Input and Interaction

INTERACTION

One of the most important advances in computer technology was enabling users to interact with computer displays.

eg: The user sees an image on the display ,user gives input through mouse ,the image changes the response.

Although rendering is the prime concern of most modern API, including openGL, interactivity is an important component of most applications. openGL however, does not support interaction directly.

This difficulty is overcome partially by using GLUT toolkit. This toolkit provides the minimal functionality, such as opening of windows, use of keyboard and mouse & creation of pop-up menus.

INPUT DEVICES:

We can think about input devices in two distinct ways

1) Physical Devices

Physical devices such as a keyboard or a mouse .From the perspective of an application programmer we do not need to know the details of a particular physical device to write an interaction program so the devices are referred with logical devices.

2) Logical devices

Logical devices whose properties are specified in terms of what they do from the perspective of the application program.

A logical device is characterized by its high-level interface with the application program rather than by its physical characteristics.

PHYSICAL INPUT DEVICES:

- From the physical perspective, each input devices has properties that make it more suitable for certain tasks than for others.
- There are two primary types of physical devices:
  - Pointing devices ,
  - Keyboard devices.
- The pointing devices allow the user to indicate a position on a display and almost always incorporates one or more buttons to allow the user to send to send signals and interrupts to the computer.
- The keyboard devices is almost always a physical keyboard but can generalized to include any devices that returns character codes.

Prepared By: Shatananda Bhat P

MOUSE AND TRACKBALL:

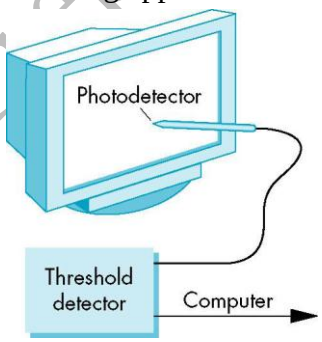
- The mouse and the trackball are similar in use and often in construction as well, when turned over a mechanical mouse looks like a track ball.
- In both, the mouse as well as the track ball the motion of ball is converted to signals sent back to the computer by pairs of encoders inside the devices.
- There are many variant of these devices such as optical detectors rather than the mechanical detectors to measure motion, small trackballs are popular with portable computers because they can be incorporated directly into the keyboard.
- Relative positioning of mouse or trackball is not always desirable, they are not suitable for operations such as tracing a diagram.

DATA TABLETS:

- Data tablets provide absolute positioning.
- A typical data tablet 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.
- Touch-sensitive transparent screens that can be placed over the face of a CRT have many of the same properties as the data tablet.
- These touchpads can be considered as a relative or absolute positioning devices, some are capable of multitouch, i.e. touch simultaneously with two fingers in different spots of the screen.

LIGHTPEN:

- The light pen has a long history in computer graphics
- The light pen contains a light-sensing device, such as a photocell
- If the lighten is positioned on the face of CRT at a location opposite where the electron beam strikes the phosphor, the light emitted exceeds s threshold in the photo detector and a signal is sent to the computer.
- The light pen was originally used on random scan devices so the time of interrupt could easily be matched to a piece of code in the display list, thus making the light pen ideal for selecting application-defined objects.



Prepared By: Shatananda Bhat P

JOYSTICK:

- The motion of the stick in two orthogonal directions is encoded, interpreted as two velocities and integrated to identify a screen location.
- The integration implies that is the stick is left in its resting position, there is no change in the cursor position and that the farther the stick is moved from its resting position, the faster the screen location changes.
- The advantage of this device is that the device can be constructed using the mechanical elements such as springs and clampers, that gives resistance to the user who is pushing the stick.
- The joystick is well suited for applications like flight simulator and as game controllers.

SPACEBALL:

- A spaceball looks like a joystick with a ball on the end of a stick
- The ball does not move
- The Pressure sensors in the ball measure the forces applied by the user
- The spaceball can measure not only the three direct forces (up-dowN, front-back, left-right) but also three independent twists.

Prepared By: Shatananda Bhat P

LOGICAL DEVICES:

Two major characteristics describes the logical behaviour of an input device:-

1. **The measurements that the device returns to the user program.**
2. **The time when the device returns those measurements.**

Six classes of input forms available to a developer of graphical applications

1. String:

- A string device is a logical device that provides ASCII strings to the user program
- It is usually implemented by means of a physical keyboard
- OpenGL usually do not distinguish between a logical string device and a physical keyboard

2. Locator:

- It provides a position in world coordinates to the user program
- It is usually implemented by means of a pointing device such as a mouse or a trackball
- In OpenGL we usually use the pointing device in this manner.

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 a locator, but has a separate software interface to the user program.
- In OpenGL we can use a process called selection to accomplish picking

4. Choice:

- Choice devices allow 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 device.
- Typical widgets include menus, scrollbars and graphical buttons.

5. Valuators:

- It provides analog input to the user program
- On some graphics systems, there are boxes or dials to provide valuator input
- Ex: Slidebar and radio boxes

6. Stroke:

- A stroke device returns an array of locations
- It is often implemented such that an action, say, pushing down a mouse button, starts the transfer of data into the specified array, and a second actions such as releasing the button, ends this transfer.

Prepared By: Shatananda Bhat P

INPUT MODES:

Input devices provide input to an application program can be described in terms of two entities:
1.A measure process
2.A device Trigger

- The **measure** of a device is what the device returns to the user program
- The **trigger** of a device is a physical input on the device with which the user can signal the computer

Ex: Measure: Position of the locator
Trigger: Button on a Physical Device

The application program obtain the measure of a device in three distinct modes:

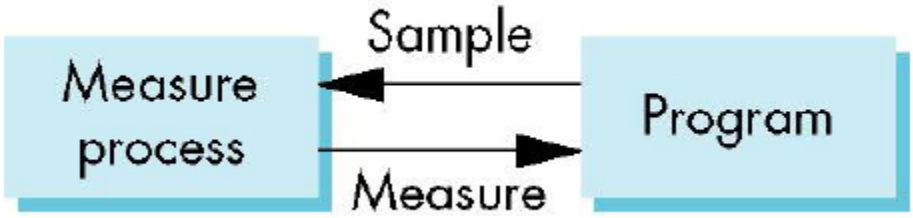
1. **Request Mode**
2. **Sample Mode**
3. **Event Mode**

1.Request Mode:

- In request mode the measure of the device is not returned to the program until the device is triggered.
- This input mode is standard in non-graphical applications. When the program needs the input, it halts when it encounters the ***scanf*** statement and waits while we type character at our terminal.
- The trigger will cause to the location to be returned to the application program.

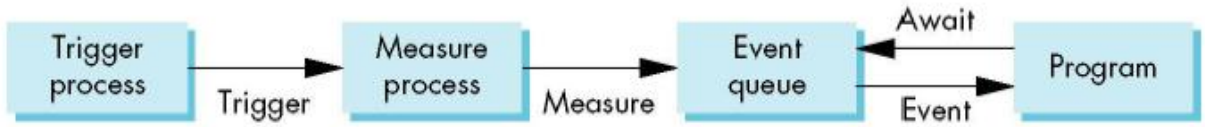
Prepared By: Shatananda Bhat P

2. Sample mode:



- Sample mode input is immediate.
- As soon as the function call in the user program is encountered, the measure returned. Hence no trigger is needed.
- One characteristic of both request and sample modes inputs in APIs that support them is that the user identify which device is to provide the input.
- Both request and sample modes are useful for situations where the program guides the user but not useful applications where the user controls the flow of the program.
- More generally request and sample modes are not sufficient for handling the variety of possible human computer interactions.

3. Event Mode:

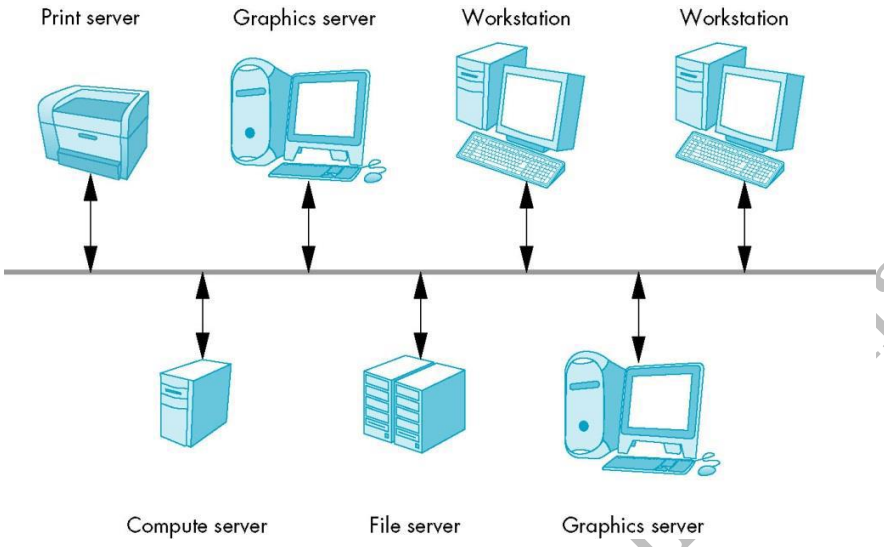


- Used in an environment with multiple input devices, each with its own trigger and each running a measure process.
- Each time that a device is triggered, an event is generated
- The device measure, including the identifier for the device is placed in an event queue.
- The process of placing events in the event queue is completely independent of what the application program does with these events.

Prepared by: Shatananda Bhat P

CLIENTS AND SERVERS:

Clients and servers can be distributed over a network or contained entirely in a single computational unit.

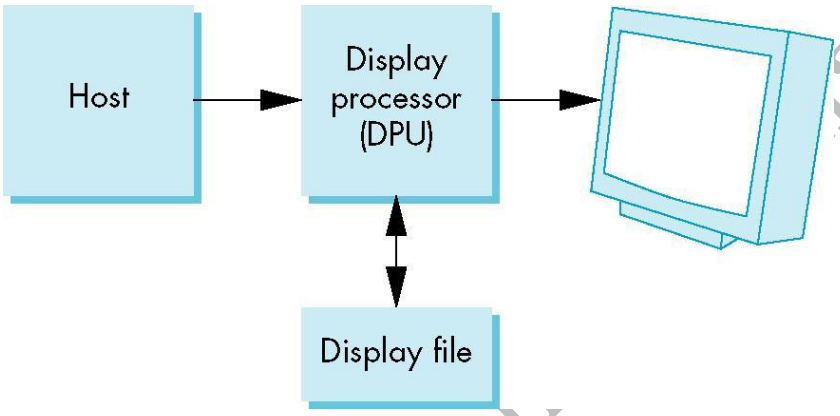


- User and user programs that make use of the services provided the servers are clients or client programs
- Servers can also exist at a lower level of granularity within a single operating system.
- A workstation can be both a client and a server
- A workstation may run client programs and server programs concurrently
- A workstation with a raster display, a keyboard and a pointing device such as a mouse is a graphics server
- The graphics server can provide output services on its display and input services through the keyboard and mouse
- These services are potentially available to the clients anywhere on the network
- Our OpenGL application programs are clients that use the graphics server.

Prepared By: Shatananda Bhat P

DISPLAY LISTS:

- Display lists illustrate how we can use clients and servers on a network to improve interactive graphics performance
- Early computers were slow and expensive, so the cost of keeping even a simple display refreshed was high
- The solution to this problem was to build a special purpose computer called a **display processor** with an organization



- The display processor had a limited instruction set (for drawing only primitives on display)
- The user program was processed in the host computer
- The resulting compiled list of instructions was then sent to the display processor
- The display processor stored these instructions in a display memory as a display file or display list.
- Today, the display processor has become a graphics server and the application program running on the host has become the client
- The major bottleneck is the amount of traffic that passes between the client and the server
- **Two modes to send graphical entities to the display:**
    1. **Immediate mode graphics**
    2. **Retained mode graphics**

1. Immediate mode graphics:

- As soon as the program executes a statement that defines a primitive, that primitive is sent to the server for possible display
- No memory of it is retained in the system
- To redisplay after clearing the screen, or in a new position after an interaction, the program must re-specify the primitive and them must resend the information through the display process
- This process can cause a considerable quantity of data to pass from the client to the server

Prepared By: Shatananda Bhat P

2. Retained mode graphics:

- We define the object once, and then put its description in a display list
- The display list is stored in the server and redisplayed by a simple function call issued form the client to the server.
- Retained mode graphics Advantages:
  - Reduced network traffic
  - Allows much of the overhead in executing commands to be done once and have the results stored in the display list on the graphics server
- Retained mode graphics Disadvantages:
  - Display lists require memory on the server
  - There is an overhead of creating a display list if the data are changing

Definition and Execution of Display Lists:

- Display lists are defined similarly to geometric primitives

  Ex:

```
#define BOX 1   //or some other unused integer
glNewList(BOX, GL_COMPILE);
        glBegin(GL_POLYGON);
                glColor3f(1.0,0.0,0.0);
                gVertex2f(-1.0,-1.0);
                gVertex2f(1.0,-1.0);
                gVertex2f(1.0,1.0);
                gVertex2f(-1.0,1.0);
        glEnd();
glEndList();
```

- There is a **glNewList** at the beginning and a **glEndList** at the end, with the contents in between.
- Each display list must have a unique identifier – an integer
- The flag **GL_COMPILE** tells 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, we can use the flag **GL_COMPILE_AND_EXECUTE**
- Each time that we wish to draw the box on the server, we execute the function as follows:

  glCallList(BOX);
- The current state determines which transformations are applied to the primitives in the display list.

Prepared By: Shatananda Bhat P

Ex:

```
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);
}
```

- Each time that **glCallList** is executed, the box is redrawn with a larger clipping rectangle.
- The current state can be safeguarded by using the matrix and attribute stacks provided by OpenGL
- A **stack** is a data structure in which the item placed most recently in the structure is first removed
- We can save the present values of attributes and matrices by **pushing** them on the top of the appropriate stack
- We can recover them later by **popping** them from the stack.
- A standard and safe procedure is always to push both the attributes and matrices on their own stack when we enter the display list, and to pop them when we exit.
- In the beginning of the display list,
  glPushAttrib(GL_ALL_ATTRIB_BITS);
  glPushMatrix();
- At the End
  glPopAttrib();
  glPopMatrix();
- We can create multiple display lists with consecutive identifiers more easily if we use the function ,
  glGenLists(number)
- We can execute multiple display list using the following single function call,
  glCallLists();

Prepared By: Shatananda Bhat P

TEXT AND DISPLAY LISTS:

Two types of Text:

1. Raster Text
2. Stroke Text

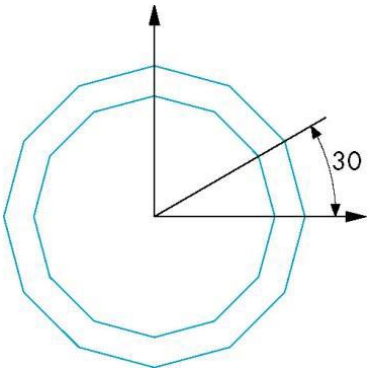
- Regardless of which type we chose, we need a reasonable amount of code to describe a set of characters.
- An efficient strategy is to define the font once, using a display list for each character, and then to store the font on the server.
- First, we define a function OurFont(char c), which will draw any ASCII character c that can appear in our string.
- The function is:

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

- Within each case, we can use the translate function to get the desired spacing or shift the vertex positions.

```
case 'O':
        glTranslate(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 degree in radians*/
            glVertex2f(0.4*cos (angle)+0.5; 0.4*sin(angle)+0.5);
            glVertex2f(0.4*cos(angle)+0.5; 0.5*sin(angle)+0.5);
        }
        glEnd();
        break;
```

Prepared By: Shatananda Bhat P



- We are working with two-dimensional characters.
- Hence, each character is defined in the plane z=0,
- We assume that each character fits inside the box.
- The usual strategy is to start over the lower-left corner of the first character in the string and to draw one string at a time.
- The first translation moves to center of the "O" character's box, which we set to be unit square.
- We then define our vertices using two concentric circles centered at this point.
- After the 12 quadrilaterals in the strip are defined, we move to the lower right corner of the box.
- Now suppose we want to generate a 256-character set. The required code is:-

```
base=glGenLists(256) ;

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

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

glListBase(base);

Finally, our drawing of a string is accomplished in the server by the function call

```
char *text_string;
glCallLists((Glint) strlen(text_string), GL_BYTE, text_string);
```

which make the use of the standard C library function ***strlen*** to find the length of the string. The argument in the above function is the number of list to be executed.

Prepared by: Shatananda Bhat P

Fonts in GLUT

- We prefer to use an existing font rather than to define our own
- GLUT provides a few raster and stroke fonts
- They do not make use of the display lists
- We can access a single character from a monotype font by the following function call:

  glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)
  glutBitmapCharacter(GLUT_BITMAP_8_BY_13, in
  t character)

Display Lists and Modelling:

- •Since display lists can call other display lists , they can be used to build hierarchical models
- Consider a simple face modeling system.

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

#define EAR 2
glNewList(EAR );
/* ear code */
glEndList();

#define NOSE 3
glNewList(NOSE );
/* nose code */
glEndList();
#define MOUTH 4
glNewList(MOUTH );
/* mouth code */
glEndList();
```

```
#define FACE 5
glNewList(FACE );
        /*draw the outline*/
        glTranslatef(....);
        glCallList(EYE);
        glTranslatef(....);
        glCallList(EYE);
        glTranslatef(....);
        glCallList(EAR);
        glTranslatef(....);
        glCallList(EAR);
        glTranslatef(....);
        glCallList(NOSE);
        glTranslatef(....);
        glCallList(MOUTH);
glEndList();
```

Prepared By: Shatananda Bhat P

PROGRAMMING EVENT-DRIVEN INPUT:

- GLUT Library recognizes a small set of events that is common to most window systems.
- These are sufficient for developing basic interactive graphics programs.

Using the Pointing Device:

- A move event is generated when the mouse is moved with one of the buttons pressed.
- If the mouse is moved without a button being held down, this event is called a passive move event.
- A mouse event occurs when one of the mouse buttons is either pressed or released.
- The information sent to the application program after the mouse event include the following:
  1. The button that generated the event
  2. The state of the button after the event (up or down)
  3. The position of the cursor tracking the moue in window coordinates (with the origin in the upper-left corner of the window)

We register the mouse callback function inside the main function , by means of the GLUT function as follows:

glutMouseFunc(myMouse);

The mouse callback must have the form:

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

Within the callback function, we define the actions that we want we want to take place if the specifies event occurs.

Prepared By: Shatananda Bhat P

Window Event:

- Most window systems allow a user to resize the window interactively, usually by using the mouse to drag a corner of the window to a new location.
- The reshape event is generated whenever the window is resized

Reshape Callback Function:

glutReshapeFunc(myReshape);

```
void myReshape(GLsizei w, GLsizei h)
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, (GLdouble)w, 0.0, (GLdouble)h, -1.0, 1.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glViewport(0, 0, w, h);
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glFlush();

        ww = w;
        wh = h;
}
```

Keyboard Events:

- Keyboard events are generated when the mouse is in the window and one of the keys is pressed or released.
- **glutKeyboardFunc** is the callback for events generated by pressing a key
- **glutKeyboardUpFunc** is the callback for events generated by releasing a key

```
glutKeyboardFunc(myKey);

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

Prepared By Shatananda Bhat P

Mouse Motion Event

glutMotionFunc(function_name);

The motion callback for a window is called when the mouse moves within the window while one or more mouse buttons are pressed.

glutPassiveMotionFunc(function_name);

The passive motion callback for a window is called when the mouse moves within the window while no mouse buttons are pressed.

Display Callback:

glutDisplayFunc(myDisplay);

- It is invoked when GLUT determines that the window should be redisplayed.
- One such situation occurs when the window is opened initially
- Another happens after a resize event
- A display callback is a good place to put the code that generates most non-interactive output

glutPostRedisplay();

- This function causes the display to be redrawn
- It ensures that the display will be drawn only once each time the program goes through the loop
- Avoids extra or unnecessary screen drawings

Idle Callback:

glutIdleFunc(myIdleFunc);

- This function is invoked when there are no other events.
- Its default is the null function pointer
- Its typical use is to generate graphical primitives through a display function while nothing else is happening.

Prepared By: Shatananda Bhat P

Window Management:

Id=glutCreateWindow("Second Window");

- The returned integer value allows us to select this window as the current window into which objects will be rendered as follows:

glutSetWindow(id);

MENUS:

- GLUT provides pop-up menus, that we can use with mouse to create sophisticated interactive applications.
- Using menu involves the following steps:
  - We must define the action corresponding to each entry in the menu.
  - We must link the menu to a particular mouse button
  - Finally, we must register a callback function for **each menu**

Ex:

glutCreateMenu(demo_Menu);

//Registers the callback function

glutAddMenuEntry("Quit", 1);

// The second argument in each entry's definition is the identifier passed to the callback when the entry is selected

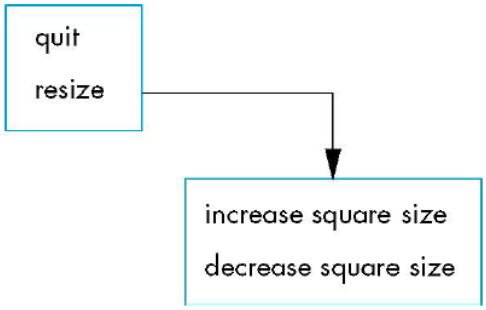
glutAddMenuEntry("Increase Square Size", 2);
glutAddMenuEntry(" Increase Square Size ", 3);
glutAttachMenu(GLUT_RIGHT_BUTTON);

Sample call back function:

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

Prepared By: Shatananda Bhat P

GLUT also supports hierarchical menus.



Ex:
sub_menu= glutCreateMenu(size_menu);
glutAddMenuEntry("Increase the Square Size",1);
glutAddMenuEntry("Decrease the Square Size",2);
glutCreateMenu(top_menu);
glutAddMenuEntry("Quit",1;)
glutAddSubMenu("Resize", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);

Program to Increase and Decrease the size of a square using Mouse Callback and MENU:

```
#include<stdio.h>
#include<GL\gl.h>
#include<GL\glut.h>

GLsizei wh = 500, ww = 500;
GLint size=50;


void myinit()
{
        glViewport(0, 0, ww, wh);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0, (GLdouble)wh, 0.0, (GLdouble)ww);
        glMatrixMode(GL_MODELVIEW);
        glClearColor(1.0, 1.0, 1.0, 1.0);
        glColor3f(1.0, 0.0, 0.0);
}

void display()
{
```

Prepared By: Shatananda Bhat P

```
        glClear(GL_COLOR_BUFFER_BIT);
}

void drawsquare(int x, int y, int size)
{
        y = wh - y; //match window coordinates to view volume coordinates

        glColor3f(0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_LINE_LOOP);
        glVertex2f(x - size, y - size);
        glVertex2f(x - size, y + size);
        glVertex2f(x + size, y + size);
        glVertex2f(x + size, y - size);
        glEnd();
        glFlush();
}

void myMouse(int btn, int state, int x, int y)
{
        if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
                drawsquare(x, y, size/2);
}

void sub_menu(int id)
{
        switch (id)
        {
        case 1: size = size * 2;
                        glutMouseFunc(myMouse);
                        break;

        case 2: size = size / 2;
                glutMouseFunc(myMouse);
                break;
        }
        glutPostRedisplay();
}

void main_menu(int id)
```

Prepared By: Shatananda Bhat P

```
{
        switch (id)
        {
        case 1: sub_menu(id);
                break;

        case 2: exit(0);
                break;
        }
        glutPostRedisplay();
}


int main(int argc, char **argv)
{
        int submenu;
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(500, 500); /* 500 x 500 pixel window */
        glutInitWindowPosition(0, 0); /* place window top left on display */
        glutCreateWindow("Draw square on click"); /* window title */
        myinit();
        glutDisplayFunc(display);
        glutMouseFunc(myMouse);
        submenu = glutCreateMenu(sub_menu);                     //Create submenu
        glutAddMenuEntry("Increase Rectangle Size", 1);
        glutAddMenuEntry("Decrease Rectangle Size", 2);
        glutCreateMenu(main_menu);
        glutAddSubMenu("Resize", submenu);              //add submenu to the main menu
        glutAddMenuEntry("Quit", 1);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
        glutMainLoop();
}
```

Prepared By: Shatananda Bhat P

PICKING:

- Picking is the logical input operation that allows the user to identify an object on the display
- Picking uses the pointing device
- The information returned to application program after picking in not a position
- Old display processors could accomplish picking easily by means of a lightpen
- Each redisplay of the screen would start at a precise time
- The lightpen would generate an interrupt when the redisplay passed through its sensor
- The processor could determine which object was being displayed
- One reason for the difficulty of picking in modern systems is the forward nature of the rendering pipeline
- Converting from a location on the display to the corresponding primitive is not a direct calculation

PICKING- SELECTION:

- Selection involves adjusting the clipping region and viewport
- We can keep track of which primitives in a small clipping region are being rendered into a region near the cursor.
- These primitives go into a hit list that can be examined later by the user program.
- A simple approach is to use bounding boxes or extents for objects of interest.
- The extent of an object is the smallest rectangle , aligned with the coordinate axes, that contains the object.
- For 2D applications , bounding box is a rectangle
- For 3D applications , bounding box is a right parallelepiped
- Another simple approach involves using the back buffer and an extra rendering
- When we use double buffering , we use two color buffers: a front buffer and a back buffer
- We can use the back buffer for purposes other than for rendering the scene that we wish to display.

Steps:

1. We draw the objects into the back buffer with the pick colors.
2. We get the position of the mouse using the mouse callback.
3. We use the function glReadPixels() to find the color at the position in the frame buffer corresponding to the mouse button.
4. We search the table of colors to find the object corresponds to the color read.

- The basic idea of selection mode is that the objects in a scene can be rendered, but not necessarily in the color buffer.

Prepared By: Shatananda Bhat P

- As we render objects, OpenGL can keep track of which objects render to any chosen area, by determining whether they are in a specified viewing volume.
- The function **glRenderMode** lets us select one of three modes:
  - GL_RENDER: Normal rendering to the color buffer
  - GL_SELECT: Selection Mode
  - GL_FEEDBACK: Feedback Mode
- The return value from **glRenderMode** can be used to determine the number of primitives that were in the clipping volume.
- When we enter selection mode and render a scene, each primitive within the clipping volume generates a message called a hit
- This message is stored in a buffer called the name stack
- We use the function **glSelectBuffer** to identify an array for the selection data.

There are four functions for :

–Initializing the name stack
–Pushing Information on to it
–Popping Information on it
–Manipulating the top entry of the stack

The information that we produce is called the **hit list**

•It can be examined after the rendering to obtain the information needed for picking.

- The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.
- **glLoadName** causes name to replace the value on the top of the name stack.
- The name stack is always empty while the render mode is not **GL_SELECT**. Calls to **glLoadName** while the render mode is not **GL_SELECT** are ignored.

void glSelectBuffer(Glsizei n,Gluint *buff);

- specifies the array buff 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

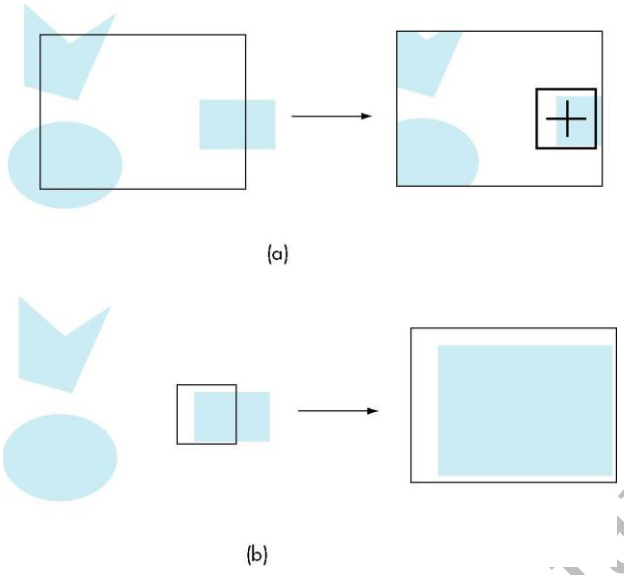
Prepared By: Shatananda Bhat P

void glLoadName(Gluint name)

-replaces the top of the name stack with name

void glPickMatrix(x,y,w,h,*vp)

- It creates a projection matrix for picking that restricts drawing to **w x h** area centered at **(x,y)** in window coordinates within the viewport **vp**




```
glRectf(x1, y1, x2, y2);
or
glBegin(GL_POLYGON);
glVertex2( x1, y1 );
glVertex2( x2, y1 );
glVertex2( x2, y2 );
glVertex2( x1, y2 );
glEnd( );
```

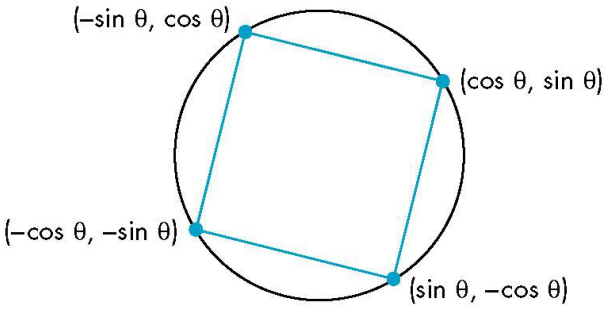
Prepared By: Shatananda Bhat P

Double Buffering:

Double buffering uses the following gunctions:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutSwapBuffers();
glDrawBuffer(GL_BACK);
glDrawBuffer(GL_FRONT_AND_BACK);
```

Ex:




```
/*
*  double.c
*  This program demonstrates double buffering for
*  flicker-free animation.  The left and middle mouse
*  buttons start and stop the spinning motion of the square.
*/

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

#define DEGREES_TO_RADIANS 3.14159/180.0

static GLfloat spin = 0.0;
GLfloat x, y;
int singleb, doubleb;

void square()
{
        glBegin(GL_QUADS);
        glVertex2f(x, y);
        glVertex2f(-y, x);
```

Prepared By: Shatananda Bhat P

```
        glVertex2f(-x, -y);
        glVertex2f(y, -x);
        glEnd();
}


void displayd()
{
        glClear(GL_COLOR_BUFFER_BIT);
        square();
        glutSwapBuffers();
}

void displays()
{
        glClear(GL_COLOR_BUFFER_BIT);
        square();
        glFlush();
}

void spinDisplay(void)
{
        spin = spin + 2.0;
        if (spin > 360.0) spin = spin - 360.0;
        x = 25.0*cos(DEGREES_TO_RADIANS * spin);
        y = 25.0*sin(DEGREES_TO_RADIANS * spin);
        glutSetWindow(singleb);
        glutPostRedisplay();
        glutSetWindow(doubleb);
        glutPostRedisplay();
}

void myinit()
{
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glColor3f(1.0, 1.0, 1.0);
        //glShadeModel(GL_FLAT);
}
```

Prepared By: Shatananda Bhat P

```
void mouse(int btn, int state, int x, int y)
{
        if (btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        glutIdleFunc(spinDisplay);
        if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        glutIdleFunc(NULL);
}

void myReshape(int w, int h)
{
        glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (w <= h)
                glOrtho(-50.0, 50.0, -50.0*(GLfloat)h / (GLfloat)w,
                50.0*(GLfloat)h / (GLfloat)w, -1.0, 1.0);
        else
                glOrtho(-50.0*(GLfloat)w / (GLfloat)h,
                50.0*(GLfloat)w / (GLfloat)h, -50.0, 50.0, -1.0, 1.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

}

int main(int argc, char** argv)
{

        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        singleb = glutCreateWindow("single buffered");
        myinit();
        glutDisplayFunc(displays);
        glutReshapeFunc(myReshape);
        glutIdleFunc(spinDisplay);
        glutMouseFunc(mouse);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowPosition(500, 0);
        doubleb = glutCreateWindow("double buffered");
        myinit();
        glutDisplayFunc(displayd);
```

Prepared By: Shatananda Bhat P

```
    glutReshapeFunc(myReshape);
    glutIdleFunc(spinDisplay);
    glutMouseFunc(mouse);
    glutMainLoop();

}
```

Using a Timer:

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

- Execution of this function starts a timer in the event loop
- It delays the loop for delay milliseconds
- When the timer has counted down the callback, the function time_func is executed.

Ex:

```
int n=60; /*desired frame rate */
glutTimerFunc(100,myTimer,n);
void myTimer(int v)
{
glutPostRedisplay();
glutTimerFunc(1000/n,myTimer,v);
}
```

Design of Interactive Programs

A good interactive program should include the following features:

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

Logic Operations

- OpenGL supports 16 logical modes
- GL_COPY (Copy mode) is the default mode

```
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_XOR);
```

Prepared By: Shatananda Bhat P

Curved Surfaces:

Equations for objects with curved boundaries can be expressed in either a parametric or a nonparametric form.

The various objects that are often useful in graphics applications include quadric surfaces, superquadrics, polynomial and exponential functions, and spline surfaces. These input object descriptions typically are tessellated to produce polygon-mesh approximations for the surfaces.

Quadric Surfaces:

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

Quadric surfaces, particularly spheres and ellipsoids, are common elements of graphics scenes, and routines for generating these surfaces are often available in graphics packages. Also, quadric surfaces can be produced with rational spline representations.

Sphere:

In Cartesian coordinates, 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 (Figure 2):

$$x = r\cos\phi\cos\theta, \qquad -\pi/2 \le \phi \le \pi/2$$

$$y = r\cos\phi\sin\theta, \qquad -\pi \le \theta \le \pi$$

$$z = r\sin\phi$$

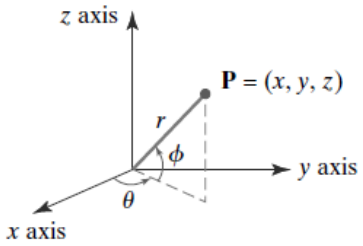


FIGURE 2
Parametric coordinate position (r, θ, ϕ) on the surface of a sphere with radius r.

The parametric representation in Equations 2 provides a symmetric range for the angular parameters θ and φ.

Shatananda Bhat P

Alternatively, we could write the parametric equations using standard spherical coordinates, where angle φ is specified as the colatitude (Figure 3). Then, φ is defined over the range $0 \le \varphi \le \pi$, and θ is often taken in the range $0 \le \theta \le 2\pi$.

We could also set up the representation using parameters u and v defined over the range from 0 to 1 by substituting $\varphi = \pi u$ and $\theta = 2\pi v$.

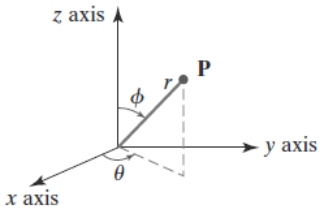


FIGURE 3
Spherical coordinate parameters (r, θ, ϕ), using colatitude for angle ϕ.

Ellipsoid:

An ellipsoidal surface can be described as an extension of a spherical surface where the radii in three mutually perpendicular directions can have different values (Figure 4).

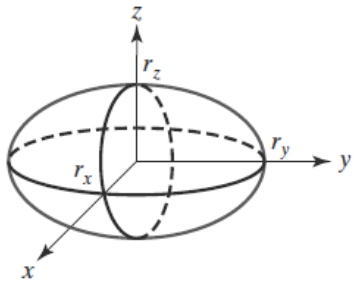


FIGURE 4
An ellipsoid with radii r_x, r_y, and r_z, centered on the coordinate origin.

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 in Figure 2 is

$$x = r_x \cos\phi \cos\theta, \qquad -\pi/2 \le \phi \le \pi/2$$

$$y = r_y \cos\phi \sin\theta, \qquad -\pi \le \theta \le \pi$$

$$z = r_z \sin\phi$$

Prepared By: Shatananda Bhat P

Torus:

Adoughnut-shaped object is called a torus or anchor ring. Most often it is described as the surface generated by rotating a circle or an ellipse about a coplanar axis line that is external to the conic.

The defining parameters for a torus are then the distance of the conic center from the rotation axis and the dimensions of the conic.

A torus generated by the rotation of a circle with radius r in the yz plane about the z axis is shown in Figure 5.

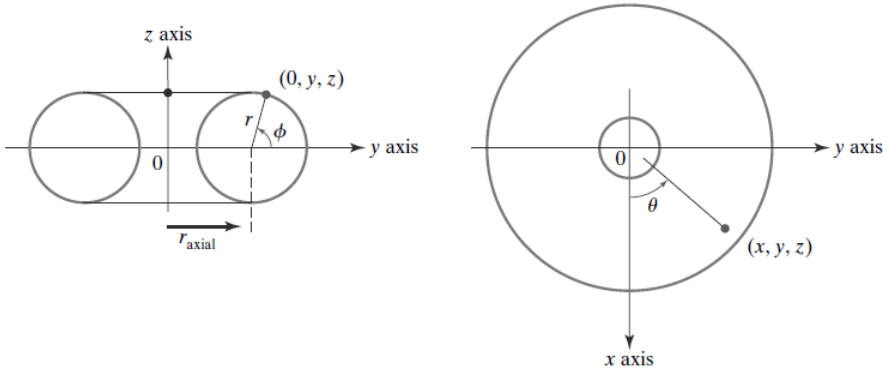


FIGURE 5
A torus, centered on the coordinate origin, with a circular cross-section and with the torus axis along the *z* axis.

With the circle center on the y axis, the axial radius, $\mathbf{r_{axial}}$, of the resulting torus is equal to the distance along the y axis to the circle center from the z axis (the rotation axis); and the cross-sectional radius of the torus is the radius of the generating circle.

The equation for the cross-sectional circle shown in the side view of Figure 5 is

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

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

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

We could also generate a torus by rotating an ellipse, instead of a circle, about the z axis. For an ellipse in the yz plane with semimajor and semiminor axes denoted as $\mathbf{r_y}$ and $\mathbf{r_z}$, 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:

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

Other variations on the preceding torus equations are possible. For example, we could generate a torus surface by rotating either a circle or an ellipse along an elliptical path around the rotation axis.

OpenGL Quadric-Surface and Cubic-Surface Functions:

GLUT Quadric-Surface Functions:

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

glutWireSphere (r, nLongitudes, nLatitudes);

or

glutSolidSphere (r, nLongitudes, nLatitudes);

where the sphere radius is determined by the double-precision floating-point number assigned to parameter **r**. Parameters **nLongitudes** and **nLatitudes** are used to select the integer number of longitude and latitude lines that will be used to approximate the spherical surface as a quadrilateral mesh.

Edges of the quadrilateral surface patches are straight-line approximations of the longitude and latitude lines.

The sphere is defined in modeling coordinates, centered at the world-coordinate origin with its polar axis along the z axis.

A GLUT cone is obtained with

glutWireCone (rBase, height, nLongitudes, nLatitudes);

or

glutSolidCone (rBase, height, nLongitudes, nLatitudes);

We set double-precision, floating-point values for the radius of the cone base and for the cone height using parameters **rbase** and **height**, respectively.

Prepared By: Shatananda Bhat P

As with aGLUT sphere, parameters **nLongitudes** and **nLatitudes** are assigned integer values that specify the number of orthogonal surface lines for the quadrilateral mesh approximation.

A cone longitude line is a straight-line segment along the cone surface from the apex to the base that lies in a plane containing the cone axis.

Each latitude line is displayed as a set of straight-line segments around the circumference of a circle on the cone surface that is parallel to the cone base and that lies in a plane perpendicular to the cone axis.

The cone is described in modeling coordinates, with the center of the base at the world-coordinate origin and with the cone axis along the world z axis.

Wire-frame or surface-shaded displays of a torus with a circular cross-sectionare produced with

glutWireTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);

or

glutSolidTorus (rCrossSection, rAxial, nConcentrics, nRadialSlices);

The torus obtained with these GLUT routines can be described as the surface generated by rotating a circle with radius **rCrossSection** about the coplanar z axis,

where the distance of the circle center from the z axis is **rAxial** (see Section 4).

We select a size for the torus using double-precision, floating-point values for these radii in the GLUT functions.

And the size of the quadrilaterals in the approximating surface mesh for the torus is set with integer values for parameters **nConcentrics** and **nRadialSlices**.

Parameter **nConcentrics** specifies the number of concentric circles (with center on the z axis) to be used on the torus surface, and parameter **nRadialSlices** specifies the number of radial slices through the torus surface. These two parameters designate the number of orthogonal grid lines over the torus surface, with the grid lines displayed as straight-line segments (the boundaries of the quadrilaterals) between intersection positions.

The displayed torus is centered on the world-coordinate origin, with its axis along the world z axis.

GLUT Cubic-Surface Teapot Function:

The data set for the Utah teapot, as constructed by Martin Newell in 1975, contains 306 vertices, defining 32 bicubic Bezier surface patches.

Since determining the surface coordinates for a complex object is time-consuming, these data sets, particularly the teapot surface mesh, became widely used.

We can display the teapot, as a mesh of over 1,000 bicubic surface patches, using either of the following two GLUT functions:

glutWireTeapot (size);

or

Prepared By: Shatananda Bhat P

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, we need to assign a name to the quadric, activate the GLU quadric renderer, and designate values for the surface parameters.

In addition, we can set other parameter values to control theappearance of a GLU quadric surface.

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);
```

A name for the quadric object is defined in the first statement, and, for this example, we have chosen the name **sphere1**.

This name is then used in other GLU functions to reference this particular quadric surface. Next, the quadric renderer is activated with the **gluNewQuadric** function, and then the display mode GLU LINE is selected for sphere1 with the **gluQuadricDrawStyle** command. Thus, the sphere is displayed in a wire-frame form with a straight-line segment between each pair of surface vertices.

Parameter **r** is assigned a double-precision value for the sphere radius, and the sphere surface is divided into a set of polygon facets by the equally spaced longitude and latitude lines.

We specify the integer number of longitude lines and latitude lines as values for parameters **nLongitudes** and **nLatitudes**.

Three other display modes are available for GLU quadric surfaces. Using the symbolic constant GLU_POINT in the gluQuadricDrawStyle, we display a quadric surface as a point plot.

For the sphere, a point is displayed at each surface vertex formed by the intersection of a longitude line and a latitude line. Another option is the symbolic constant GLU_SILHOUETTE.

This produces a wire-frame display without the shared edges between two coplanar polygon facets. And with the symbolic constant GLU_FILL, we display the polygon patches as shaded fill areas.

We generate displays of the other GLU quadric-surface primitives using the same basic sequence of commands. To produce a view of a cone, cylinder, or tapered cylinder, we replace the gluSphere function with

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

Prepared By: Shatananda Bhat P

The base of this object is in the xy plane (z=0), and the axis is the z axis.We assign adouble-precision radius value to the base of this quadric surface using parameter **rBase**, and we assign a radius to the top of the quadric surface using parameter **rTop**.

If rTop=0.0,weget a cone; if rTop=rBase,weobtain a cylinder. Otherwise, a tapered cylinder is displayed.

A double-precision height value is assigned to parameter height, and the surface is divided into a number of equally spaced vertical and horizontal lines as determined by the integer values assigned to parameters **nLongitudes** and **nLatitudes**.

A flat, circular ring or solid disk is displayed in the xy plane (z=0) andcentered 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**, which specify the number of radial slices to be used in the tessellation and the number of concentric circular rings, respectively.

Orientation for the ring is defined with respect to the z axis, with the front of the ring facing in the +z direction and the back of the ring facing in the −z direction.

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

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

The double-precision parameter **startAngle** designates an angular position in degrees in the xy plane measured clockwise from the positive y axis.

Similarly, parameter **sweepAngle** denotes an angular distance in degrees from the **startAngle** position. Thus, 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);

Parameter normalVectorDirection is assigned either GLU_OUTSIDE or GLU_INSIDE to indicate a direction for the surface normal vectors, where “outside” indicates the front-face direction and “inside” indicates the back-face direction. The default value is GLU_OUTSIDE.

Prepared By: Shatananda Bhat P

For the flat, circular ring, the default front-face direction is in the direction of the positive z axis ("above" the disk).

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

gluQuadricNormals (quadricName, generationMode);

A symbolic constant is assigned to parameter generationMode to indicate howsurface-normal vectors should be generated. The default is GLU_NONE, which means that no surface normals are to be generated and no lighting conditions typically are applied to the quadric surface.

For flat surface shading (a constant color value for each surface), we use the symbolic constant GLU_FLAT. This produces one surface normal for each polygon facet. 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.

Other options for GLU quadric surfaces include setting surface-texture parameters. In addition, 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);

Cubic-Spline Interpolation Methods:

This class of splines is most often used to set up paths for object motions orto provide a representation for an existing object or drawing, but interpolation splines are also used sometimes to design object shapes.

Cubic polynomials offer a reasonable compromise between flexibility and speed of computation. Compared to higher-order polynomials, cubic splines require less calculations and storage space, and they are more stable.

Compared to quadratic polynomials and straight-line segments, cubic splines are more flexible for modeling objectshapes.

Given a set of control points, cubic interpolation splines are obtained by fitting the input points with a piecewise cubic polynomial curve that passes through every control point. Suppose that we have n + 1 control points specified with coordinates

$$p_k = (x_k , y_k , z_k), \qquad k = 0, 1, 2, \ldots , n$$

Acubic interpolation fit of these points is illustrated in Figure 9.We can describe the parametric cubic polynomial that is to be fitted between each pair of control points with the following set of equations:

$$\begin{aligned} x(u) &= a_x u^3 + b_x u^2 + c_x u + d_x \\ y(u) &= a_y u^3 + b_y u^2 + c_y u + d_y, \qquad (0 \le u \le 1) \\ z(u) &= a_z u^3 + b_z u^2 + c_z u + d_z \end{aligned}$$

Prepared By: Shatananda Bhat P

For each of these three equations, we need to determine the values for the four coefficients a, b, c, and d in the polynomial representation for each of the n curve sections between the n+1 control points.

We do this by setting enough boundary conditions at the control-point positions between curve sections so that we can obtain numerical values for all the coefficients. In the following sections, we discuss common methods for setting the boundary conditions for cubic interpolation splines.

Natural Cubic Splines

One of the first spline curves to be developed for graphics applications is the natural cubic spline. This interpolation curve is a mathematical representation of the original drafting spline.

We formulate a natural cubic spline by requiring that two adjacent curve sections have the same first and second parametric derivatives at their common boundary. Thus, natural cubic splines have $\mathbf{C^2}$ continuity.

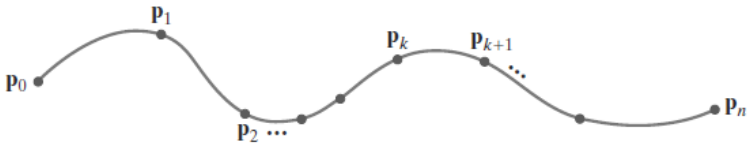


FIGURE 9
A piecewise continuous cubic-spline interpolation of $n + 1$ control points.

If we have **n + 1** control points, as in Figure 9, then we have n curve sections with a total of **4n** polynomial coefficients to be determined.

At each of the **n − 1** interior control points, we have four boundary conditions: The two curve sections on either side of a control point must have the same first and second parametric derivatives at that control point, and each curve must pass through that control point. This gives us 4n−4 equations to be satisfied by the 4n polynomial coefficients.

We obtain an additional equation from the first control point $\mathbf{p_0}$, the position of the beginning of the curve, and another condition from control point $\mathbf{p_n}$, which must be the last point on the curve. However, we still need two more conditions to be able to determine values for all the coefficients.

One method for obtaining the two additional conditions is to set the second derivatives at p_0 and p_n equal to 0. Another approach is to add two extra control points (called dummy points), one at each end of the original control-point sequence. That is, we add a control point labeled $\mathbf{p_{-1}}$ at the beginning of the curve and a control point labeled $\mathbf{p_{n+1}}$ at the end. Then all the original control points are interior points, and we have the necessary **4n** boundary conditions.

Although natural cubic splines are a mathematical model for the drafting spline, they have a major disadvantage. If the position of any of the control points is altered, the entire curve is affected. Thus, natural cubic splines allow for no "local control," so that we cannot restructure part of the curve without specifying an entirely new set of control points. For this reason, other representations for a cubic-spline interpolation have been developed.

Prepared By: Shatananda Bhat P

Hermite Interpolation:

A Hermite spline (named after the French mathematician Charles Hermite) is an interpolating piecewise cubic polynomial with a specified tangent at each control point. Unlike the natural cubic splines, Hermite splines can be adjusted locally because each curve section depends only on its endpoint constraints.

If **P(u)** represents a parametric cubic point function for the curve sectionbetween control points $\boldsymbol{p_k}$ and $\boldsymbol{p_{k+1}}$, as shown in Figure 10, then the boundary conditions that define this Hermite curve section are

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \mathbf{D}\mathbf{p}_k \\ \mathbf{P}'(1) &= \mathbf{D}\mathbf{p}_{k+1} \end{aligned}$$

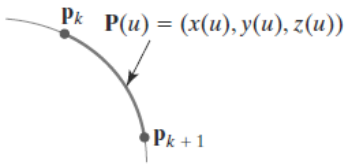


FIGURE 10
Parametric point function $\mathbf{P}(u)$ for a Hermite curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

with $\mathbf{Dp_k}$ and $\mathbf{Dp_{k+1}}$ specifying the values for the parametric derivatives (slope of the curve) at control points $\mathbf{p_k}$ and $\mathbf{p_{k+1}}$, respectively.

We can write the vector equivalent for this Hermite curve section as

$$\mathbf{P}(u) = \mathbf{a}\,u^3 + \mathbf{b}\,u^2 + \mathbf{c}\,u + \mathbf{d}, \qquad 0 \le u \le 1$$

where the x component of P(u) is $x(u) = a_x\,u^3 + b_x\,u^2 + c_x\,u + d_x$, and similarly for the y and z components. The matrix equivalent of Equation 10 is

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{11}$$

and the derivative of the point function can be expressed as

$$\mathbf{P}'(u) = [3u^2 \quad 2u \quad 1 \quad 0] \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{12}$$

Substituting endpoint values 0 and 1 for parameter u into the preceding two equations, we can express the Hermite boundary conditions 9 in the matrix form

$$\begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{13}$$

Solving this equation for the polynomial coefficients, we get

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \\ \mathbf{d} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

$$= \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

$$= \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix} \tag{14}$$

where $\mathbf{M}_H$, the Hermite matrix, is the inverse of the boundary constraint matrix. Equation 11 can thus be written in terms of the boundary conditions as

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_H \cdot \begin{bmatrix} \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{D}\mathbf{p}_k \\ \mathbf{D}\mathbf{p}_{k+1} \end{bmatrix}$$

Finally, we can determine expressions for the polynomial Hermite blending functions, $H_k(u)$ for $k = 0, 1, 2, 3$, by carrying out the matrix multiplications in Equation 15 and collecting coefficients for the boundary constraints to obtain the polynomial form

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_k(2u^3 - 3u^2 + 1) + \mathbf{p}_{k+1}(-2u^3 + 3u^2) + \mathbf{D}\mathbf{p}_k(u^3 - 2u^2 + u) \\ &\quad + \mathbf{D}\mathbf{p}_{k+1}(u^3 - u^2) \\ &= \mathbf{p}_k H_0(u) + \mathbf{p}_{k+1} H_1 + \mathbf{D}\mathbf{p}_k H_2 + \mathbf{D}\mathbf{p}_{k+1} H_3 \end{aligned} \tag{16}$$

Cardinal Splines:

As with Hermite splines, the cardinal splines are interpolating piecewise cubic polynomials with specified endpoint tangents at the boundary of each curve section.

The difference is that we do not input the values for the endpoint tangents. For a cardinal spline, the slope at a control point is calculated from the coordinates of the two adjacent control points.

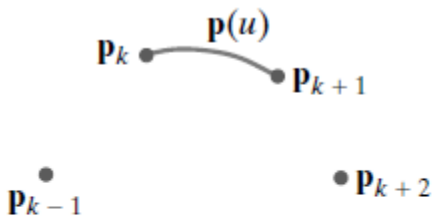


FIGURE 12
Parametric point function $\mathbf{P}(u)$ for a cardinal-spline section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$.

anda Bhat P

A cardinal spline section is completely specified with four consecutive control-point positions. The middle two control points are the section endpoints, and the other two points are used in the calculation of the endpoint slopes. If we take $\mathbf{P}(u)$ as the representation for the parametric cubic point function for the curve section between control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$, as in Figure 12, then the four control points from $\mathbf{p}_{k-1}$ to $\mathbf{p}_{k+1}$ are used to set the boundary conditions for the cardinal-spline section as

$$\begin{aligned} \mathbf{P}(0) &= \mathbf{p}_k \\ \mathbf{P}(1) &= \mathbf{p}_{k+1} \\ \mathbf{P}'(0) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+1} - \mathbf{p}_{k-1}) \\ \mathbf{P}'(1) &= \frac{1}{2}(1-t)(\mathbf{p}_{k+2} - \mathbf{p}_k) \end{aligned} \tag{17}$$

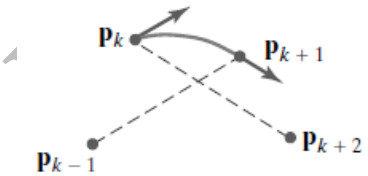


FIGURE 13
Tangent vectors at the endpoints of a cardinal-spline section are parallel to the chords formed with neighboring control points (dashed lines).

Thus, the slopes at control points $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ are taken to be proportional, respectively, to the chords $\overline{\mathbf{p}_{k-1}\mathbf{p}_{k+1}}$ and $\overline{\mathbf{p}_k\mathbf{p}_{k+2}}$ (Figure 13). Parameter t is called the **tension** parameter because it controls how loosely or tightly the cardinal spline fits the input control points. Figure 14 illustrates the shape of a cardinal curve for very small and very large values of tension t. When $t = 0$, this class of curves is referred to as **Catmull-Rom splines,** or **Overhauser splines.**

Using methods similar to those for Hermite splines, we can convert the boundary conditions 17 into the matrix form

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_C \cdot \begin{bmatrix} \mathbf{p}_{k-1} \\ \mathbf{p}_k \\ \mathbf{p}_{k+1} \\ \mathbf{p}_{k+2} \end{bmatrix} \tag{18}$$

where the cardinal matrix is

$$\mathbf{M}_C = \begin{bmatrix} -s & 2-s & s-2 & s \\ 2s & s-3 & 3-2s & -s \\ -s & 0 & s & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{19}$$

with $s = (1 - t)/2$.

Expanding Equation 18 into polynomial form, we have

$$\begin{aligned} \mathbf{P}(u) &= \mathbf{p}_{k-1}(-s\,u^3 + 2s\,u^2 - s\,u) + \mathbf{p}_k[(2-s)u^3 + (s-3)u^2 + 1] \\ &\quad + \mathbf{p}_{k+1}[(s-2)u^3 + (3-2s)u^2 + s\,u] + \mathbf{p}_{k+2}(s\,u^3 - s\,u^2) \\ &= \mathbf{p}_{k-1}\,\mathrm{CAR}_0(u) + \mathbf{p}_k\,\mathrm{CAR}_1(u) + \mathbf{p}_{k+1}\,\mathrm{CAR}_2(u) + \mathbf{p}_{k+2}\,\mathrm{CAR}_3(u) \end{aligned} \tag{20}$$

where the polynomials $\mathrm{CAR}_k(u)$ for k = 0, 1, 2, 3 are the cardinal-spline blending (basis) functions.

Prepared By

Kochanek-Bartels Splines:

These interpolating cubic polynomials are extensions of the cardinal splines.

Two additional parameters are introduced into the constraint equations defining Kochanek-Bartels splines to provide further flexibility in adjusting the shapes of curve sections.

Given four consecutive control points, labeled $\mathbf{p}_{k-1}$, $\mathbf{p}_k$, $\mathbf{p}_{k+1}$, and $\mathbf{p}_{k+2}$, we define the boundary conditions for a Kochanek-Bartels curve section between $\mathbf{p}_k$ and $\mathbf{p}_{k+1}$ as

$$\begin{aligned}
\mathbf{P}(0) &= \mathbf{p}_k \\
\mathbf{P}(1) &= \mathbf{p}_{k+1} \\
\mathbf{P}'(0)_{\text{in}} &= \frac{1}{2}(1-t)[(1+b)(1-c)(\mathbf{p}_k - \mathbf{p}_{k-1}) \\
&\qquad + (1-b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k)] \\
\mathbf{P}'(1)_{\text{out}} &= \frac{1}{2}(1-t)[(1+b)(1+c)(\mathbf{p}_{k+1} - \mathbf{p}_k) \\
&\qquad + (1-b)(1-c)(\mathbf{p}_{k+2} - \mathbf{p}_{k+1})]
\end{aligned} \tag{21}$$

where t is the **tension** parameter, b is the **bias** parameter, and c is the **continuity** parameter. In the Kochanek-Bartels formulation, parametric derivatives might not be continuous across section boundaries.

Tension parameter t has the same interpretation as in the cardinal spline formulation; that is, it controls the looseness or tightness of the curve sections. Bias, b, is used to adjust the curvature at each end of a section so that curve sections can be skewed toward one end or the other.

Parameter c controls the continuity of the tangent vector across the boundaries of sections. If c is assigned a nonzero value, there is a discontinuity in the slope of the curve across section boundaries.

Kochanek-Bartels splines were designed to model animation paths. In particular, abrupt changes in the motion of an object can be simulated with nonzero values for parameter c. These motion changes are used in cartoon animations, for example, when a cartoon character stops quickly, changes direction, or collides with some other object.

Bezier Spline Curves:

This spline approximation method was developed by the French engineer Pierre Bezier for use in the design of Renault automobile bodies.

Bezier splines have a number of properties that make them highly useful and convenient for curve and surface design. They are also easy to implement. For these reasons, Bézier splinesare widely available in various CAD systems, in general graphics packages, and in assorted drawing and painting packages.

Prepared By: Sha

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.

The degree of the Bézier polynomial is determined by the number of control points to be approximated and their relative position.

As with the interpolation splines, we can specify the Bézier curve path in the vicinity of the control points using blending functions, a characterizing matrix, or boundary conditions. For general Bézier curves, with no restrictions on the number of control points, the blending function specification is the most convenient representation.

Bézier Curve Equations

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 $\mathbf{P}(u)$, which describes the path of an approximating Bézier polynomial function between $\mathbf{p}_0$ and $\mathbf{p}_n$:

$$\mathbf{P}(u) = \sum_{k=0}^{n} \mathbf{p}_k \, \mathrm{BEZ}_{k,n}(u), \qquad 0 \le u \le 1 \tag{22}$$

The Bézier blending functions $\mathrm{BEZ}_{k,n}(u)$ are the *Bernstein polynomials*

$$\mathrm{BEZ}_{k,n}(u) = C(n, k)u^k(1 - u)^{n-k} \tag{23}$$

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

$$C(n, k) = \frac{n!}{k!(n - k)!} \tag{24}$$

Equation 22 represents a set of three parametric equations for the individual curve coordinates:

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

$$y(u) = \sum_{k=0}^{n} y_k \, \mathrm{BEZ}_{k,n}(u) \tag{25}$$

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

In most cases, a Bézier curve is a polynomial of a degree that is one less than the designated number of control points: Three points generate a parabola, four points a cubic curve, and so forth.

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

$$\text{BEZ}_{k,n}(u) = (1-u)\text{BEZ}_{k,n-1}(u) + u\,\text{BEZ}_{k-1,n-1}(u), \qquad n > k \geq 1 \qquad (27$$

with $\text{BEZ}_{k,k} = u^k$ and $\text{BEZ}_{0,k} = (1-u)^k$.

```
#include<GL/gl.h>

#include<GL/glut.h>
#include<math.h>
#include<stdio.h>
#define PI 3.14
void bezierCoefficients(int n,int *c)
{
        int k,i;
        for(k=0;k<=n;k++)
        {
                c[k]=1;
                for(i=n;i>=k+1;i--)
                c[k]*=i;
                for(i=n-k;i>=2;i--)
                        c[k]/=i;

        }
}

void display()
{

        int cp[4][2]={{10,10},{100,200},{200,20},{200,200}};
        int c[4],k,n=3;
        float x,y,u,blend;
        bezierCoefficients(n,c);
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0,0.0,0.0);
        glLineWidth(5.0);
        glBegin(GL_LINE_STRIP);

        for(u=0;u<1.0;u+=0.01)
        {x=0;y=0;
                for(k=0;k<4;k++)
```

Prepared By: Shatananda Bhat P

```
                {
                        blend=c[k]*pow(u,k)*pow(1-u,n-k);
                        x+=cp[k][0]*blend;
                        y+=cp[k][1]*blend;
                }
                glVertex2f(x,y);

        }
        glEnd();
        glFlush();
}

void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(5.0);
        gluOrtho2D(0.0,250.0,0.0,300.0);
}

int main(int argc, char ** argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(600,600);
        glutCreateWindow("Bezier Curve");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
        return 0;
}
```

Prepared By: Shatananda Bhat P

Properties of Bézier Curves:

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

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

From these expressions, we see that the slope at the beginning of the curve is along the line joining the first two control points, and the slope at the end of the curve is along the line joining the last two endpoints. Similarly, the parametric second derivatives of a Bézier curve at the endpoints are calculated as

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

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:

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

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. In Figure 23, a single coordinate position is input as two control points, and the resulting curve is pulled nearer to this position.

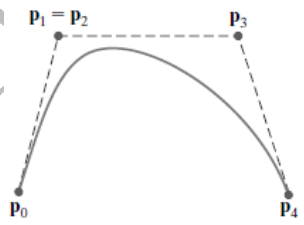


FIGURE 23

- We can fit a Bézier curve to any number of control points, but this requires the calculation of polynomial functions of higher degree.

Prepared By: Shatananda Bhat P

- 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.
- 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.

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 into the Bezier curve Equation, are

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

- The form of the blending functions determine how the control points influence the shape of the curve for values of parameter u over the range from 0 to 1.
- A cubic Bézier curve always begins at control point p_0 and ends at the position of control point p_3.
- Bézier curves do not allow for local control of the curve shape.
- If we reposition any one of the control points, the entire curve is affected.

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

$$\mathbf{P}(u) = [u^3 \quad u^2 \quad u \quad 1] \cdot \mathbf{M}_{Bez} \cdot \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} \tag{34}$$

where the **Bézier matrix** is

$$\mathbf{M}_{Bez} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{35}$$

Shatananda Bhat P

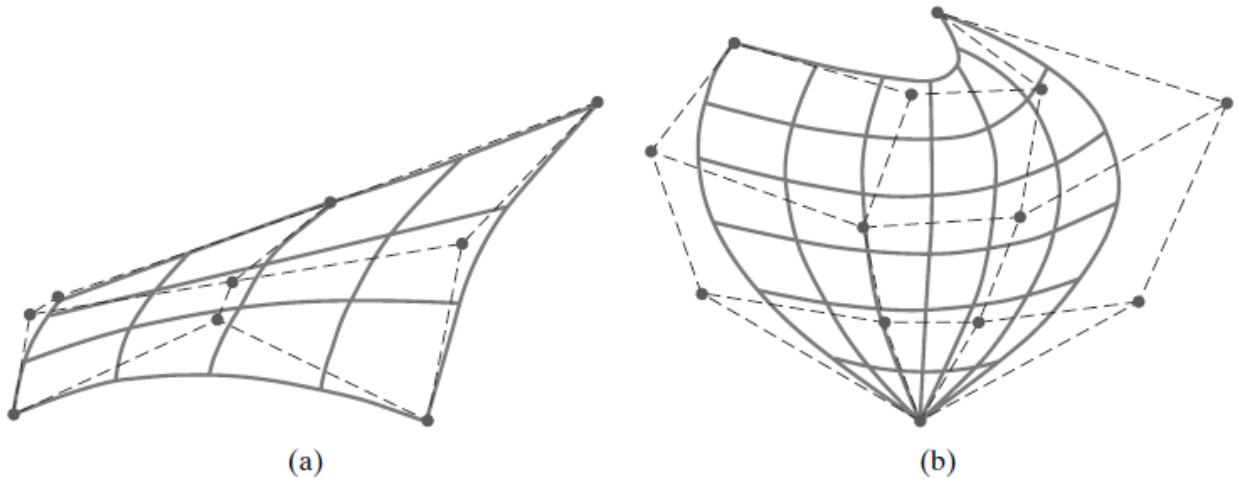
Bézier Surfaces:

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

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

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

Figure 26 illustrates two Bézier surface plots.



(a) (b)

- 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 three dimensional 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.

Prepared By: Shatananda Bhat P

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);

- A suffix code of f or d is used with glMap1 to indicate either floating-point or double precision for the data values.
- Minimum and maximum values for the curve parameter u are specified in uMin and uMax, although these values for a Bézier curve are typically set to 0 and 1.0, respectively.
- The three-dimensional, floating-point, Cartesian-coordinate values for the Bézier control points are listed in array ctrlPts, and the number of elements in this array is given as a positive integer using parameter nPts.
- Parameter 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.
- For a list of three-dimensional control-point positions, we set stride=3.

After we have set up the Bézier parameters and activated the curve-generation routines, we need to evaluate positions along the spline path and display the resulting curve. 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**. The suffix code for this function can be either **f** or **d**, and we can also use the suffix code **v** to indicate that the value for the argument is given in an array.

When **glEvalCoord1** processes a value for the curve parameter *u*, it generates a **glVertex3** function.

To obtain a Bézier curve, we thus repeatedly invoke the **glEvalCoord1** function to produce a set of points along the curve path, using selected values in the range from **uMin** to **uMax**.

Joining these points with straight-line segments, we can approximate the spline curve as a polyline.

Prepared By: Shatananda Bhat P

Example Code:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>

GLfloat ctrlpoints[4][3] = {
        { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
        {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord1f((GLfloat) i/30.0);
   glEnd();
   /* The following code displays the control points as dots. */
   glPointSize(5.0);
   glColor3f(1.0, 1.0, 0.0);
   glBegin(GL_POINTS);
      for (i = 0; i < 4; i++)
         glVertex3fv(&ctrlpoints[i][0]);
   glEnd();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
              5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
```

Prepared By: Shatananda Bhat P

```
  else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
            5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Although the previous example generated a spline curve with evenly spaced parameter values,wecan use the **glEvalCoord1f** function to obtain any spacing for parameter *u*.

Usually, however, 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);

- 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**, and 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.

For a curve that is to be displayed as a polyline, the output of these two functions is the same as the output from the following code, except that the argument of **glEvalCoord1** is set either to **u1** or to **u2** if $k = 0$ or $k = n$, respectively, to avoid round-off error. 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 ( );
```

Prepared By: Shatananda Bhat P

Thus, in the previous programming example, we could replace the block of code containing the loop for generating the Bézier curve with the following statements.

```
glColor3f (0.0, 0.0, 1.0);
glMapGrid1f (50, 0.0, 1.0);
glEvalMesh1 (GL_LINE, 0, 50);
```

OpenGL Bézier-Spline Surface Functions:

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);
```

- 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*.
- The three-dimensional Cartesian coordinates for the Bézier control points are listed in the doublesubscripted array **ctrlPts**, and the integer size of the array is given with parameters **nuPts** and **nvPts**.
- 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**.
- The integer offset between the beginning of coordinate values for control point $\mathbf{p}_{j,k}$ and the beginning of coordinate values for $\mathbf{p}_{j+1,k}$ is given in **uStride**; and the integer offset between the beginning of coordinate values for control point **p***j*,*k* and the beginning of coordinate values for **p***j*,*k*+1 is given in **vStride**.
- This allows the coordinate data to be intertwined with other data, so that we need to specify only the offsets to locate coordinate values.
- 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);
```

Instead of using the **glEvalCoord2** function, we can generate evenly spaced parameter values over the surface with

```
glMapGrid2* (nu, u1, u2, nv, v1, v2);
glEvalMesh2 (mode, nu1, nu2, nv1, nv2);
```

- The suffix code for glMapGrid2 is again either f or d, and parameter mode can be assigned the value GL POINT, GL LINE, or GL FILL.

Prepared By Shatananda Bhat P

- A two-dimensional grid of points is produced, with **nu** equally spaced intervals between u1 and u2, and with **nv** equally spaced intervals between v1 and v2.
- The corresponding integer range for parameter **u** is **nu1** to **nu2**, and the corresponding integer range for parameter v is **nv1** to **nv2**.

Example Code:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>

GLfloat ctrlpoints[4][4][3] = {
  {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
   {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
  {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
   {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
  {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
   {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
  {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
   {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};

void display(void)
{
  int i, j;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glPushMatrix ();
  glRotatef(85.0, 1.0, 1.0, 1.0);
  for (j = 0; j <= 8; j++) {
    glBegin(GL_LINE_STRIP);
    for (i = 0; i <= 30; i++)
      glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
    glEnd();
    glBegin(GL_LINE_STRIP);
    for (i = 0; i <= 30; i++)
      glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
    glEnd();
  }
  glPopMatrix ();
  glFlush();
}
```

Prepared By: Shatananda Bhat P

```
void init(void)
{
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
        0, 1, 12, 4, &ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  //glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_FLAT);
}

void reshape(int w, int h)
{
  glViewport(0, 0, (GLsizei) w, (GLsizei) h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if (w <= h)
    glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
          5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
  else
    glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
          5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}
```

Prepared By: Shatananda Bhat P