le\pi$$
$$z=rz\sin\varphi$$

5.2.3 OpenGL Quadric-Surface and Cubic-Surface Functions

➢ A number of 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

Function:

glutWireSphere (r, nLongitudes, nLatitudes);

or

glutSolidSphere (r, nLongitudes, nLatitudes);

where,

- r is sphere radius which is double precision point.
- nLongitudes and nLatitudes is number of longitude and latitude lines used to approximate the sphere.

Cone

Function:

glutWireCone (rBase, height, nLongitudes, nLatitudes);

or

glutSolidCone (rBase, height, nLongitudes, nLatitudes);

where,

- rBase is the radius of cone base which is double precision point.
- height is the height of cone which is double precision point.
- nLongitudes and nLatitudes are assigned integer values that specify the number of orthogonal surface lines for the quadrilateral mesh approximation.

Torus

Function:

glutWireTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);

or

glutSolidTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);

where,

- 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

Function:

glutWireTeapot (size);

or

glutSolidTeapot (size);

- ✓ The teapot surface is generated using OpenGL B´ezier 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.

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, and

  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.

- ❖ To produce a view of a cone, cylinder, or tapered cylinder, we replace the **gluSphere** function with

  gluCylinder (quadricName, rBase, rTop, height, nLongitudes, nLatitudes);

  - ✓ The base of this object is in the *xy* plane (*z*=0), and the axis is the *z* axis.
  - ✓ rBase is the radius at base and rTop is radius at top
  - ✓ If rTop=0.0,weget a cone; if rTop=rBase,weobtain a cylinder
  - ✓ Height is the height of the object and latitudes and longitude values will be given as nLatitude and nLongitude.

- ❖ A flat, circular ring or solid disk is displayed in the *xy* plane (*z*=0) and centered on the world-coordinate origin with

  gluDisk (ringName, rInner, rOuter, nRadii, nRings);

  - ✓ We set double-precision values for an inner radius and an outer radius with parameters **rInner** and **rOuter**. If **rInner** = 0, the disk is solid.
  - ✓ Otherwise, it is displayed with a concentric hole in the center of the disk.
  - ✓ The disk surface is divided into a set of facets with integer parameters **nRadii** and **nRings**

❖ We can specify a section of a circular ring with the following GLU function:

gluPartialDisk (ringName, rInner, rOuter, nRadii, nRings, startAngle, sweepAngle);

- ✓ **startAngle** designates an angular position in degrees in the *xy* plane measured clockwise from the positive *y* axis.
- ✓ parameter **sweepAngle** denotes an angular distance in degrees from the **startAngle** position.
- ✓ A section of a flat, circular disk is displayed from angular position **startAngle** to **startAngle + sweepAngle**
- ✓ For example, if **startAngle** = 0.0 and **sweepAngle** = 90.0, then the section of the disk lying in the first quadrant of the *xy* plane is displayed.

❖ Allocated memory for any GLU quadric surface can be reclaimed and the surface eliminated with

gluDeleteQuadric (quadricName);

❖ Also, we can define the front and back directions for any quadric surface with the following orientation function:

gluQuadricOrientation (quadricName, normalVectorDirection);

Where,

Parameter **normalVectorDirection** is assigned either **GLU_OUTSIDE** or **GLU_ INSIDE**

❖ Another option is the generation of surface-normal vectors, as follows:

gluQuadricNormals (quadricName, generationMode);

Where,

- ✓ A symbolic constant is assigned to parameter **generationMode** to indicate how surface-normal vectors should be generated. The default is **GLU_NONE.**
- ✓ For flat surface shading (a constant color value for each surface), we use the symbolic constant **GLU_FLAT**

- ✓ When other lighting and shading conditions are to be applied, we use the constant **GLU_SMOOTH**, which generates a normal vector for each surface vertex position.

- ❖ We can designate a function that is to be invoked if an error occurs during the generation of a quadric surface:

  gluQuadricCallback (quadricName, GLU_ERROR, function);

Example Program Using GLUT and GLU Quadric-Surface Functions

```
#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 ( );
}
```

5.2.4 Bézier Spline Curves

- ➢ It was developed by the French engineer Pierre Bézier for use in the design of Renault automobile bodies.
- ➢ **Bézier splines** have a number of properties that make them highly useful and convenient for curve and surface design. They are also easy to implement.
- ➢ In general, a Bézier curve section can be fitted to any number of control points, although some graphic packages limit the number of control points to four.

<u>Bézier Curve Equations</u>

- ✓ We first consider the general case of *n* + 1 control-point positions, denoted as $\mathbf{p}_k = (x_k, y_k, z_k)$, with *k* varying from 0 to *n*.
- ✓ These coordinate points are blended to produce the following position vector **P***(u)*, which describes the path of an approximating Bézier polynomial function between $\mathbf{p}_0$ and $\mathbf{p}_n$:

$$P(u) = \sum_{k=0}^{n} p_k \, \mathrm{BEZ}_{k,n}(u), \qquad 0 \le u \le 1$$

- ✓ The Bézier blending functions BEZ*k*,*n(u)* are the *Bernstein polynomials*

$$\mathrm{BEZ}_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

where parameters *C(n*, *k)* are the binomial coefficients

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

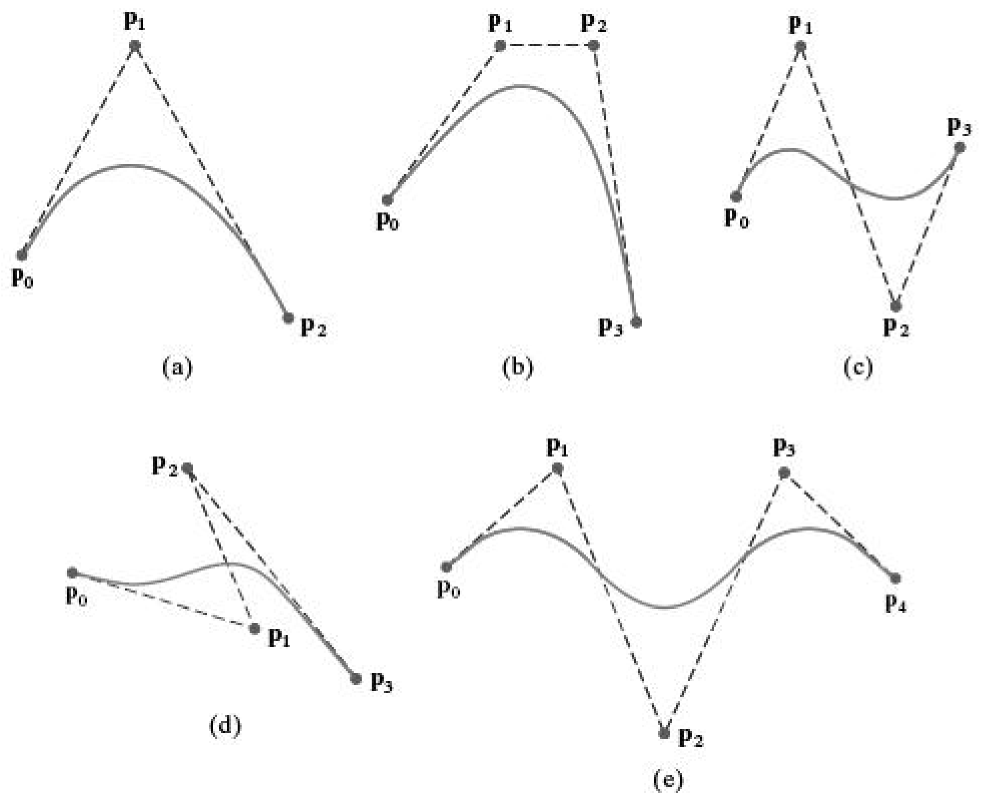
- ✓ A set of three parametric equations for the individual curve coordinates can be represented as

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

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

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

✓ Below Figure demonstrates the appearance of some Bézier curves for various selections of control points in the *xy* plane ($z = 0$).



✓ Recursive calculations can be used to obtain successive binomial-coefficient values as

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

for $n \geq k$. Also, the Bézier blending functions satisfy the recursive relationship

$BEZ_{k,n(u)} = (1 - u)BEZ_{k,n-1(u)} + u\ BEZ_{k-1,n-1(u)}),\ n > k \geq 1$ **(27)**

with $BEZ_{k,k} = uk$ and $BEZ_{0,k} = (1 - u)k$.

Program

```
#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 ( );
}
/* Compute binomial coefficients C for given value of n. */
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;
        /* Allocate space for binomial coefficients */
        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 = 4, 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 ( );
}
```

Properties of Bézier Curves

Property1:

- A very useful property of a Bézier curve is that the curve connects the first and last control points.
- Thus, a basic characteristic of any Bézier curve is that

$$\mathbf{P}(0) = \mathbf{p}0$$
$$\mathbf{P}(1) = \mathbf{p}n$$

- Values for the parametric first derivatives of a Bézier curve at the endpoints can be calculated from control-point coordinates as

$$\mathbf{P}'(0) = -n\mathbf{p}_0 + n\mathbf{p}_1$$
$$\mathbf{P}'(1) = -n\mathbf{p}_{n-1} + n\mathbf{p}_n$$

- The parametric second derivatives of a Bézier curve at the endpoints are calculated as

$$\mathbf{P}''(0) = n(n-1)[(\mathbf{p}_2 - \mathbf{p}_1) - (\mathbf{p}_1 - \mathbf{p}_0)]$$
$$\mathbf{P}''(1) = n(n-1)[(\mathbf{p}_{n-2} - \mathbf{p}_{n-1}) - (\mathbf{p}_{n-1} - \mathbf{p}_n)]$$

Property 2:

- Another important property of any Bézier curve is that it lies within the convex hull (convex polygon boundary) of the control points.
- This follows from the fact that the Bézier blending functions are all positive and their sum is always 1:

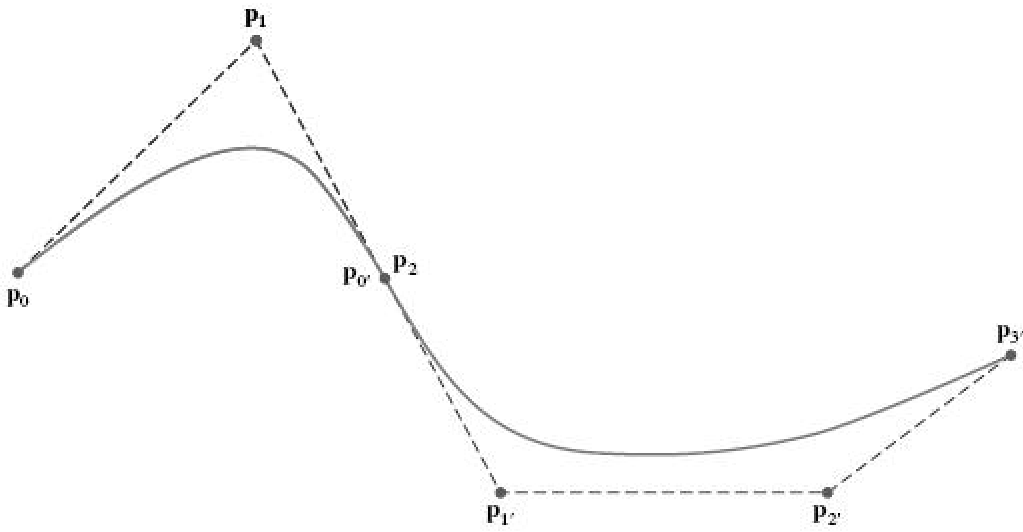
$$\sum_{k=0}^{n} \mathrm{BEZ}_{k,n}(u) = 1$$

Other Properties:

- ❖ The basic functions are real.
- ❖ The degree of the polynomial defining the curve segment is one less than the number of defining points.
- ❖ The curve generally follows the shape of the defining polygon,
- ❖ The tangent vectors at the ends of the curve have the same direction as the first and last polygon spans respectively.

Design Techniques Using Bézier Curves

- ✓ A closed Bézier curve is generated when we set the last control-point position to the coordinate position of the first control point.
- ✓ Specifying multiple control points at a single coordinate position gives more weight to that position a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position.
- ✓ When complicated curves are to be generated, they can be formed by piecing together several Bézier sections of lower degree.
- ✓ Generating smaller Bézier-curve sections also gives us better local control over the shape of the curve.
- ✓ Because Bézier curves connect the first and last control points, it is easy to match curve sections.
- ✓ Also,Bézier curves have the important property that the tangent to the curve at an endpoint is along the line joining that endpoint to the adjacent control point to obtain first-order continuity between curve sections, we can pick control points **p**0' and **p**1'for the next curve section to be along the same straight line as control points **p***n*−1 and **p***n* of the preceding section



- ✓ If the first curve section has *n* control points and the next curve section has *n*' control points, then we match curve tangents by placing control point **p**1' at the position

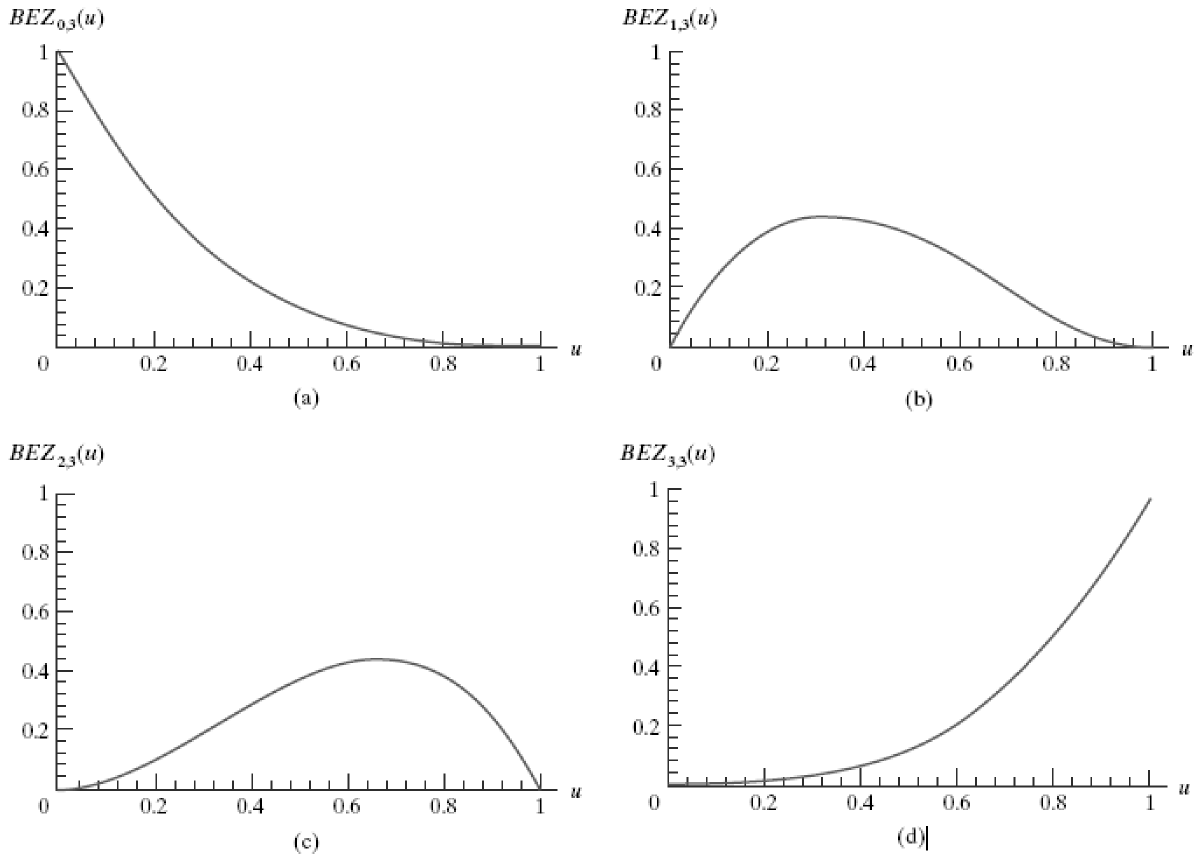
$$p_{1'} = p_n + \frac{n}{n'}(p_n - p_{n-1})$$

Cubic Bézier Curves

- ✓ Cubic Bézier curves are generated with four control points. The four blending functions for cubic Bézier curves, obtained by substituting $n = 3$ in the equations below, they are

$$
\begin{aligned}
\mathrm{BEZ}_{0,3} &= (1-u)^3\\
\mathrm{BEZ}_{1,3} &= 3u(1-u)^2\\
\mathrm{BEZ}_{2,3} &= 3u^2(1-u)\\
\mathrm{BEZ}_{3,3} &= u^3
\end{aligned}
$$

- ✓ Plots of the four cubic Bézier blending functions are given in Figure



- ✓ At the end positions of the cubic Bézier curve, the parametric first derivatives (slopes) are

$$\mathbf{P}'(0) = 3(\mathbf{p}_1 - \mathbf{p}_0), \qquad \mathbf{P}'(1) = 3(\mathbf{p}_3 - \mathbf{p}_2)$$

and the parametric second derivatives are

$$\mathbf{P}''(0) = 6(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2), \qquad \mathbf{P}''(1) = 6(\mathbf{p}_1 - 2\mathbf{p}_2 + \mathbf{p}_3)$$

- ✓ A matrix formulation for the cubic-Bézier curve function is obtained by expanding the polynomial expressions for the blending functions and restructuring the equations as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_{\text{Bez}} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix}$$

where the **Bézier matrix** is

$$\mathbf{M}_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

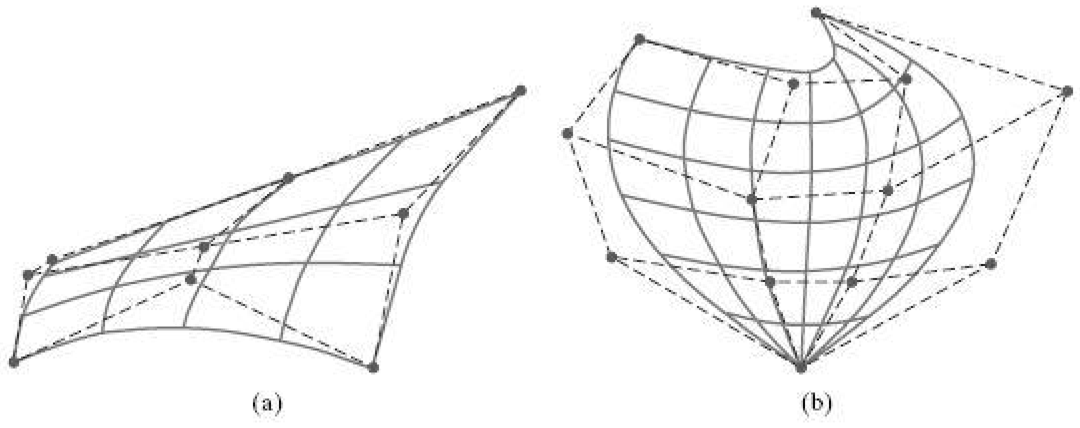
5.2.5 Bézier Surfaces

- ❖ The parametric vector function for the Bézier surface is formed as the tensor product of Bézier blending functions:

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

with $\mathbf{p}_{j,k}$ specifying the location of the $(m + 1)$ by $(n + 1)$ control points

- ❖ Figure below illustrates two Bézier surface plots. The control points are connected by dashed lines, and the solid lines show curves of constant u and constant v.
- ❖ Each curve of constant u is plotted by varying v over the interval from 0 to 1, with u fixed at one of the values in this unit interval. Curves of constant v are plotted similarly.

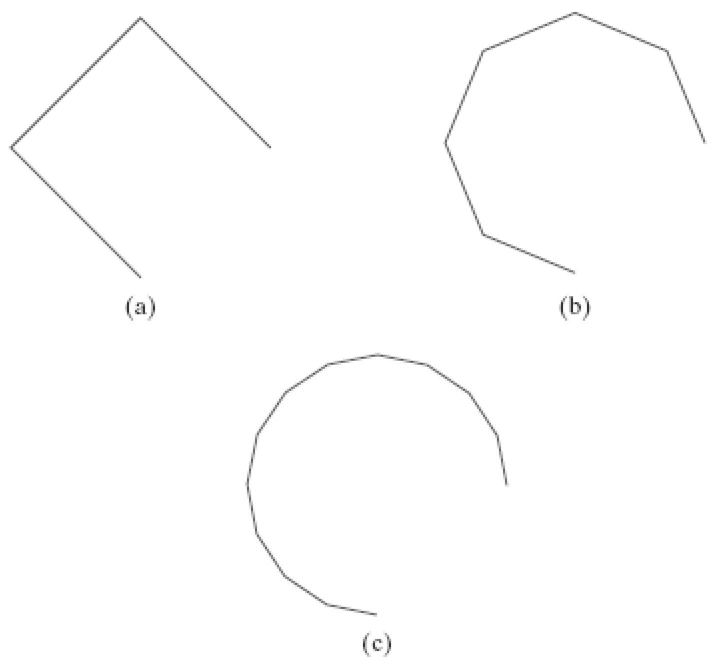


- ❖ Bézier surfaces have the same properties as Bézier curves, and they provide a convenient method for interactive design applications.
- ❖ To specify the threedimensional coordinate positions for the control points, we could first construct a rectangular grid in the *xy* "ground" plane.

- We then choose elevations above the ground plane at the grid intersections as the *z*-coordinate values for the control points.

5.2.6 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.



- Figure above illustrates various polyline displays that could be used for a circle segment.
- A third alternative is to write our own curve-generation functions based on the algorithms with respect to line drawing and circle drawing.

5.2.7 OpenGL Approximation-Spline Functions

OpenGL Bézier-Spline Curve Functions

- We specify parameters and activate the routines for Bézier-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. M
- inimum 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

$$u = \frac{u_{\text{value}} - u_{\text{min}}}{u_{\text{max}} - u_{\text{min}}}$$

which maps the **uValue** to the interval from 0 to 1.0.

- A spline curve is generated with evenly spaced parameter values, and OpenGL provides the following functions, which we can use to produce a set of uniformly spaced parameter values:

```
glMapGrid1* (n, u1, u2);
glEvalMesh1 (mode, n1, n2);
```

Where,

- The suffix code for **glMapGrid1** can be either **f** or **d**.
- Parameter **n** specifies the integer number of equal subdivisions over the range from **u1** to **u2**.
- Parameters **n1** and **n2** specify an integer range corresponding to **u1** and **u2**.
- Parameter **mode** is assigned either **GL POINT** or **GL LINE**, depending on whether we want to display the curve using discrete points (a dotted curve) or using straight-line segments
- In other words, with **mode = GL LINE**, the preceding OpenGL commands are equivalent to

```
glBegin (GL_LINE_STRIP);
    for (k = n1; k <= n2; k++)
        glEvalCoord1f (u1 + k * (u2 - u1) / n);
glEnd ( );
```

<u>OpenGL Bézier-Spline Surface Functions</u>

- Activation and parameter specification for the OpenGL Bézier-surface routines are accomplished with

```
glMap2* (GL_MAP2_VERTEX_3, uMin, uMax, uStride, nuPts, vMin,
         vMax, vStride, nvPts, *ctrlPts);
glEnable (GL_MAP2_VERTEX_3);
```

Where,

- A suffix code of **f** or **d** is used with **glMap2** to indicate either floating-point or double precision for the data values.
- For a surface, we specify minimum and maximum values for both parameter *u* and parameter *v*.

- ➔ If control points are to be specified using four-dimensional homogeneous coordinates, we use the symbolic constant **GL_MAP2_VERTEX_4** instead of **GL_MAP2_VERTEX_3**

- ➢ We deactivate the Bézier-surface routines with

  glDisable {GL_MAP2_VERTEX_3}
- ➢ Coordinate positions on the Bézier surface can be calculated with

  glEvalCoord2* (uValue, vValue);

  or

  glEvalCoord2*v (uvArray);

Where,

- ➔ Parameter **uValue** is assigned some value in the interval from **uMin** to **uMax**,
- ➔ Parameter **vValue** is assigned some value in the interval from **vMin** to **vMax**.

$$u = \frac{u\text{Value} - u\text{Min}}{u\text{Max} - u\text{Min}}, \qquad v = \frac{v\text{Value} - v\text{Min}}{v\text{Max} - v\text{Min}}$$

which maps each of uValue and vValue to the interval from 0 to 1.0

<u>GLU B-Spline Curve Functions</u>

- ✓ Although the GLU B-spline routines are referred to as NURBs functions, they can be used to generate B-splines that are neither nonuniform nor rational.
- ✓ The following statements illustrate the basic sequence of calls for displaying a B-spline curve:

```
GLUnurbsObj *curveName;
curveName = gluNewNurbsRenderer ( );
gluBeginCurve (curveName);
gluNurbsCurve   (curveName,   nknots,   *knotVector,   stride,   *ctrlPts,
                        degParam, GL_MAP1_VERTEX_3);
gluEndCurve (curveName);
```

- ✓ We eliminate a defined B-spline with

  gluDeleteNurbsRenderer (curveName);
- ✓ A B-spline curve is divided automatically into a number of sections and displayed as a polyline by theGLUroutines.

- ✓ However, a variety of B-spline rendering options can also be selected with repeated calls to the following GLU function:

  gluNurbsProperty (splineName, property, value);

<u>GLU B-Spline Surface Functions</u>

```
GLUnurbsObj *surfName
surfName = gluNewNurbsRenderer ( );
gluNurbsProperty (surfName, property1, value1);
gluNurbsProperty (surfName, property2, value2);
gluNurbsProperty (surfName, property3, value3);
...
gluBeginSurface (surfName);
gluNurbsSurface (surfName, nuKnots, uKnotVector, nvKnots, vKnotVector,
uStride, vStride, &ctrlPts [0][0][0], uDegParam, vDegParam,
GL_MAP2_VERTEX_3);
gluEndSurface (surfName);
```

- ✓ As an example of property setting, the following statements specify a wire-frame, triangularly tessellated display for a surface:

```
gluNurbsProperty (surfName, GLU_NURBS_MODE, GLU_NURBS_TESSELLATOR);
gluNurbsProperty (surfName, GLU_DISPLAY_MODE, GLU_OUTLINE_POLYGON);
```

- ✓ To determine the current value of a B-spline property, we use the following query function:

  gluGetNurbsProperty (splineName, property, value);
- ✓ When the property **GLU_AUTO_LOAD_MATRIX** is set to the value **GL_FALSE**, we invoke

  gluLoadSamplingMatrices (splineName, modelviewMat, projMat, viewport);

- ✓ Various events associated with spline objects are processed using

  gluNurbsCallback (splineName, event, fcn);

- ✓ Data values for the **gluNurbsCallback** function are supplied by

  gluNurbsCallbackData (splineName, dataValues);

<u>GLU Surface-Trimming Functions</u>

- ✓ A set of one or more two-dimensional trimming curves is specified for a B-spline surface with the following statements:

  gluBeginTrim (surfName);

  gluPwlCurve (surfName, nPts, *curvePts, stride, GLU_MAP1_TRIM_2);

  ...

  gluEndTrim (surfName);

Where,

- ➔ Parameter **surfName** is the name of the B-spline surface to be trimmed.
- ➔ A set of floating-point coordinates for the trimming curve is specified in array parameter **curvePts**, which contains **nPts** coordinate positions.
- ➔ An integer offset between successive coordinate positions is given in parameter **stride**

Summary of OpenGL Bezier Functions

Function	Description
`glMap1`	Specifies parameters for Bézier-curve display, color values, etc., and activate these routines using `glEnable`.
`glEvalCoord1`	Calculates a coordinate position for a Bézier curve.
`glMapGrid1`	Specifies the number of equally spaced subdivisions between two Bézier-curve parameters.
`glEvalMesh1`	Specifies the display mode and integer range for a Bézier-curve display.
`glMap2`	Specifies parameters for a Bézier-surface display, color values, etc., and activate these routines using `glEnable`.
`glEvalCoord2`	Calculates a coordinate position for a Bézier surface.
`glMapGrid2`	Specifies a two-dimensional grid of equally spaced subdivisions over a Bézier surface.
`glEvalMesh2`	Specifies the display mode and integer range for the two-dimensional Bézier-surface grid.

Summary of OpenGL B-Spline Functions

Function	Description
`gluNewNurbsRenderer`	Activates the GLU B-spline renderer for an object name that has been defined with the declaration `GLUnurbsObj *bsplineName`.
`gluBeginCurve`	Begins the assignment of parameter values for a specified B-spline curve with one or more sections.
`gluEndCurve`	Signals the end of the B-spline curve parameter specifications.
`gluNurbsCurve`	Specifies the parameter values for a named B-spline curve section.
`gluDeleteNurbsRenderer`	Eliminates a specified B-spline.
`gluNurbsProperty`	Specifies rendering options for a designated B-spline.
`gluGetNurbsProperty`	Determines the current value of a designated property for a particular B-spline.

Summary of OpenGL B-Spline Functions (*Continued*)

Function	Description
`gluBeginSurface`	Begins the assignment of parameter values for a specified B-spline surface with one or more sections.
`gluEndSurface`	Signals the end of the B-spline surface parameter specifications.
`gluNurbsSurface`	Specifies the parameter values for a named B-spline surface section.
`gluLoadSamplingMatrices`	Specifies viewing and geometric transformation matrices to be used in sampling and culling routines for a B-spline.
`gluNurbsCallback`	Specifies a callback function for a designated B-spline and associated event.
`gluNurbsCallbackData`	Specifies data values that are to be passed to the event callback function.
`gluBeginTrim`	Begins the assignment of trimming-curve parameter values for a B-spline surface.
`gluEndTrim`	Signals the end of the trimming curve parameter specifications.
`gluPwlCurve`	Specifies trimming-curve parameter values for a B-spline surface.

5.3 Animation

Introduction:

- To 'animate' is literally 'to give life to'.
- 'Animating' is moving something which can't move itself.
- Animation adds to graphics the dimension of time which vastly increases the amount of information which can be transmitted.
- **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.
- Two basic methods for constructing a motion sequence are
  1. **real-time animation** and
     - 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.
  2. **frame-by-frame animation**
     - For a 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 video 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.
- ➢ Because 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 is to employ two refresh buffers.
- ✓ We create a frame for the animation in one of the 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 frame rate is reduced to 20 frames 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.

<u>Generating Animations Using Raster Operations</u>

- ➢ 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, so 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.**
- ➢ Here 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.

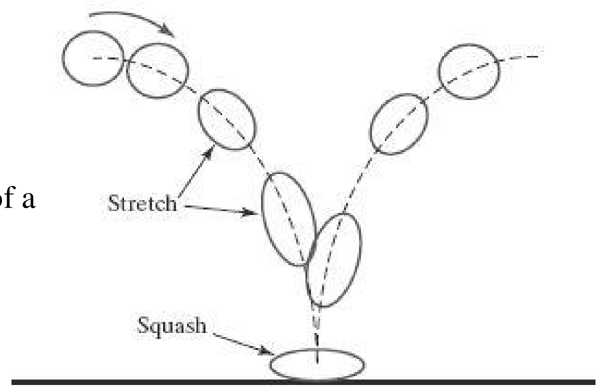
1. Storyboard layout
2. Object definitions.
3. Key-frame specifications
4. Generation of in-between frames.

- ✓ This approach of carrying out animations is applied to any other applications as well, although 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.
- ✓ And the intermediate frames between the key frames are In-betweens. And the 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. For a 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.
- ✓ A number of other tasks may be carried out depending upon the application requirement for example synchronization of a sound track.

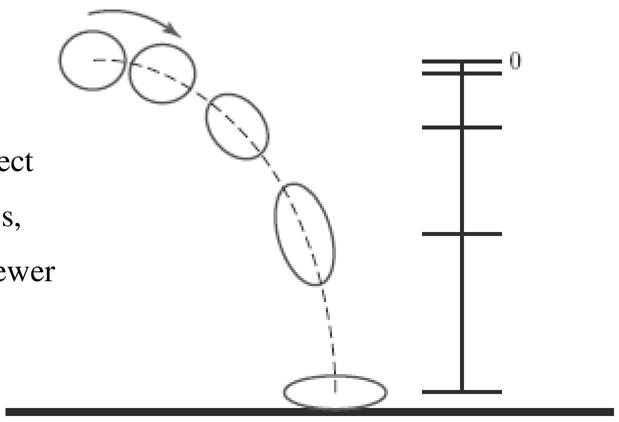
5.3.3 Traditional Animation Techniques

- ✓ 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.



- ✓ Object movements can also be emphasized by creating preliminary actions that indicate an **anticipation** of a coming motion

5.3.4 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.

5.3.5 OpenGL Animation Procedures

- Double-buffering operations, if available, 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.

Example Program

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

const double TWO_PI = 6.2831853;
GLsizei winWidth = 500, winHeight = 500; // Initial display window size.
GLuint regHex; // Define name for display list.
static GLfloat rotTheta = 0.0;

class scrPt {
        public:
                GLint x, y;
};
static void init (void)
{
        scrPt hexVertex;
        GLdouble hexTheta;
        GLint k;
        glClearColor (1.0, 1.0, 1.0, 0.0);
        /* Set up a display list for a red regular hexagon.
        * Vertices for the hexagon are six equally spaced
        * points around the circumference of a circle.
        */
        regHex = glGenLists (1);
        glNewList (regHex, GL_COMPILE);
        glColor3f (1.0, 0.0, 0.0);
        glBegin (GL_POLYGON);
        for (k = 0; k < 6; k++) {
                hexTheta = TWO_PI * k / 6;
```

```
                    hexVertex.x = 150 + 100 * cos (hexTheta);
                    hexVertex.y = 150 + 100 * sin (hexTheta);
                    glVertex2i (hexVertex.x, hexVertex.y);
            }
            glEnd ( );
            glEndList ( );
}
void displayHex (void)
{
            glClear (GL_COLOR_BUFFER_BIT);
            glPushMatrix ( );
            glRotatef (rotTheta, 0.0, 0.0, 1.0);
            glCallList (regHex);
            glPopMatrix ( );
            glutSwapBuffers ( );
            glFlush ( );
}
void rotateHex (void)
{
            rotTheta += 3.0;
            if (rotTheta > 360.0)
            rotTheta -= 360.0;
            glutPostRedisplay ( );
}
void winReshapeFcn (GLint newWidth, GLint newHeight)
{
            glViewport (0, 0, (GLsizei) newWidth, (GLsizei) newHeight);
            glMatrixMode (GL_PROJECTION);
            glLoadIdentity ( );
            gluOrtho2D (-320.0, 320.0, -320.0, 320.0);
            glMatrixMode (GL_MODELVIEW);
```

```
        glLoadIdentity ( );
        glClear (GL_COLOR_BUFFER_BIT);
}
void mouseFcn (GLint button, GLint action, GLint x, GLint y)
{
        switch (button) {
                case GLUT_MIDDLE_BUTTON: // Start the rotation.
                                                if (action == GLUT_DOWN)
                                                glutIdleFunc (rotateHex);
                                                break;
                case GLUT_RIGHT_BUTTON: // Stop the rotation.
                                                if (action == GLUT_DOWN)
                                                glutIdleFunc (NULL);
                                                break;
                default:
                                                break;
        }
}
void main(int argc, char ** argv)
{
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowPosition (150, 150);
        glutInitWindowSize (winWidth, winHeight);
        glutCreateWindow ("Animation Example");
        init ( );
        glutDisplayFunc (displayHex);
        glutReshapeFunc (winReshapeFcn);
        glutMouseFunc (mouseFcn);
        glutMainLoop ( );
}
```