Reading the Keyboard with Carbon Events

Author: Mark Szymczyk
Last Update: December 3, 2005

When writing games for Mac OS X, you can assume the player has two input devices: keyboard and mouse. You can't assume the player's mouse has more than one mouse button, which means if your game involves more than pointing and clicking, you must handle keyboard input from the player. An easy and efficient way to support the keyboard is with Carbon events. This article shows you how to use Carbon events to read the keyboard and provides an introduction to Carbon events for general application developers.

Introduction

Events are the lifeblood of most applications. When you press a key on the keyboard, move the mouse, make a menu selection, or resize a window, you're generating events. The Carbon Event Manager is what Carbon programs use to handle events on Mac OS X.

Event Targets

The building blocks of the Carbon Event Manager are event handlers, which are functions that handle events. The Carbon Event Manager has an event target containment hierarchy. At the top of the hierarchy is the application target, which can handle any event in the application. Below the application are the window and menu targets. The window target can handle any event that occurs in the window, and the menu target can handle any menu-related event. Below the window are control targets that can handle events involving that control.

There are event handlers for each level of the containment hierarchy. If you click a button in a window, the Carbon Event Manager sends the event to the button's event handler. If the button's event handler doesn't handle the click, the event moves to the window's event handler. If the window's event handler doesn't handle the click, the application event handler handles it.

Standard Event Handlers

Apple supplies standard event handlers for application and window targets. The standard application event handler deals with the user switching applications as well as menu events. If you choose About from the application menu, your program's about box opens without you having to write any code. The standard window event handler deals with the user moving, closing, minimizing, and resizing the window as well as standard behavior for controls.

These event handlers handle the most common cases, saving you a lot of work. All you have to do is install the standard event handlers and you get standard Mac OS X behavior.

What You Have to Write

You must write event handlers to handle events that are specific to your application and to handle events you want to handle differently than the operating system. An event handler is just a function. If you have application-specific menus and menu items, you must write an event handler to handle those menu item selections. Games normally require keyboard and mouse event handlers to deal with player input.

Your event handlers complement the standard handlers. The standard event handler handles any events your handler doesn't handle. Don't waste your time writing code that repeats the standard event handler's behavior. Only write code to handle your application's specific needs.

Installing the Standard Event Handlers

To use Carbon events in your application, you must call the function `RunApplicationEventLoop()`. This function installs the standard application event handler and dispatches events to event handlers.

```
void GameApp::EventLoop(void)
{
    RunApplicationEventLoop();
}
```

For a fullscreen game, calling `RunApplicationEventLoop()` is all you need to call to use the event handlers you write. Write event handlers for the keyboard and mouse and install a Carbon event timer (Read my article on Carbon event timers for more information) to run your game loop. Now you have a full-featured control scheme without writing a lot of code.

If you're using Mac OS X windows in your application, you should install the standard window event handler. Call `InstallStandardEventHandler()` and supply a target, which will be a window for a window event handler.

```
WindowRef window;
OSStatus error;

error = InstallStandardEventHandler(GetWindowEventTarget(window));
```

Installing Your Event Handler

You must install the event handlers you write to be able to use them in your application. Installing the event handler involves three steps.

1) Decide what type of event handler to write.
2) Specify the events you want the event handler to handle.
3) Call the appropriate function to install the handler.

Deciding What Type of Event Handler to Write

If you read the introduction, you may remember that Carbon events have four types of targets: application, window, menu, and control. The event handler you write must be for one of these four targets. The usual choices are application and window. If you're writing a fullscreen game, you'll be writing application event handlers. Window event handlers are more common in applications where you're doing a lot of work in windows, such as word processors, spreadsheets, board games, card games, and strategy games.

Specifying the Events to Handle

To specify the events you want to handle, create an array of type `EventTypeSpec` (or a variable if you want to handle only one event). The `EventTypeSpec` data structure has two components: an event class, and an event type. Common event classes include keyboard, mouse, text input, command (for menu selections), window, and application. The event type specifies the actual event. Some common mouse events include mouse movement, mouse button down, and scroll wheel movement. The following declaration:

```
EventTypeSpec keyboardHandlerEvents =
    { kEventClassKeyboard, kEventRawKeyDown };
```

Tells the handler to handle key down events, which occur when the user presses a key on the keyboard. Key down events are what you're most interested in for game development.

Installing the Handler

To install an application event handler, call the function `InstallApplicationEventHandler()`. This function takes five arguments.

- The event handler you're installing. Generally you call the function `NewEventHandlerUPP()` and supply the name of your event handler function.
- The number of event types the handler will handle.
- The events the handler will handle.
- Data you want to pass to the event handler.
- A data reference to the event handler. You can pass `NULL`, but if you want to add events to or remove events from the event handler, you'll need the data reference.

The following code provides an example of installing an application event handler in C++:

```
void GameApp::InstallKeyboardEventHandler(void)
{
    EventTypeSpec keyboardHandlerEvents =
      { kEventClassKeyboard, kEventRawKeyDown };

    InstallApplicationEventHandler(NewEventHandlerUPP
      (KeyboardEventHandler), 1, &keyboardHandlerEvents,
      this, NULL);

}
```

If you want to install a window event handler, call the function `InstallWindowEventHandler()`. Its first argument is the window you want to attach the event handler to. The remaining arguments are the same as the arguments to `InstallApplicationEventHandler()`. The functions `InstallMenuEventHandler()` and `InstallControlEventHandler()` install menu and control event handlers respectively. They work similarly to `InstallWindowEventHandler()` but take a menu or a control as the first argument instead of a window.

Writing the Event Handler

Now it's time to write the event handler. Event handlers take the form.

```
pascal OSStatus EventHandler(EventHandlerCallRef myHandlerChain,
    EventRef event, void* userData)
```

The `pascal` keyword tells the compiler to use Pascal language parameter passing conventions. Event handling functions require the Pascal language conventions. The `myHandlerChain` argument contains the hierarchy of event handlers that could handle this particular event. Use the `myHandlerChain` argument if you want to call another event handler in your event handler function. I won't be using the `myHandlerChain` argument in this article. The `event` argument is the event you're passing to the handler. The `userData` argument is the data you supplied when you installed the event handler.

C++ programs that want event handlers to be members of a class must declare the timer to be a static function in the header file.

```
static pascal OSStatus EventHandler(EventHandlerCallRef myHandlerChain,
    EventRef event, void* userData);
```

You can give your event handler function any name you want, but the function name must match the name you supply to the function `NewEventHandlerUPP()` when installing the event handler.

Determining Which Key the User Pressed

The code you write for an event handler depends on the types of events you want to handle. For a keyboard event handler you must declare a variable of type `UInt32` (unsigned 32-bit integer) to store the key the user pressed. Call the function `GetEventParameter()` to get the key the user pressed. `GetEventParameter()` takes six arguments.

- The event.
- The name of the parameter you want to get. For a keyboard event, you normally want `kEventParamKeyCode`, which is the virtual keycode for the key the user pressed.
- The parameter type you want `GetEventParameter()` to return. For the keyboard the desired parameter type is `typeUInt32`.
- The actual parameter type `GetEventParameter()` returns. You can get away with passing `NULL`.
- The desired size of the data `GetEventParameter()` returns. For the keyboard the size will be four bytes, the size of a `UInt32` variable.
- The actual size of the data `GetEventParameter()` returns. You can get away with passing `NULL`.
- The data `GetEventParameter()` returns. For the keyboard the data is the key the user pressed.

Virtual keycodes require some more explanation. A virtual keycode identifies a key on the keyboard. Using the virtual keycode allows you to separate the physical key from its character. Separating the physical key from its character makes international support for your application easier. Not every Mac user has the American QWERTY keyboard layout. Suppose you want to use the W key to move the player up in your game. By using virtual keycodes, everybody presses the same key, whether it's the W key on an American keyboard or the Z key on a French keyboard.

Event Handler Example

After calling `GetEventParameter()`, write a `switch` statement with the keys you want your program to handle. Let's look at a simple example of a keyboard event handler.

```
pascal OSStatus GameApp::KeyboardEventHandler(
    EventHandlerCallRef myHandlerChain,
    EventRef event, void* userData)
{
    UInt32 keyPressed;
    OSStatus result;

    GameAppPtr currentApp = (GameAppPtr)userData;

    GetEventParameter(event, kEventParamKeyCode, typeUInt32, NULL,
                      sizeof(UInt32), NULL, &keyPressed);

    switch (keyPressed) {
        case kEscapeKey:
            currentApp->Pause();
            result = noErr;
            break;

        default:
            // The standard application event handler
            // handles any events our event handler
            // doesn't handle.
            result = eventNotHandledErr;
            break;
    }

    return result;
}
```

The event handler pauses the game when the player press the Escape key. Notice how the event handler returns `eventNotHandledErr` if the user presses a key the event handler doesn't handle. This is very important. If your event handler does not handle an event, you must return `eventNotHandledErr`. Returning `eventNotHandledErr` dispatches the event to the standard event handler to handle. Returning `noErr` instead of `eventNotHandledErr` can cause your event handler to work improperly.

Cleaning Up

To stop the application event loop, call the function `QuitApplicationEventLoop()`. The operating system calls `QuitApplicationEventLoop()` automatically if you tag the Quit menu item with the `kHICommandQuit` menu command ID. You can tag the Quit menu item in Interface Builder.

When the user quits your program, you may want to remove the event handlers you installed and the event handler universal procedure pointers (UPP) you created when you installed the event handlers. Call `RemoveEventHandler()` to remove an event handler. If you passed `NULL` as the event handler's data reference when installing the event handler, you won't be able to remove the event handler. Not being able to remove the event handler is why I used the term "may want to" instead of "should" at the start of the paragraph. Call `DisposeEventHandlerUPP()` to dispose of any event handler UPPs you created by calling `NewEventHandlerUPP()`.

If you want your code to be perfect, store your event handler's data reference and UPP variables. Remove your event handlers and dispose of your UPPs when the user quits your program. However, leaving your event handlers and UPPs alone is not a big problem. Technically, you have memory leaks if you don't remove the event handlers and dispose of the UPPs. But the operating system disposes of all allocated memory when the user quits your application. Event handlers normally stay active for the entire time your program runs so the memory leaks aren't a problem. If you were creating thousands of event handlers that stayed in effect for a short time, you would need to remove the handlers and dispose of the UPPs.

Conclusion

This article introduced you to Carbon events, but no single article can cover everything there is to know about them. For more information on Carbon events, read the *Carbon Event Programming Guide* and *Carbon Event Reference* documents. You can read these documents at Apple's developer site or in Xcode by choosing Help > Documentation. These documents are in the Carbon documentation section under Events and Other Input.