Computer Graphics - Chapter 3
Input and Interaction

Objectives are to learn about:

Introducing variety of devices that are used for interaction.

Learn about two different perspectives from which the input devices are considered:

1) The way that the physical devices can be described by the real-world properties,
2) The way that these devices appear to the application program.

How a client-server networks and graphics work.

Some of the questions we may ask are:

How users interact with the computer display?
What kind of interactions are possible with different input devices?
Does OpenGL support direct interaction?
What does OpenGL use for interactivity?
What are the common interactive devices.

Input Devices

There are two different ways to look at the devices:

Physical devices: keyboard or mouse, and discuss how they work
Logical devices: the way application programs look at the devices.

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

In C input and output are done using, *printf, scanf, getchar, and putchar*.

In computer graphics, the use of logical devices is slightly more complex. This is because the forms that input can take are more varied than the strings of bits or characters. In a nongraphical application, we restricted to bits or characters.

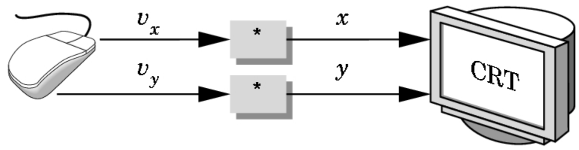
Each device has properties that make it more suitable for certain tasks than for others. There are two primary types of physical devices:

1) pointing devices
2) keyboard devices

Input Devices – cont.

We can view the output of the mouse or trackball as two independent values provided by the device.

Relative
Positioning
Devices



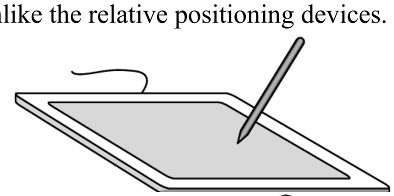
It is not necessary that the output of the mouse or trackball encoders be interpreted as a position. Instead, either the device driver or a user program can interpret the information from the encoder as two independent velocities.

The computer can then integrate these values to obtain a two-dimensional position.

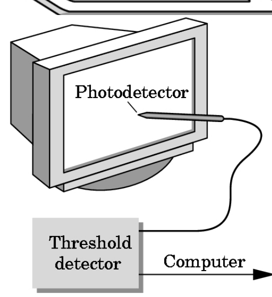
By integrating the distance traveled by the ball as a velocity, we can use the device as a variable-sensitivity input device in which large deviations from the rest causes rapid large changes and small deviations cause slow or small changes.

Input Devices – cont.

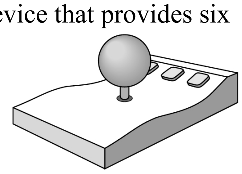
Data tablets provide absolute positioning unlike the relative positioning devices. A typical data tablet has rows and columns of wires embedded under its surface.



The lightpen is another device used in computer graphics. The lightpen contains a light-sensing device, such as a photocell. This device provide a direct positioning device.



Joystick is another physical device. In this device, the stick will control the two dimensional orthogonal motions. This two motions are integrated to identify the location on the screen.
Spaceball is another device that provides six degrees of freedom.



Logical Devices

The main characteristics that describes the logical behavior of an input device:

1) what measurements the device returns to the user program, and
2) when the device returns those measurements.

In general, there are six classes of logical input devices:

1. String – provides ASCII strings to the user program (logical implemented via keyboard)
2. Locator – provides a position in world coordinates to the user program (pointing devices and conversions may be needed)
3. Pick – returns the identifier of an object to the user program. (pointing devices and conversions may be needed)
4. Choice – allows users to select one of the distinct number of options (widgets – menus, scrollbars, and graphical buttons)
5. Dial – provides analog input to the user program (widgets – slidebars,…)
6. Stroke – it returns an array of locations (similar to multiple use of a locator, continuous)

Measure and Trigger

The manner by which physical and logical input devices provide input to an application program can be described in terms of two entities:

1) A measure process, and 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.

Example: The measure of a keyboard is a string,
The trigger could be the “return” or “enter” key.
Example: For a locator the measure includes the location and
The trigger can be the button on the pointing device.

When we develop an application program, we have to account for the user triggering the device while she is not pointing an object.

We can include, as part of the measure, a status variable that indicates that the user is not pointing to an object.

Input Modes

In addition to multiple types of logical input devices, we can obtain the measure of a device in three distinct modes:

1) Request mode, 2) Sample mode, and 3) Event mode.

It defined by the relationship between the measure process and the trigger. Normally, the initialization of an input device starts a measure process.

1) Request mode: In this mode the measure of the device is not returned to the program until the device is triggered.

A locator can be moved to different point of the screen. The Windows system continuously follows the location of the pointer, but until the button is depressed, the location will not be returned.

Input Modes – cont.

2) Sample mode: Input is immediate. As soon as the function call in the user program is encountered, the measure is returned, hence no trigger is needed.

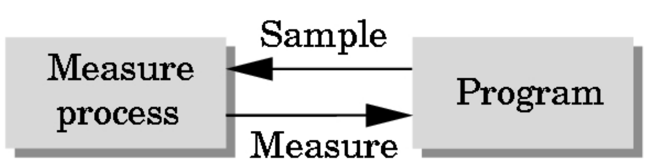
For both of the above modes, the user must identify which devices is to provide the input.

request_locator(device_id, &measure);

sample_locator(device_id, &measure);

identifier location

Think of a flight simulator with many input devices



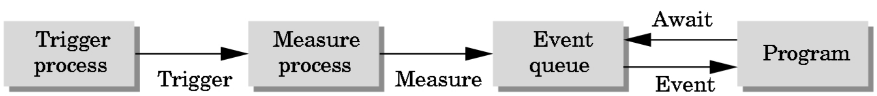
Input Modes – cont.

3) Event mode: The previous two modes are not sufficient for handling the variety of possible human-computer interactions that arise in a modern computing environment. The can be done in three steps:

1) Show how event mode can be described as another mode within the measure-trigger paradigm. 2) Learn the basics of client-servers when event mode is preferred, and 3) Learn how OpenGL uses GLUT to do this.

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, with the identifier for the device, is placed in an **event queue.** The user program executes the events from the queue. When the queue is empty, it will wait until an event appears there to execute it.

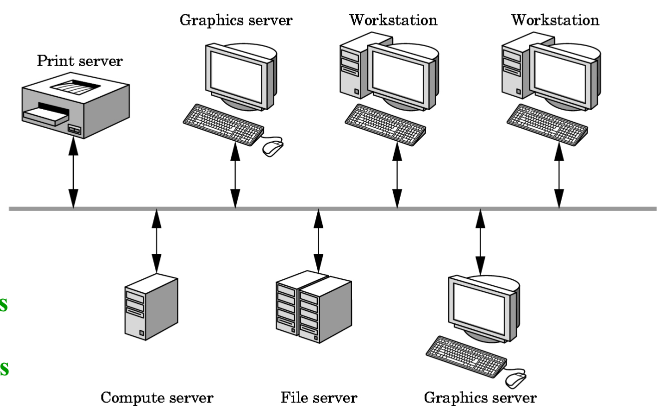
Another approach is to associate a function called a **callback** with a specific type of event. This is the approach we are taking.



Client and Servers

If a computer graphics is to be useful for a variety of real applications, it must function well in a world of distributed computing and networks.

In this world, the building blocks are entities called **servers** that can perform tasks for **clients**.

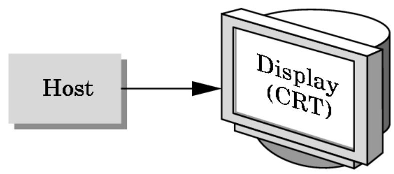


- **Print servers**
- **Compute servers**
- **File servers**
- **Terminal servers**

Display Lists

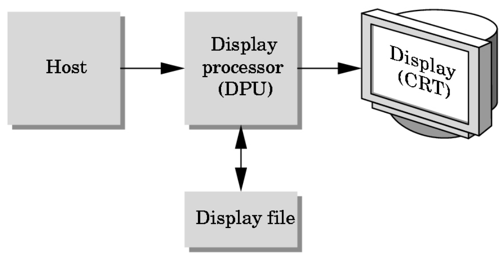
Display lists illustrate how we can use clients and servers on a network to improve graphics performance.

The computer would send out the necessary information to redraw the display at a rate sufficient to avoid noticeable flicker. In the past, since computers were slow and expensive, the cost of keeping even a simple display refreshed was prohibitive for all but a few applications.



The solution is to this problem was to build a special-purpose computer, called a **display processor**.

Today, the display processor is a a graphics server, and the user program on the host computer is a client.



Display Lists – cont.

We can send graphical entities to a display in one of the two ways:

1) Send the complete description of our objects to the graphics server. For a typical geometric primitives, this transfer consists of; sending vertices, attributes, and primitive types, in addition to viewing information.

2) Define the object once, then put its description in a display list. The display list is stored in the server and redisplayed by a simple function call issued from the client to the server. This method is called **retained mode** graphics.

Disadvantages associated with the use of display list:

1) Display lists require memory on the server,

2) There is an overhead for creating a display list.

Definition and Execution of Display Lists

Display lists have many things in common with ordinary files. OpenGL has a small set of functions to manipulate display lists, and places only a few restrictions on display-list contents.

Display lists are defined similarly to geometric primitives. There is a *glNewList* at he beginning and a *glEndList* at the end, and the contents in between. Each display list must have a unique identifier, usually an integer defined using *#define* directive. Example:

```
#define BOX 1 // or some other unused integer
glNewList(BOX, GL_COMPILE); //send the list to the server, but not to display its contents
        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;
```


Definition and Execution of Display Lists – cont.

Each time we wish to draw the box on the server, we execute the function: *glCallList(BOX);*

Note that the present state of the system determines which transformations are applied to the primitives in the display list. Thus, if we change the model-view or projection matrices between executions of the display list, the box will appear in different places or even will no longer appear. Example:

```
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);
}
```

Every time that the *glCallList* is executed, the box is redrawn with a different clipping rectangle.

Definition and Execution of Display Lists – cont.

The OpenGL stack data structure can be used to keep the matrix and its attribute. At the beginning of a display list, place:

glPushAttrib(GL_ALL_ATTRIB_BITS);
glPushMatrix( );

At the end place:
glPopAttrib( );
glPopMatrix( );

If you are not sure about which number to use for a list, use *glGenLists(number)*. This returns the first integer (or base) of number consecutive integers that are unused labels.

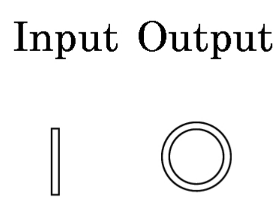
The function *glCallLists* allows us to execute multiple display lists with a single function call.

Text and Display Lists

Both raster and stroke text require a reasonable amount of code to describe set of characters.
Suppose we have used a 12x10 pattern of bits to store each character. 12x10 = 120 bits, 15 bytes to store each character.
To define a stroke font we need many more bytes.

Filled string (a)

Magnified outlines (b)



It will take a significant amount of resources to display a string of character. One way to fix this problem is to use a display list for each character, and then to store the font on the server via these display lists. Similar technique is used to display bitmap fonts.
Bit-map fonts are stored in ROM and each character is selected and displayed based on single byte; its ASCII code.

Text and Display Lists – cont.

We can define either the standard 96 printable ASCII characters or we can define patterns for a 256-character extended ASCII character set.

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

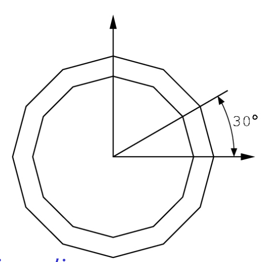
We have to be careful about spacing. Each character in the string must be displayed to the right of the previous character.

We can use the translate function *glTranslate* to get the desired spacing.

Text and Display Lists – cont.

Suppose we want to display letter O and we wish to fit it in a unit square:

```
switch( c )
{
    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), 0.4*sin(angle));
            glVertex2f(0.5*cos(angle), 0.5*sin(angle));
        }
        glEnd( );
        glTranslatef(0.5, -0.5,  0.0) ; //move to lower right
    break;
    …
}
```



Text and Display Lists – cont.

To generate 256 characters using this method, we can use a code like this one:

```
Base = glGenLists(256);  //return the index of first of 256 consecutive available
                                    //ids
for( I = 0 ; I < 256; I++)
{
        glNewList(base+I , GL_COMPILE);
        OurFont( I );
        glEndList( );
}
```

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

glListBase(base);

Then a string defined as *char *text_string;*

Can be drawn as:

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

Fonts in GLUT

Previous method requires us to create all letters. We prefer to use an existing font, rather than to define our own. GLUT provides a few raster and stroke fonts.

We can access a single character from a **monotype** (evenly spaced) font by the function call:

glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)

GLUT_STROKE_ROMAN provides proportionally spaced characters with a size of approximately 120 units maximum. Note that this may not be an appropriate size for your program, thus resizing may be needed.

We control the location of the first character using a translation. In addition, once each character is printed there will be a translation to the bottom right of that character.

Note that translation and scaling may affect the OpenGL state. It is recommended that we use the *glPushMatrix* and *glPopMatrix* as necessary, to prevent undesirable results.

Fonts in GLUT – cont.

Raster or bitmap characters are produced in a similar manner. For example:
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)

will produce an 8x13 character.
The position is defined directly in the frame buffer and are not subject to geometric transformations. A **raster position** will keep the position of the next raster primitive. This position can be defined using *Raster-Pos*( )*.

If a character have a different width, we can use the function
glutBitmapWidth(font, char)
To return the width of a particular character.

```
glRasterPos2i(rx, ry);
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, k);
rx += glutBitmapWidth(GLUT_BITMAP_8_BY_13, k);
```


Programming Event-Driven Input Using the Pointing Device

Two types of events are associated with the pointing device.

move event: is generated when the mouse is moved with one of the buttons depressed, for a mouse the **mouse event** happens when one of the buttons is depressed or released.

passive move event: is generated when the mouse is moved without a button being hold down.

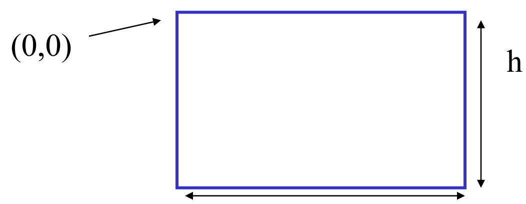
The mouse callback function looks like this:
glutMouseFunc(mouse_callback_func)

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

Within the callback function, we define what action we want to take place if the specified event occurs. There may be multiple actions defined in the mouse callback function corresponding to the many possible button and state combinations.

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert *y* coordinate returned by callback by height of window
  - *y = h – y;*



Using the Pointing Device

Suppose we want the program to terminate when the left button is depressed.

```
void mouse_callback_function(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        exit( 1);
}
```

It is obvious that depression or release of the other button will result in any action.

Window Events

Window events are relocation and the resizing of the window. In the window size changes, we have to consider three questions:

1) Do we redraw all the objects that were in the window before it was resized?

2) What do we do if the aspect ratio of the new window is different from that of the old window?

3) Do we change the size or attributes of new primitives if the size of the new window is different from that of the old?

Square Example.

The Square Program

The program is to draw an square by pressing the left button and to terminate the program by pressing the right button.

```
int main(int argc, char** argv){
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutCreateWindow("square");
   myinit ();
  glutReshapeFunc (myReshape);
  glutMouseFunc (mouse);
  glutMotionFunc(drawSquare);
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```


The Square Program

```
void myinit(void){
 /* Pick 2D clipping window,match size of screen window This choice avoids having to scale object
  coordinates each time window is resized */
 glViewport(0,0,ww,wh);
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 /* set clear color to black and clear window */
 glOrtho(0.0, (GLdouble) ww , 0.0, (GLdouble) wh , -1.0, 1.0)
 glClearColor (1.0, 0.0, 0.0, 1.0);
 glClear(GL_COLOR_BUFFER_BIT);
 glFlush();/* callback routine for reshape event */
 glutReshapeFunc(myReshape);
}

void mouse(int btn, int state, int x, int y){
 if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
 exit(1);
}/* display callback required by GLUT 3.0 */

void display(void){ }
```

The Square Program

```
#include <GL/glut.h>
#include<math.h>#include<stdlib.h>
GLsizei wh = 500, ww = 500; /* initial window size */
GLfloat size = 3.0;      /* half side length of square */

void drawSquare(int x, int y){
 y=wh-y;
 glColor3ub( (char) rand()%256, (char) rand()%256, (char) rand()%256);
 glBegin(GL_POLYGON);
 glVertex2f(x+size, y+size);
 glVertex2f(x-size, y+size);
 glVertex2f(x-size, y-size);
 glVertex2f(x+size, y-size);
 glEnd();
 glFlush();
}/* rehaping routine called whenever window is resizedor moved */

void myReshape(  GLsizei w, GLsizei h){/* adjust clipping box */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, (GLdouble)w, 0.0, (GLdouble)h, -1.0, 1.0);
 GlMatrixMode(GL_MODELVIEW);
 GlLoadIdentity(); /* adjust viewport and clear */
 GlViewport(0,0,w,h);
 GlClearColor (1.0, 1.0, 1.0, 1.0);

 glClear(GL_COLOR_BUFFER_BIT);
 glFlush();/* set global size for use by drawing routine */
 ww = w;
 wh = h;
}
```


Keyboard Events

We can use the keyboard event as an input device. Keyboard events are generated when the mouse is in the window and one of the keys is depressed.

In GLUT, there is no callback for the release of a key. The release does not generate a second event. Only one call back function for the keyboard:

glutKeyboardFunc(keyboard);

To use the keyboard to exit a program:

```
void keyboard(unsigned char key, int x, int y)
{
        if(key == 'q' || key == 'Q')
                exit(1);
}
```

The Display and idle Callback

We have seen the display callback:*glutDisplayFunc(display);*

It is invoked when GLUT determines that the window should be redisplayed. One such situation is when the window is open initially.

Since the display event will be generated when the window is first opened, the display callback is a good place to put the code that generates most non-interactive output. GLUT requires all programs to have a display function, even if it is empty.

Some of the things we can do with the display callback:

1) Animation – various values defined in the programs may change

2) Opening multiple windows

3) iconifying a window – replacing a window with a small symbol or picture.

glutPostRedisplay( );

The **idle callback** is invoked when there are no other events. Its default is the null function.

Window Management

GLUT supports both multiple windows and subwindows of a given window.

id = glutCreateWindow("Second Window");

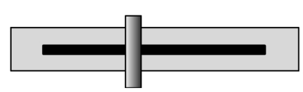
The returned integer value allows us to select this window as the current window:

glutSetWindow(id);

Menus

We can use our graphics primitives and our mouse callback to construct various graphical input devices.

How do you think we can create this one?



GLUT provides **pop-up menus**.

Using menus involves:

1) Must define the entries in the menu,
2) must link the menu to a particular mouse button, and
3) must define a callback function corresponding to each menu entry.

Example – Pop-up menu

glutCreateMenu(demo_menu);

glutAddMenuEntry("quit", 1);

glutAddMenuEntry("increase square size", 2);

glutAddMenuEntry("decrease square size", 3);

glutAttachMenu(GLUT_RIGHT_BUTTON);

The callback function looks like this:

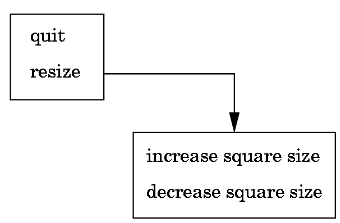
```
void demo_menu(int id)
{
        if(id == 1)  exit(1);
        else if (id == 2) size = 2*size;
        else size = size / 2;
        glutPostRedisplay( );
}
```


Example – Hierarchical menu

Suppose we want the main menu that we create to have two entries:

1) the first one to terminate the program

2) the second to pop-up a submenu.



Sub_menu = glutCreateMenu(size_menu);

glutAddMenuEntry("increase square size", 2);

glutAddMenuEntry("decrease square size", 3);

glutCreateMenu(top_menu);

glutAddMenuEntry("quit", 1);

glutAddSunMenu("Resize", sub_menu);

glutAttachMenu(GLUT_RIGHT_BUTTON);

Now we have to write the call back functions, *size_menu* and *top_menu*.

Picking

Picking is an input operation that allows the user to identify an object on the display. Although, the picking is done by a pointing device, the information returned to the application program is not a position.

A pick device is more difficult to implement than the locator device. There are two ways to do this:

1) **selection**, involves adjusting the clipping region and viewport such that we can keep track of which primitives in a small clipping region are rendered into a region near the cursor. Creates a **hit list**.

2) **bounding rectangles** or **extents,** this is the smallest rectangle, aligned with the coordinates axes, that contains the object.

A Simple Paint Program

This example illustrates the use of callbacks, display lists, and interactive program design by developing a simple paint program.

A paint program should demonstrate:

• Ability to work with geometric objects, such as line segments and polygons. It should allow us to enter the vertices interactively.

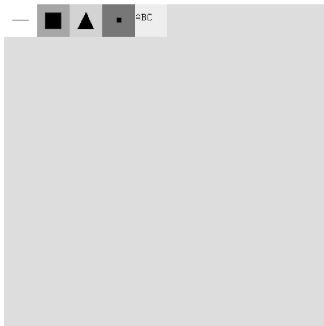
• Ability to manipulate pixels and to draw directly into the frame buffer.

• Should allow control of attributes such as color, line type, and fill pattern.

• It should include menus for controlling the application

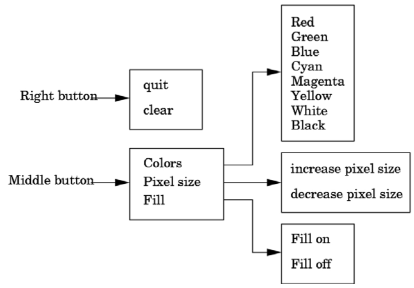
• It should behave correctly when the window is moved or resized.

Initial Window



The most difficult part is to decide what functionality to place in each of the functions.

Structure

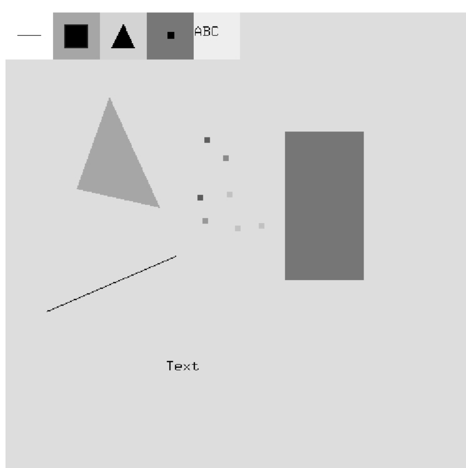


Controlled by the middle and right mouse buttons

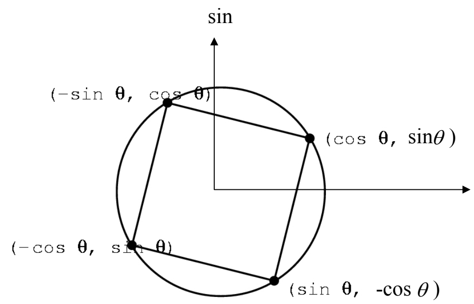
- Right button to clear the screen or to terminate the program
- Middle button allows us to change drawing color, to select between fill and no fill for the triangle and rectangle, and to change the size of the pixel rectangle

Animating Interactive Programs

The simple paint program - demo



The Rotating Square




```
void display( void )
{
  float pi = 3.14159;
  double t = 45.0*pi/180.0;   //30 degree

   for(int i = 0; i < 200; i++){
      t = t + i* (2*pi*0.1);
      glClear(GL_COLOR_BUFFER_BIT);  /*clear the window */
         glBegin(GL_POLYGON);
         glVertex2f(20*cos(t), 20*sin(t));
         glVertex2f(-20*cos(t), 20*sin(t));
         glVertex2f(-20*cos(t), -20*sin(t));
         glVertex2f(20*cos(t), -20*sin(t));
      glEnd();

      glFlush(); /* clear buffers */
  }
}
```


The idle function

The idle callback will allow us to put the program as if nothing is happening.

glutIdleFunc(idle);

We can use this in our rotate square program to rotate the square by an angle when nothing else is happening:

```
void idle( )
{
   t += 20;
   if( t >= 360)
        t  -= 360;
   glutPostRedisplay( );
}
```

Combine this with mouse callback *glutMouseFunc(mouse)* operation:

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

Don't for get to include glutMouseFunc (mouse); in the main.

Double Buffering

When we redisplay our CRT, we want to do so at a rate sufficiently high (50-80 times/sec) that we cannot notice the clearing and redrawing of the screen.

That means the contents of the frame buffer must be drawn at this rate.

One problem may occur when a complex shape is drawn. In that case the display may not be done in one refresh cycle. A moving object will be distorted.

Double buffering can provide a solution to these problems. The **front buffer** is displayed when we draw in the **back buffer**. We can swap the back and front buffer from the application program.

With each swap, the display callback is invoked.

The double buffering is set using the *GLUT_DOUBLE,* instead of *GLUT_SINGLE* in *glutInitDisplayMode.* The buffer swap function using GLUT will be: *glutSwapBuffer( );*

Double buffering does not speed of the process of displaying a complex display. It only ensures that we never see a partial display.

Design of Interactive Programs

A good interactive program includes features such as:

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