

Events and Event Handling for Computer Graphics

Prerequisites

An understanding of the fundamentals of creating a lighted object with your graphics API and of applying simple transformations.

Introduction

Graphics programming can focus entirely on creating one single image based on a set of data, but more and more we are seeing the value of writing programs that allow the user to interact with the world through graphical presentations, or that allow the user to control the way an image is created. These are called interactive computer graphics programs, and the ability to interact with information through an image is critically important to the success of this field.

Our emphasis in this module is on graphical interaction, not on user interfaces. Certainly many user interfaces use graphical presentations that give information to the user, take graphical actions, and interpret the results for program control, but we simply view these as applications of our graphics. A genuine discussion of user interfaces is much too deep for us to undertake here.

Interactive programming in computer graphics generally takes advantage of the event-handling capabilities of modern systems, so we must understand something of what events are and how to use them in order to write interactive graphics programs. Events are fairly abstract and come in several varieties, so we will need to go into some details as we develop this idea below. But modern graphics APIs handle events pretty cleanly, and you will find that once you are used to the idea, it is not particularly difficult to write event-driven programs. You should realize that some basic APIs do not include event handling, so it is sometimes necessary to use an extension to the API for this. For example, the OpenGL API generally uses the Graphics Library Utility Toolkit GLUT (or a similar extension) for event and window handling.

Definitions

An event is a formal record of some system activity, often an activity from a device such as a keyboard or mouse. An event record contains information that identifies the event and any data corresponding to the event. A keyboard event record contains the identity of the key that was pressed and the location of the cursor when it was pressed, for example; a mouse event record contains the mouse key that was pressed, if any, and the cursor's location on the screen when the event took place. Events are recorded in the event queue, which is managed by the operating system; this holds the event records and keeps track of the sequence in which events happen. When an event occurs, we say that the event is posted to the event queue. The operating system manages the event queue and as each event gets to the front of the queue, passes the event to the appropriate process to handle it. In general, events that involve a screen location get passed to whatever program owns that location, so if the event happens outside the program's window, that program will not get the event.

Programs that use events for control — and most interactive programs do this — manage that control through functions that are called event handlers. While these can gain access to the event queue in a number of ways, most APIs use functions called callbacks to handle events. When the system passes an event to the program, the program determines what kind of event it is and if any callback function has been registered for the event, passes control to that function. In fact, most interactive programs contain initialization and action functions, callback functions, and a main event loop. In OpenGL with the GLUT extension, this main event loop is quite explicit as a call to the function `glutMainLoop()` as the last action in the main program.

What happens in the main event loop is straightforward — the program gives up direct control of the flow of execution and places it in the hands of the user. From here on, the user will cause events that the program will respond to through the callbacks that have been created. We will see many examples of this approach in this, and later, sections of these notes.

A callback is a function that is executed when a particular event is recognized by the program. This recognition happens when an event comes off the event queue and the program has expressed an interest in the event. The key to being able to use a certain event in a program, then, is to express an interest in the event and to indicate what function is to be executed when the event happens. This is called registering a callback for an event, and we will see examples of this soon.

Some examples of events

keypress events, such as `keyDown`, `keyUp`, `keyStillDown`, ... Note that there are two kinds of keypress events — those that use the regular keyboard and those that use the so-called “special keys” such as the function keys or the cursor control keys. There may be different event handlers for these different kinds of keypresses. You should be careful when you use special keys, because different computers may have different special keys, and those that are the same may be laid out in different ways.

mouse events, such as `leftButtonDown`, `leftButtonUp`, `leftButtonStillDown`, ... Note that different “species” of mice have different numbers of buttons, so for some kinds of mice some of these events are collapsed.

system events, such as `idle` and `timer`, that are generated by the system based on the state of the event queue or the system clock, respectively.

software events, which are posted by programs themselves in order to get a specific kind of processing to occur next.

These events are very detailed, and many of them are not used in the APIs or API extensions commonly found with graphics. However, all could be used by going deeply enough into the system on which programs are being developed.

Note that event-driven actions are fundamentally different from actions that are driven by polling — that is, by querying a device or some other part of the system on some schedule and basing system activity on the results. There are certainly systems that operate by polling various kinds of input and interaction devices, but these are outside our current approach.

Callback registering

Below we will list some kinds of events and will then indicate the function that is used to register the callback for each event. Following that, we will give some code examples that register and use these events for some programming effects. This now includes only examples from OpenGL, but it should be extensible to other APIs fairly easily.

Event	Callback Registration Function
<code>idle</code>	<code>glutIdleFunc(animate)</code> requires a function as a parameter, here called <code>animate</code> , with template <code>void functionname(void)</code> . This function is the event handler that determines what is to be done at each idle cycle. Often this function will end with a call to <code>glutPostRedisplay()</code> as described below.

display `glutDisplayFunc(display)`
requires a function as a parameter, here called `display`, with template `void functionname(void)`. This function is the event handler that generates a new display whenever the display event is received.

keyboard `glutKeyboardFunc(keybd)`
requires a function as a parameter, here called `keybd`, with template `void functionname(unsigned char,int,int)`. This parameter function is the event handler that receives the character and the location of the cursor (`x,y`) when a key is pressed. As is the case for all callbacks that involve a screen location, the location on the screen has been converted to coordinates relative to the window. Again, this function will often end with a call to `glutPostRedisplay()` to re-display the scene with the changes caused by the particular keyboard event.

special `glutSpecialFunc(special)`
requires a function as a parameter, here called `special`, with template `void functionname(int key, int x, int y)`. This event is generated when one of the “special keys” is pressed; these keys are the function keys, directional keys, and a few others. The first parameter is the key that was pressed; the second and third are the integer window coordinates of the cursor when the keypress occurred. The usual approach is to use a special symbolic name for the key, such as `GLUT_KEY_F1` for the F1 function key or `GLUT_KEY_LEFT` for the left directional key. The only difference between the special and keyboard callbacks is that the events come from different kinds of keys.

menu `glutCreateMenu(options_menu)`
requires a function as a parameter, here called `options_menu`, with template `void functionname(int)`. This creates a menu that is brought up by a mouse button down event, specified by
`glutAttachMenu(event),`
and the function
`glutAddMenuEntry(string, int)`
identifies each of the choices in the menu and defines the value to be returned by each one. The menu choices are identified before the menu itself is attached, as illustrated in the lines below.

```
glutAddMenuEntry("text", VALUE);  
...  
glutAttachMenu(GLUT_RIGHT_BUTTON)
```

Note that the Macintosh uses a slightly different menu attachment with the same parameters,

```
glutAttachMenuName(event, string),
```

that attaches the menu to a name on the system menu bar. The Macintosh menu is activated by selecting the menu name from the menu bar, while the windows for Unix and Windows are popup windows that appear where the mouse is clicked and that do not have names attached.

Along with menus one can have sub-menus — items in a menu that cause a cascaded sub-menu to be displayed when they are selected. Sub-menus are created by the use of the function

`glutAddSubMenu(string, int)`
where the `string` is the text displayed in the original menu and the `int` is the identifier of the menu to cascade from that menu item. For more details, see the GLUT manuals.

`mouse` `glutMouseFunc(Mouse)`
requires a function as a parameter, here called `Mouse`, with a template such as `void Mouse(int button, int state, int mouseX, int mouseY)` where `button` indicates which button was pressed (an integer typically made up of one bit per button, so that a three-button mouse can indicate any value from one to seven), the `state` of the mouse (symbolic values such as `GLUT_DOWN` to indicate what is happening with the mouse) — and both raising and releasing buttons causes events — and integer values `xPos` and `yPos` for the window-relative location of the cursor in the window when the event occurred.

The mouse event does not use this function if it includes a key that has been defined to trigger a menu.

`mouse active motion` `glutMotionFunc(motion)`
requires a function as a parameter, here called `motion`, with a template like `void motion(int xPos, int yPos)` where `xPos` and `yPos` are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with one or more buttons pressed.

`mouse passive motion` `glutPassiveMotionFunc(pmotion)`
requires a function as a parameter, here called `pmotion`, with a template like `void pmotion (int xPos, int yPos)` where `xPos` and `yPos` are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with no buttons pressed.

`timer` `glutTimerFunc(msec, timer, value)`
requires an integer parameter, here called `msec`, that is to be the number of milliseconds that pass before the callback is triggered; a function, here called `timer`, with a template such as `void timer(int)` that takes an integer parameter, and an integer parameter, here called `value`, that is to be passed to the `timer` function when it is called.

Note that in any of these cases, the function `NULL` is an acceptable option. Thus you can create a template for your code that includes registrations for all the events your system can support, and simply register the `NULL` function for any event that you want to ignore.

Besides the kind of device events we generally think of, there are also software events such as the display event, created by a call to `glutPostRedisplay()`. There are also device events for devices that are probably not found around most undergraduate laboratories: the spaceball, a six-degree-of-freedom device used in high-end applications, and the graphics tablet, a device familiar to the computer-aided design world and still valuable in many applications. If you want to know more about handling these devices, you should check the GLUT manual.

A word to the wise...

This section discusses the mechanics of interaction through event handling, but it does not cover the critical questions of how a user would naturally control an interactive application. There are many deep and subtle issues involved in designing the user interface for such an application, and this module does not begin to cover them. The extensive literature in user interfaces will help you get a start in this area, but a professional application needs a professional interface — one designed, tested, and evolved by persons who focus in this area. When thinking of a real application, heed the old cliché: Kids, don't try this at home!

The examples below do their best to present user controls that are not impossibly clumsy, but they are designed much more to focus on the event and callback than on a clever or smooth way for a user to work. When you write your own interactive projects, think carefully about how a user might perceive the task, not just about an approach that might be easiest for you to program.

Code examples

This section presents four examples. This first is a simple animation that uses an idle event callback and moves a cube around a circle, in and out of the circle's radius, and up and down. The user has no control over this motion. When you compile and run this piece of code, see if you can imagine the volume in 3-space inside which the cube moves.

The second example uses keyboard callbacks to move a cube up/down, left/right, and front/back by using a simple keypad on the keyboard. This uses keys within the standard keyboard instead of using special keys such as a numeric keypad or the cursor control keys. A numeric keypad is not used because some keyboards do not have them; the cursor control keys are not used because we need six directions, not just four.

The third example uses a mouse callback to pop up a menu and make a menu selection, in order to set the color of a cube. This is a somewhat trivial action, but it introduces the use of pop-up menus, which are a very standard and useful tool.

Finally, the fourth example uses a mouse callback with object selection to identify one of two cubes that are being displayed and to change the color of that cube. Again, this is not a difficult action, but it calls upon the entire selection buffer process that is the subject of another later module in this set. For now, we suggest that you focus on the event and callback concepts and postpone a full understanding of this example until you have read the material on selection.

All of these examples are available as full source code in the accompanying source set.

Example: idle event callback

In this example, we assume we have a function named `cube()` that will draw a simple cube at the origin $(0, 0, 0)$. We want to move the cube around by changing its position with time, so we will let the idle event handler set the position of the cube and the display function draw the cube using the positions determined by the idle event handler. Much of the code for a complete program has been left out, but this illustrates the relation between the display function, the event handler, and the callback registration.

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void )
{
    glPushMatrix();
```

```

        glVertexf( cubex, cubey, cubez );
        cube();
        glPopMatrix();
    }

void animate(void)
{
    #define deltaTime 0.05

    // Position for the cube is set by modeling time-based behavior.
    // Try multiplying the time by different constants to see how that
    // behavior changes.

    time += deltaTime; if (time > 2.0*M_PI) time -= 2.0*M_PI;
    cubex = sin(time);
    cubey = cos(time);
    cubez = cos(time);
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
    /* Standard GLUT initialization precedes the functions below*/
    ...
    glutDisplayFunc(display);
    glutIdleFunc(animate);

    myinit();
    glutMainLoop();
}

```

Example: keyboard callback

Again we start with the familiar `cube()` function. This time we want to let the user move the cube up/down, left/right, or backward/forward by means of simple keypresses. We will use two virtual keypads:

Q	W		I	O
A	S		J	K
Z	X		N	M

with the top row controlling up/down, the middle row controlling left/right, and the bottom row controlling backward/forward. So, for example, if the user presses either Q or I, the cube will move up; pressing W or O will move it down. The other rows will work similarly.

Again, much of the code has been omitted, but the display function works just as it did in the example above: the event handler sets global positioning variables and the display function performs a translation as chosen by the user. Note that in this example, these translations operate in the direction of faces of the cube, not in the directions relative to the window.

```

GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time = 0.0;

void display( void )

```

```

{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void keyboard(unsigned char key, int x, int y)
{
    ch = ' ';
    switch (key)
    {
        case 'q' :
        case 'Q' :
        case 'i' :
        case 'I' :
            ch = key; cubey -= 0.1; break;
        case 'w' :
        case 'W' :
        case 'o' :
        case 'O' :
            ch = key; cubey += 0.1; break;
        case 'a' :
        case 'A' :
        case 'j' :
        case 'J' :
            ch = key; cubex -= 0.1; break;
        case 's' :
        case 'S' :
        case 'k' :
        case 'K' :
            ch = key; cubex += 0.1; break;
        case 'z' :
        case 'Z' :
        case 'n' :
        case 'N' :
            ch = key; cubez -= 0.1; break;
        case 'x' :
        case 'X' :
        case 'm' :
        case 'M' :
            ch = key; cubez += 0.1; break;
    }
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
    /* Standard GLUT initialization */
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    myinit();
    glutMainLoop();
}

```

The similar function, `glutSpecialFunc(. . .)`, can be used in a very similar way to read input from the special keys (function keys, cursor control keys, ...) on the keyboard.

Example: menu callback

Again we start with the familiar `cube()` function, but this time we have no motion of the cube. Instead we define a menu that allows us to choose the color of the cube, and after we make our choice the new color is applied.

```
#define RED      1
#define GREEN   2
#define BLUE    3
#define WHITE   4
#define YELLOW  5

void cube(void)
{
    ...

    GLfloat color[4];

    // set the color based on the menu choice

    switch (colorName) {
        case RED:
            color[0] = 1.0; color[1] = 0.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case GREEN:
            color[0] = 0.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case BLUE:
            color[0] = 0.0; color[1] = 0.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case WHITE:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case YELLOW:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
    }

    // draw the cube

    ...
}

void display( void )
{
    cube();
}

void options_menu(int input)
{
    colorName = input;
    glutPostRedisplay();
}

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

    glutCreateMenu(options_menu);           // create options menu
```

```

        glutAddMenuEntry("Red", RED);           // 1 add menu entries
        glutAddMenuEntry("Green", GREEN);       // 2
        glutAddMenuEntry("Blue", BLUE);        // 3
        glutAddMenuEntry("White", WHITE);      // 4
        glutAddMenuEntry("Yellow", YELLOW);    // 5
        glutAttachMenu(GLUT_RIGHT_BUTTON, "Colors");

        myinit();
        glutMainLoop();
    }

```

Example: mouse callback for object selection

This example is more complex because it illustrates the use of a mouse event in object selection. This subject is covered in more detail in the later chapter on object selection, and the full code example for this example will also be included there. We will create two cubes with the familiar `cube()` function, and we will select one with the mouse. When we select one of the cubes, the cubes will exchange colors. (NOTE: this may change, because it is very easy to change the behavior of the cubes' colors. We'll see what we eventually choose to do.

In this example, we start with a full `Mouse(...)` callback function, the `render(...)` function that registers the two cubes in the object name list, and the `DoSelect(...)` function that manages drawing the scene in `GL_SELECT` mode and identifying the object(s) selected by the position of the mouse when the event happened. Finally, we include the statement in the `main()` function that registers the mouse callback function.

```

    glutMouseFunc(Mouse);

    ...

    void Mouse(int button, int state, int mouseX, int mouseY)
    {
        if (state == GLUT_DOWN) { /* find which object was selected */
            hit = DoSelect((GLint) mouseX, (GLint) mouseY);
        }
        glutPostRedisplay();
    }

    ...

    void render( GLenum mode )
    {
        // Always draw the two cubes, even if we are in GL_SELECT mode,
        // because an object is selectable iff it is identified in the name
        // list and is drawn in GL_SELECT mode
        if (mode == GL_SELECT)
            glLoadName(0);
        glPushMatrix();
        glTranslatef( 1.0, 1.0, -2.0 );
        cube(cubeColor2);
        glPopMatrix();
        if (mode == GL_SELECT)
            glLoadName(1);
        glPushMatrix();
        glTranslatef( -1.0, -2.0, 1.0 );
        cube(cubeColor1);
        glPopMatrix();
    }

```

```

    glFlush();
    glutSwapBuffers();
}

...

GLint DoSelect(GLint x, GLint y)
{
    GLint hits, temp;

    glSelectBuffer(MAXHITS, selectBuf);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);

    // set up the viewing model
    glPushMatrix();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    // set up the matrix that identifies the picked object(s), based on
    // the x and y values of the selection and the information on the
    // viewport
    gluPickMatrix(x, windH - y, 4, 4, vp);
    glClearColor(0.0, 0.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    gluPerspective(60.0, 1.0, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //          eye point      center of view      up
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    render(GL_SELECT); // draw the scene for selection

    glPopMatrix();
    // find the number of hits recorded and reset mode of render
    hits = glRenderMode(GL_RENDER);
    // reset viewing model into GL_MODELVIEW mode
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //          eye point      center of view      up
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    // return the label of the object selected, if any
    if (hits <= 0) {
        return -1;
    }
    // carry out the color changes that will be the effect of a selection
    temp = cubeColor1; cubeColor1 = cubeColor2; cubeColor2 = temp;
    return selectBuf[3];
}

void main(int argc, char** argv)
{
    ...
    glutMouseFunc(Mouse);

    myinit();
}

```

```
    glutMainLoop();  
}
```

Example: mouse callback for mouse motion

This example shows the callback for the motion event. This event can be used for anything that uses the position of a moving mouse with button pressed as control. It is fairly common to see a graphics program that lets the user hold down the mouse and drag the cursor around in the window, and the program responds by moving or rotating the scene around the window. The program this code fragment is from uses the integer coordinates to control spin, but they could be used for many purposes and the application code itself is omitted.

```
void motion(int xPos, int yPos)  
{  
    spinX = (GLfloat)xPos;  
    spinY = (GLfloat)yPos;  
}  
  
int main(int argc, char** argv)  
{  
    ...  
    glutMotionFunc(motion);  
  
    myinit();  
    glutMainLoop();  
}
```

Source codes

The source codes for these examples may be found in the files

movingcube.c

slidingcube.c

menucube.c

choicecube.c

All of these use the cube() function found in the file

cube.c