Black Art of 3D Game Programming: Writing Your Own High-Speed 3D Polygon Video Games in C

Black Art of 3D Game Programming: Writing Your Own High-Speed 3D Polygon Video Games in C by Andre Lamothe.

Contents

1 Introduction

2 Part 1: Incantations

3 Part 2: Alchemy

4 Part 3: Spells

Introduction

- Black Art of 3D Game Programming, About the Author
- Black Art of 3D Game Programming, Acknowledgments
- Black Art of 3D Game Programming, Foreward
- Black Art of 3D Game Programming, Preface
- Black Art of 3D Game Programming, Installation

Part 1: Incantations

- Black Art of 3D Game Programming, Chapter 1: The Genesis of 3D Games
- Black Art of 3D Game Programming, Chapter 2: Elements of the Game

Part 2: Alchemy

- Black Art of 3D Game Programming, Chapter 3: The Mysterious VGA Card
- Black Art of 3D Game Programming, Chapter 4: Waking the Dead with Animation
- Black Art of 3D Game Programming, Chapter 5: Communicating with the Outside World

- Black Art of 3D Game Programming, Chapter 6: Dancing with Cyberdemons
- Black Art of 3D Game Programming, Chapter 7: The Magic of Thought
- Black Art of 3D Game Programming, Chapter 8: The Art of Possession
- Black Art of 3D Game Programming, Chapter 9: Multiplayer Game Techniques
- Black Art of 3D Game Programming, Chapter 10: 3D Fundamentals
- Black Art of 3D Game Programming, Chapter 11: Building a 3D Graphics Engine
- Black Art of 3D Game Programming, Chapter 12: Solid Modeling and Shading
- Black Art of 3D Game Programming, Chapter 13: Universal Transformations
- Black Art of 3D Game Programming, Chapter 14: Hidden Surface and Object Removal
- Black Art of 3D Game Programming, Chapter 15: Clipping and Rendering the Final View
- Black Art of 3D Game Programming, Chapter 16: Voxel Graphics

Part 3: Spells

- Black Art of 3D Game Programming, Chapter 17: Optimizing the 3D Engine
- Black Art of 3D Game Programming, Chapter 18: Kill or Be Killed
- Black Art of 3D Game Programming, Appendix A: Cybersorcerer and Cyberwizard Contests

---

Copyright 2006 Andre LaMothe

Black Art of 3D Game Programming, Chapter 5: Communicating with the Outside World

At this point in the game we've learned how to control the PC and the VGA to do our bidding. We can generate incredible imagery that links us to the netherworld of Cyberspace. However, we are missing one crucial element, and that is the ability to communicate with the world of the living. In other words, we must learn how to interpret and control the various input devices that the humans playing our games may connect to their PCs. In this chapter we'll cover the keyboard, joystick, and mouse and write a complete software library to communicate with each device in an effortless manner.

Contents

1 The Ins and Outs of Input Devices

2 The Keyboard

3 Reading Keys with BIOS and C

- 3.1 Listing 5-1 Reading an ACSII key using C and BIOS

4 Scan Codes

- 4.1 Listing 5-2 Reading the scan code of a key

5 Make and Break Codes

6 The Shift State

- 6.1 Listing 5-3 Function to read the shift state of the keyboard

7 Tracking Multiple Keypresses

8 Writing the Keyboard Driver

- 8.1 Installing the Driver
  - 8.1.1 Listing 5-4 Function to install the new keyboard driver
- 8.2 Removing the Driver
  - 8.2.1 Listing 5-5 Function to remove the keyboard driver

9 The New Keyboard Driver

- 9.1 Listing 5-6 The new keyboard driver

10 Keying in the Right Combination

- 10.1 Listing 5-7 A program to demonstrate the keyboard driver

11 Joysticks

12 Reading the Buttons

12.1 Listing 5-8 Function to read the buttons of the joysticks
13 Computing the Position of the Stick
13.1 Listing 5-9 Reading a joystick on the PC
13.2 Listing 5-10 A joystick position function based on BIOS
14 Calibrating the Joystick
14.1 Listing 5-11 A joystick calibration function
15 Joystick Detection Techniques
15.1 Listing 5-12 Detecting the presence of a joystick
16 Driving the Stick Crazy
16.1 Listing 5-13 A joystick demonstration program
17 The Mouse
18 Talking to the Mouse
19 A Mouse Interface Function
19.1 Listing 5-14 A mouse interface function
20 Point and Squash!
20.1 Listing 5-15 A demo that shows a good use of the mouse!
21 Creating an Input System
22 Input Device Conditioning
23 Summary
24 Continue

The Ins and Outs of Input Devices

There are so many input devices today that it would be practically impossible to list and explain all of them. Nevertheless, they have many attributes in common and hence can be collected into distinct groups. Each group has a device or two that can be thought of as the most representative of that grouping. Thus, we will focus on the most common input devices used in games on the PC. Granted, many people may have flight sticks, track balls, or flying mice, but each of these devices functions similarly to the devices that we will cover.

As the player manipulates an input device, he is trying to tell the computer to do what he is thinking. Of course, much is lost in the translation. Many input devices are fairly crude in design and can't translate concepts such as "english" or finesse. This will improve as time marches forward; however, we must make the best use of the input devices that we have at our disposal to allow the player to control the game as closely as possible.

The keyboard is the most common of all input devices, and you can be almost certain that every PC has one. Amazingly enough, the keyboard is the most complex input device of all. It contains a simple processor and has a very complex design. The next and probably simplest input device is the joystick. It's uncomplicated in construction and is used mostly in simulations or flight games. Today, you can bet that most game players have a joystick, but it's still not guaranteed like the keyboard. Therefore, don't write a game that only uses the joystick for input, or else you'll be in trouble! The final and most un-game-like input device is the mouse. The mouse has no real mapping to the world of video game inputs other than a pointing device. And since pointing at something to select it is not frequently done in most games (except for targeting maybe), mice usually find their best use in games as user interface controllers and pseudotrackballs. As far as betting on a PC having a mouse, it's probably a good bet. Most people on Earth now have MS Windows/Windows 95 or some other window-based GUI, and these GUIs commonly use a mouse; so writing a game that is specifically targeted for a mouse is safer than a joystick, but still not wise in all cases. Now let's discuss each input device in detail.

The Keyboard

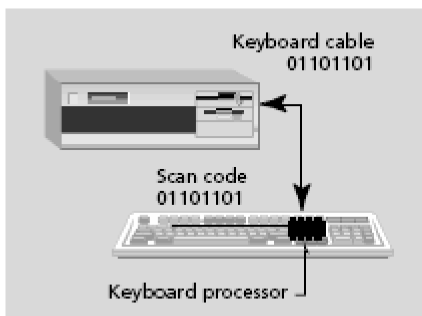

Figure 1 - The hardware layout of the keyboard

We're going to begin with the hardest input device and finish with the easiest input device, so let's begin with the keyboard. The keyboard is such a complex input device, I'm surprised there aren't a million reference books on the topic. However, the truth is, there aren't. Maybe the reason for this is that the PC keyboard is used commonly as a text input device for programs and not a real-time video game input device. Using the keyboard for a text input device is easily accomplished with C or BIOS calls and, accordingly, there's not much to it. The problems arise when we wish to take the keyboard over and use it as a set of mutually exclusive switches, much like a control panel on an arcade or console game machine.

The keyboard is basically a small computer with the single mission in life of detecting keypresses, encoding them, and sending the information serially to the host PC. Figure 5-1 shows the hardware layout of the keyboard setup. When the keyboard detects a keypress, the keyboard processor looks up the key's scan code in a table and encodes it with some extra information. (The scan code is a unique bit pattern representing the key; it is different from the ASCII code.) The information is then sent to the PC serially over the keyboard cable into a buffer area within the PC. Furthermore, this event causes an interrupt to occur. The processor, in response to the interrupt signal, reads the scan code from the keyboard buffer area, converts the scan code to ASCII, and stores the results in a larger software buffer. This buffer is the standard DOS 13-character look-ahead buffer that I'm sure you're used to.

The translation from scan code to ASCII code is very important. Many people don't realize that the keyboard doesn't send ASCII codes, but it sends low-level raw information called scan codes. These scan codes are what our keyboard driver will be primarily interested in obtaining. Also, a keypress is really two events. A scan code is sent when a key is pressed, and another related scan code is sent when the key is released. These two codes are referred to as the make code and break code and are both required to determine when a key is being held down and released and so forth.

This will all make sense in a while, but now you have the big picture, so let's start painting finer strokes.

Reading Keys with BIOS and C

For a real-time video game to obtain keyboard input and player commands in a clean and crisp manner, the C and BIOS input functions are hardly appropriate. However, they do have their uses in simple situations where real-time or multiple keypresses aren't necessary, so it's worthwhile to see how they work. The first and simplest input technique is to use *getch( )*. This function allows the user to input a character without pressing [ENTER] An example use of *getch( )* could be to control the direction of a ship. Take a look at this fragment:

```
direction = getch();

if (direction=='R')
    x++;
if (direction=='L')
    x--;
```

The above fragment is conceptually in the ballpark, but it lacks many important features. Obviously, we are trying to move a ship or something that uses the variable x as one of its coordinates. The user's input is obtained via getch( ) followed by a conditional statement on the result of the function call. But there are two glaring problems. First, the *getch( )* function waits for an input, and hence the game will get stuck at *getch( )* if there isn't an input key ready. This isn't acceptable because the whole game would stop. The second problem is that the *getch( )* function retrieves the ASCII code of the key pressed and

not the scan code; hence, depending on the state of the [CAPSLOCK] key, the program may or may not work since the test is done using the uppercase version of the [R] and [L] keys. Let's see if we can improve our dilemma a bit. There are a couple of ways supported by most C libraries to detect whether a key has been pressed. One such way is the function called *kbhit( )*, which returns a true if there is a key buffered and ready to be retrieved. Let's add this function, along with tests for both upper and lowercase characters, to our little program and see what we gain.

```
if (kbhit())
   {
   direction = getch();

   if (direction=='R' || direction=='r')
       x++;
   if (direction=='L' || direction=='l')
       x--;

   } // end if keyboard hit
```

The above fragment indeed solves two of our problems. It will not lock up the system if there isn't a key waiting, and both the uppercase and lowercase versions of the input characters are tested, so the state of the [CAPS{LOCK] key is no longer a problem. However, a problem still exists that may not seem obvious, but is nevertheless there. The problem is that only one key can be pressed at once! In other words, what happens if the player is turning and firing at the same time? Well, the program we wrote wouldn't work and the player would either turn or fire, but not both. We need a way to read multiple keypresses as if the keyboard were a set of switches indexed by a table. With this structure, simple tests can be made to see if any key is currently active.

We have just seen how C functions can be used to access the keyboard, but doesn't BIOS have a better interface or something that can help us with the above problems? Not really–the C functions are based on the BIOS functions. Alas, there is nothing new, but we might as well create a clean function to retrieve a single key from the keyboard before we move on. The function will use a couple C functions that are based on BIOS functions. These functions have similar functionality to the *kbhit( )* and *getch( )*, but are implemented using a direct interface to BIOS via C. Listing 5-1 contains a function to read a single key.

Listing 5-1 Reading an ACSII key using C and BIOS

```
unsigned char Get_Key(void)
{
// test if a key press has been buffered by system and if so, return the
ASCII
// value of the key, otherwise return 0
```

```
if (_bios_keybrd(_KEYBRD_READY))
 return((unsigned char)_bios_keybrd(_KEYBRD_READ));
else return(0);

} // end Get_Key
```

First off, the above function and everything else in this chapter are in the files BLACK5.C and BLACK5.H, which can be found in the directory for this chapter. The above function is based on the _bios_keybrd( ) C function, which is a clean interface to the BIOS keyboard interrupt INT 16h through C. We can use Get_Key( ) in the main event loop of a game as long as we aren't expecting multiple keypresses. However, the function is still a bit compiler dependent and uses BIOS, which is slow.

Scan Codes

We've learned how to retrieve and test a key. However, we saw that there can be a bit of a problem testing the upper or lowercase version of a key without extra logic. There is another more elegant solution to the problem: take out the middle man and retrieve the actual scan code of a key before it has been translated by the PC's software into an ASCII code. By reading scan codes instead of ASCII codes, we are in essence reading the lowest level of keyboard information, and since there is a one-to-one correspondence between scan codes and keyboard keys, the concept of upper and lowercase is irrelevant. For example, if the scan code for the [J] key is sent, regardless of the state of the [SHIFT] keys or [CAPS{LOCK], the scan code will always be 36. This level of abstraction is much closer to our final goal of creating a set of switches like that of an arcade or console machine. We can think of a key's scan code as the value that represents that physical key instead of a value that represents the symbol on the key.

Function 00h: Read character from keyboard buffer.
Entry: AH = 00h
Exit: AH—Contains the keyboard scan code.
AL—Contains the keyboard ASCII code.

Function 01h: Get keyboard buffer status.
Entry: AH = 01h
Exit: ZF, Zero Flag = 1 means character ready else none ready.
AH—Contains the keyboard scan code.
AL—Contains the keyboard ASCII code.

Function 02h: Get keyboard shift byte.
Entry: AH = 02h
Exit: AL = Status of shift state as detailed below:

Bit Number	*Meaning*
0	Right [SHIFT] is down.
1	Left [SHIFT] is down.
2	[CONTROL] is down.
3	[ALTERNATE] is down.
4	[SCROLL LOCK] is on.
5	[NUM LOCK] is on.
6	[CAPS LOCK] is on.
7	[INSERT] is on.

Table 1 – The BIOS keyboard interrupt 16h

So how do we read scan codes? The easiest way is to again use BIOS and make a couple function calls that test if a key is ready and retrieve the scan code of the sent key. The BIOS keyboard interrupt we'll use is *INT 16h* and has the subfunctions as shown in Table 5-1.

We'll use function 0 to determine if a key is ready and function 1 to actually read the scan code of the key. A complete C function to do this is shown in Listing 5-2.

Listing 5-2 Reading the scan code of a key

```
unsigned char Get_Scan_Code(void)
{
// use BIOS functions to retrieve the scan code of the last pressed key
// if a key was pressed at all, otherwise return 0

_asm
   {
   mov ah,01h ; function #1 which is "key ready"
   int 16h ; call the BIOS keyboard interrupt
   jz buffer_empty ; if there was no key ready then exit
   mov ah,00h ; function #0: retrieve raw scan code
   int 16h ; call the BIOS keyboard interrupt
   mov al,ah ; result is placed by BIOS in ah, copy it to al
   xor ah,ah ; zero out ah
   jmp done ; jump to end so ax doesn't get reset

buffer_empty:
     xor ax,ax ; a key was retrieved so write a 0 into ax to reflect this
done:
    } // end asm

// 8 or 16 bit data is returned in AX, hence no need to explicitly say
return()

} // end Get_Scan_Code
```

The *Get_Scan_Code( )* function works similarly to the *Get_Key( )* function as far as the interface goes. If a key is ready, the scan code of the key will be retrieved, otherwise, a 0 will be returned. The function has two distinct phases to it. The first phase sets up the parameters for the function 1 call of *INT 16h* (which tests if a key is ready) and then makes a call to the function. Then, based on the results of function 0 (which are placed in the *ZF* flag), a jump is either made to exit the program returning 0, or the remainder of the function executes, retrieving the scan code of the buffered key.

Now that we have a function to read scan codes from the keyboard, we can use these scan codes in the game logic to determine which key(s) is being pressed. However, what are the scan codes for all the keys of the PC's keyboard? This is a good question and deserves a good answer. All the scan codes can be found either by looking in some reference or by pressing each key on the keyboard to produce its scan code and making a list of the codes. The latter method is a bit primitive, and I do indeed have a list for

you; but let's wait a moment before reviewing it, so we can see the scan codes for both the make codes and the break codes.

At this point, we have addressed all the issues associated with reading single keys, but we still need to solve the problem of reading multiple keys. This is an absolute must in a video game. It's now time to cast the Cybersorcery spell to solve it!

Make and Break Codes

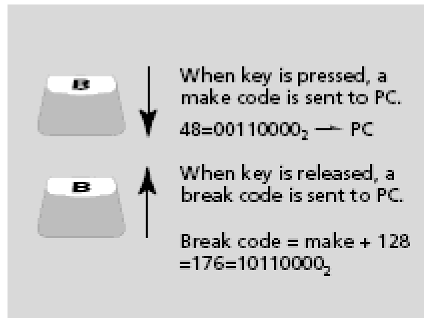

Figure 2 - The relationship between make codes and break codes.

At the beginning of the chapter we discussed that when a key is pressed, a scan code called the make code is produced, and when the key is released, a break code is produced (it is related to the make code by setting one bit). By tracking make and break codes, we can create a table in memory that contains the state of each key on the keyboard. The make codes produced by the PC's keyboard are probably arbitrary, but they are the same on every keyboard. They are set up in a way that if you were to scan across the keyboard from top to bottom, left to right, the make codes in most cases increment numerically. But we don't need to know the rhyme or reason behind the selection of the values of the make codes for each key, we just need to know them and store them in a table.

The break codes are the interesting part. When a key is released, the keyboard sends a break code. The break code is related to the make code for the particular key by changing the high bit of the make code. Specifically, break codes are make codes with the number 128 added to them. Figure 5-2 shows this graphically. When a key is pressed, a make code is sent, and then when the same key is released (sometime in the future), the break code is sent. Hence, if we have a table of the make codes for the keyboard, we can build or compute any break code with the simple rule of adding 128. In the BLACK5.H header file there is a complete list of all the make codes and break codes. Here are the make codes for you to review:

```
// these are the scan codes for the keys on the keyboard, note they are all
// from 0-127 and hence are the "make" scan codes, 128-255 are for the break
// scan codes and are computed simply by adding 128 to each of the make codes
#define MAKE_ESC 1
#define MAKE_1 2
#define MAKE_2 3
#define MAKE_3 4
#define MAKE_4 5
#define MAKE_5 6
#define MAKE_6 7
#define MAKE_7 8
#define MAKE_8 9
#define MAKE_9 10
```

```
#define MAKE_0 11
#define MAKE_MINUS 12
#define MAKE_EQUALS 13
#define MAKE_BKSP 14
#define MAKE_TAB 15
#define MAKE_Q 16
#define MAKE_W 17
#define MAKE_E 18
#define MAKE_R 19
#define MAKE_T 20
#define MAKE_Y 21
#define MAKE_U 22
#define MAKE_I 23
#define MAKE_O 24
#define MAKE_P 25
#define MAKE_LFT_BRACKET 26
#define MAKE_RGT_BRACKET 27
#define MAKE_ENTER 28
#define MAKE_CTRL 29
#define MAKE_A 30
#define MAKE_S 31
#define MAKE_D 32
#define MAKE_F 33
#define MAKE_G 34
#define MAKE_H 35
#define MAKE_J 36
#define MAKE_K 37
#define MAKE_L 38
#define MAKE_SEMI 39
#define MAKE_APOS 40
#define MAKE_TILDE 41
#define MAKE_LEFT_SHIFT 42
#define MAKE_BACK_SLASH 43
#define MAKE_Z 44
#define MAKE_X 45
#define MAKE_C 46
#define MAKE_V 47
#define MAKE_B 48
#define MAKE_N 49
#define MAKE_M 50
#define MAKE_COMMA 51
#define MAKE_PERIOD 52
#define MAKE_FORWARD_SLASH 53
#define MAKE_RIGHT_SHIFT 54
#define MAKE_PRT_SCRN 55
#define MAKE_ALT 56
#define MAKE_SPACE 57
#define MAKE_CAPS_LOCK 58
```

```
#define MAKE_F1 59
#define MAKE_F2 60
#define MAKE_F3 61
#define MAKE_F4 62
#define MAKE_F5 63
#define MAKE_F6 64
#define MAKE_F7 65
#define MAKE_F8 66
#define MAKE_F9 67
#define MAKE_F10 68
#define MAKE_F11 87
#define MAKE_F12 88
#define MAKE_NUM_LOCK 69
#define MAKE_SCROLL_LOCK 70
#define MAKE_HOME 71
#define MAKE_UP 72
#define MAKE_PGUP 73
#define MAKE_KEYPAD_MINUS 74
#define MAKE_LEFT 75
#define MAKE_CENTER 76
#define MAKE_RIGHT 77
#define MAKE_KEYPAD_PLUS 78
#define MAKE_END 79
#define MAKE_DOWN 80
#define MAKE_PGDWN 81
#define MAKE_INS 82
#define MAKE_DEL 83
```

There is a similar table in BLACK5.H that contains the break codes. The defines are all prefixed by BREAK.

Using the above table along with the break code table we can now compute whether any key is down or up, but we still can only gain access to one keypress. The problem is we need to get closer to the hardware and talk directly to the keyboard. This means that we must install a new keyboard interrupt service routine to take over the information transport from the keyboard to the PC. But before we do this, we need to take a look at a possible problem with using scan codes instead of ASCII codes.

The Shift State

Scan codes are great, and the make and break codes will give us all the information we need to track whether any key is down or up. However, what if we want to know the difference between an uppercase or lowercase letter? Or what if the user wants to type * instead of 8? This level of differentiation can only be achieved with the shift keys. In general, the PC has a few keys that further qualify the meaning of the standard keys on the keyboard, such as the shift keys [SHIFT], control keys [CTRL], the alternate keys [ALT], and other miscellaneous keys such as [SCROLLLOCK], [NUMLOCK], [SYSREQ], and so forth. Many times, these keys are pressed simultaneously with the standard keyboard keys to arrive at some desired combination.

The keyboard takes care of these keys in a special way and places the state of the keys in a bitmapped register called the *shift state*. By reading the shift state and the scan codes, a program can determine the key that is being pressed and whether it's being further qualified by a special control key. The shift state can be read in two ways: using a C call or directly with BIOS. If you look back a couple pages to Table 5-1, you'll notice that Function 02h reads the shift state of the keyboard. If you wish, you can use this technique. However, the C compiler provides a clean interface to this function, so we will write a quick function that can retrieve the shift state of the keyboard and test if any particular key is down or activated, such as [NUM{{LOCK]. The function will simply retrieve the shift state from the keyboard by using the *_bios_keybrd( )* function. However, it would be nice to have a set of defines that make the decoding of the bit vector easy. Hence, I have created a set of defines in BLACK5.H that have the following values:

```
// bitmasks for the "shift state"

// note there is a difference between "on" and "down"

#define SHIFT_RIGHT          0x0001 // right shift
#define SHIFT_LEFT           0x0002 // left shift
#define CTRL                 0x0004 // control key
#define ALT                  0x0008 // alternate key
#define SCROLL_LOCK_ON       0x0010 // the scroll lock is on
#define NUM_LOCK_ON          0x0020 // the numeric lock is on
#define CAPS_LOCK_ON         0x0040 // the capitals lock is on
#define INSERT_MODE          0x0080 // insert or overlay mode
#define CTRL_LEFT            0x0100 // the left control key is pressed
#define ALT_LEFT             0x0200 // the left alternate key is pressed
#define CTRL_RIGHT           0x0400 // the right control key is pressed
#define ALT_RIGHT            0x0800 // the right alternate key is pressed
#define SCROLL_LOCK_DOWN     0x1000 // the scroll lock key is pressed
#define NUM_LOCK_DOWN        0x2000 // the numeric lock is pressed
#define CAPS_LOCK_DOWN       0x4000 // the capitals lock key is pressed
#define SYS_REQ_DOWN         0x8000 // the system request key is pressed
```

If you refer back to the BIOS *INT 16h* function and take a look at Function 03h, you'll notice that there are only eight bits defined; hence, the C version of the shift state function has quite a bit more functionality. It accomplishes this by looking a little deeper than the BIOS function does to compute the complete shift state of the PC.

Now let's write a function that takes as a parameter one of the above constants and then returns a true if that particular bit is on. Listing 5-3 contains the function to do this.

Listing 5-3 Function to read the shift state of the keyboard

```
unsigned int Get_Shift_State(unsigned int mask)
{
// return the shift state of the keyboard logically ANDed with the sent mask

return(mask & _bios_keybrd(_KEYBRD_SHIFTSTATUS));

} // end Get_Shift_State
```

As you can see, the function is hardly complex since it is based on the C function _bios_keybrd( ). In any case, if we wanted to use the function to test whether the right alternate key was on, we could write something like this:

```
if (Get_Shift_State(ALT_RIGHT))
   {
   //.. do whatever
   } // end if right alternate on
```

We can test the other shift states in a similar manner.

We are getting very close to our final goal. We can read scan codes and make codes, compute break codes, and read the shift state to figure out exactly what the player intends. Now it's time to write a complete keyboard driver!

Tracking Multiple Keypresses

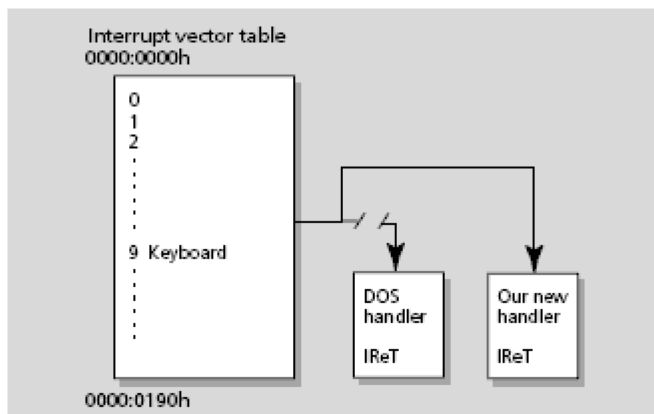

Figure 3 - Replacing the system keyboard interrupt

Tracking multiple keys is not complicated as long as we know where to look for the keys. Unfortunately, we can't wait for DOS to respond to a keypress, we must remove the DOS keyboard handler and install one of our own. This is necessary because we must know when a keypress or release occurs. This can only be done effectively if the interrupt that is generated during a keyboard event is used to call our new keyboard handler. Interrupt service routines and device drivers is a topic we will cover in depth later in the book, but for now we are going to at the very least install a new keyboard driver that will do the following:

- When key is pressed or released, the keyboard driver will update a global table and set the appropriate table element to true or false reflecting the state of the key.

- After the update is complete, the driver will reset the interrupt controller and return to the caller of the interrupt (usually DOS).
- During the process, the number of active keys will be tracked in a variable.

This variable will be used to facilitate more efficient keyboard testing logic.

The hard part of all this is writing an interrupt service routine (ISR) to replace the standard DOS keyboard ISR. Figure 5-3 shows what needs to be done. How do we replace the standard DOS keyboard interrupt with ours? Well, there is a table located in the first 400 bytes of the PC's memory that contains the interrupt vectors for all the interrupts the PC can respond to. All we need to know now is that the keyboard interrupt is INT 09h and can be found at offsets 24h to 27h in the table (each interrupt vector is 4 bytes).

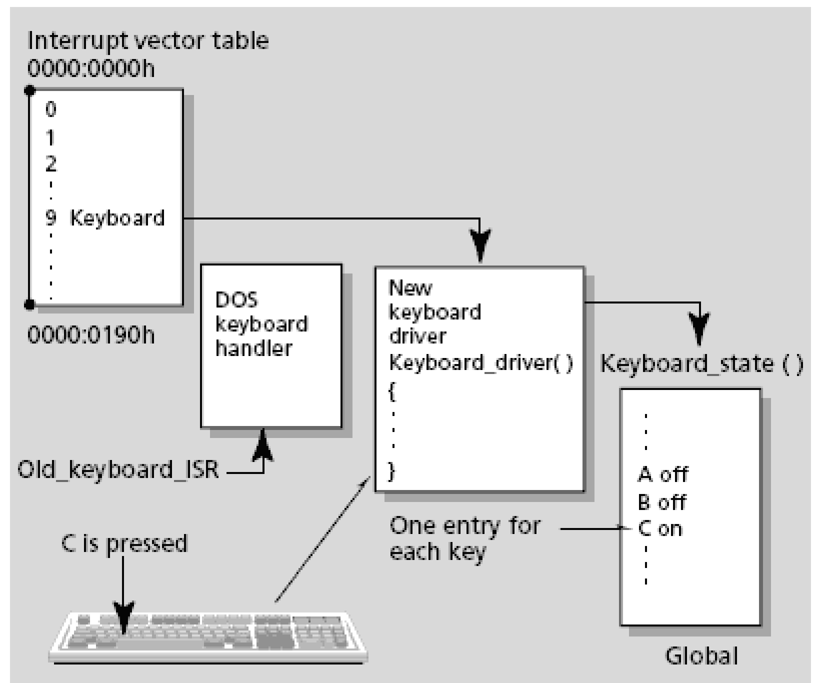

Figure 4 - Architecture of the keyboard system.

By pointing the keyboard vector to a new function (such as ours), we can take over the keyboard completely and do whatever we wish with the data. Let's review the game plan to install a new keyboard driver ISR. We will first write a new keyboard driver that will retrieve a key from the proper data port and then update a global table that can be accessed by other functions. We will then save the old keyboard ISR at interrupt vector 09h in the interrupt table and install our own. Then to test if a key or keys are pressed, we won't make a call to any keyboard function, we will simply refer to the table. Figure 5-4 shows the architecture of our new keyboard driver and its relationship to the game application.

Writing the Keyboard Driver

There are a lot of details that we need to consider when writing a new keyboard driver ISR, such as how to begin, where to look for the keyboard data, the control of the interrupt controller, and so forth. However, it turns out that all of these details aren't too bad if we tackle them one at a time. Let's begin with how to write an interrupt handler in C.

Luckily for us, the C language has had a few extensions added to it, allowing us to change interrupt vectors in the interrupt table, read any particular interrupt vector, and finally, to write complete ISRs in

C! Let's begin with the latter. To write an interrupt function, we must heed a couple of rules: the interrupt handler must be short, and it must have the proper prolog and epilog code to save the CPU registers and set up the stack. The C language has a new keyword that makes this easy, and it's called *_interrupt*. If we place this keyword in the definition of a function, it directs the compiler to make the function an interrupt and will force the proper prolog, epilog, and return code to make the function perform as a standard ISR. For example, a simple ISR that does nothing can be defined as:

```
void _interrupt _far ISR()
{
// ... interrupt code

} // end ISR
```

That's it! The function takes no parameters, but when installed in the interrupt table, it will be called when the appropriate interrupt occurs. Of course, the hard part of writing the driver is the code in between the braces, but we're getting there! We will hold off on writing the function for a bit; first, let's see exactly how we will install and remove the new driver from the interrupt table.

Installing the Driver

Before we install the new keyboard driver, we need a place to save the old interrupt vector that points to the standard DOS keyboard interrupt. Later, we will use this saved vector to restore the DOS keyboard handler when our game or program has completed. The variable that will hold the old keyboard handler is defined below:

```
void (_interrupt _far *Old_Keyboard_ISR)(); // holds old keyboard
                                             // interrupt handler
```

As you can see, it is a pointer to a function that is both *FAR* and *_interrupt*. You will also note that the function takes no parameters. To further facilitate the installation of our new keyboard driver, here are a few defines to make the code more readable:

```
// defines for the keyboard interface

#define KEYBOARD_INTERRUPT    0x09 // the keyboard interrupt number
#define KEY_BUFFER            0x60 // the port of the keyboard buffer
#define KEY_CONTROL           0x61 // the port of the keyboard controller
#define PIC_PORT              0x20 // the port of the peripheral
                                   // interrupt controller (PIC)
```

Only *KEYBOARD_INTERRUPT* is of interest at this point. It is simply aliased to the number 09h, which is the keyboard interrupt vector number that is the 9th entry in the interrupt table. The remaining defines are to communicate with the keyboard buffer and control logic, which we'll get to shortly.

We're about ready to write a function to install the new keyboard driver, but we don't know its name yet! So we'll call it *Keyboard_Driver( )*, and here's its formal prototype:

```
void _interrupt _far Keyboard_Driver();
```

Notice it has no parameters. This is typical of ISRs since they are not called by an application, but started by an event; therefore, there's really no need for parameters. We now have all the elements to install the new keyboard driver. To install it, we'll use the C function *_dos_getvect( )* to first save the old DOS keyboard handler, and then we'll install our new handler with *_dos_setvect( )*. Listing 5-4 contains the function to install the new keyboard driver.

Listing 5-4 Function to install the new keyboard driver

```
void Keyboard_Install_Driver(void)
{
// this function installs the new keyboard driver

int index;

// clear out keyboard state table
for (index=0; index<128; index++)
     keyboard_state[index]=0;

// save the old keyboard driver
Old_Keyboard_ISR = _dos_getvect(KEYBOARD_INTERRUPT);

// install the new keyboard driver
_dos_setvect(KEYBOARD_INTERRUPT, Keyboard_Driver);

} // end Keyboard_Install_Driver
```

The C interrupt functions need a little explanation. The *_dos_setvect( )* function takes as parameters the interrupt number to change along with a pointer to the new interrupt. Likewise, *_dos_getvect( )* takes as a parameter the interrupt number to retrieve and returns a pointer to it. You can refer to your C compiler's run-time library reference for a more complete explanation of these and other related

functions. About the only question we should be asking is, why are we using a C function to install the driver at all? Why not index into the memory from 0000:0000h to 0000:0190h and do it ourselves? The answer is a very important one. The problem is: it's possible that during the vector change an interrupt could occur and the half-changed interrupt vector might be the one that is called! The result would be a crash, so instead of taking a chance by doing it ourselves, we can rest assured that the C version performs the vector update in an "atomic" fashion that can't be interrupted. (Actually, it inhibits all interrupts and performs a 32-bit write, but let's just pretend it's magick.)

We're ready to move on to the next step, which is removing the driver (which doesn't exist yet) from the system when our game is complete.

Removing the Driver

To remove the new keyboard driver and replace or restore the old DOS one, we only need to do a single _dos_setvect( ) with the old keyboard handler that was saved in the variable Old_Keyboard_ISR. Listing 5-5 contains a function to do this.

Listing 5-5 Function to remove the keyboard driver

```
void Keyboard_Remove_Driver(void)
{

// this function restores the old keyboard driver (DOS version) with the
// previously saved vector

_dos_setvect(KEYBOARD_INTERRUPT, Old_Keyboard_ISR);

} // end Keyboard_Remove_Driver
```

The only function we are missing at this point is the most important one, which, of course, is the new keyboard driver. Let's discuss it.

The New Keyboard Driver

The new keyboard driver will be called whenever the user presses or releases a key. Basically, when the keyboard hardware retrieves a key, the keyboard tells the PC to initiate an interrupt to service the keyboard event. This interrupt service routine can do whatever it wants to, but at the very least, it should read the scan code out of the PC keyboard buffer and reset the PC's keyboard buffer logic, so another key can be retrieved. This reset involves a couple steps. The first step is to reset the keyboard buffer flip-flop, which is used as a single memory bit to track that a key is in the buffer. Second, the *peripheral interrupt controller* or PIC must be reset so that another interrupt can be processed by the PC. The first step is achieved by writing to one of the keyboard controller ports, and similarly, the second

step is done by writing a 20h to the PIC's control register, which is at port 20h (yes a 20h to 20h—coincidence?).

Resetting the PIC is easy enough and can be done in a single line, but reading the key from the keyboard and resetting the keyboard logic is a bit complex, so let's cover it before we continue. The keyboard buffer is at I/O port 60h and has been defined in our software as *KEYBOARD_BUFFER*. The keyboard control register exists at port 61h and is defined in our software as *KEYBOARD_CONTROL*. When the keyboard detects a keypress, the scan code will be stored in the *KEYBOARD_BUFFER* port and can be retrieved with a simple read. However, after the scan code is retrieved, a specific sequence of operations must be performed to the keyboard control registers to reset the keyboard and indicate to the logic that the key has been processed. This is done by the following steps:

1. The KEYBOARD_CONTROL register is read and logically ORed with 82h.
2. The result is written back out to the KEYBOARD_CONTROL port.
3. The result is then again logically ANDed with 7Fh, and the result is again written out to the KEYBOARD_CONTROL port.
4. Finally, a value of 20h is written to the PIC control register at 20h, and this

completes the keyboard reset and interrupt re-enable.

Of course, before all this happens, the scan code will be read into some variable for processing. In our case, this variable is named *raw_scan_code* and is defined below:

```
int raw_scan_code=0;  // the global keyboard scan code
```

This value is used to update the global keyboard state table, which tracks all keys. The name of the table is *keyboard_state[ ]* and is defined like this:

```
int keyboard_state[128];
```

Now here's the tricky part. The function must place either a 1 or a 0 into each location of the keyboard state table that is indexed by the *raw_scan_code*. The value in raw_scan_code can either be a make code or a break code. We know that all make codes are from 0 to 127, so we can use this information to conclude that if a scan code is retrieved from 0 to 127, it must be a make code, and hence, a 1 will be placed into the *keyboard_state[ ]* table at the location indexed by the scan code. On the other hand, if the scan code is greater than 127, it must be a break code; thus, 128 is subtracted from the *raw_scan_code* value and this is used as an index into the *keyboard_state[ ]* table, and a 0 is placed into the position to reflect that the key has been released.

That completes the steps to write a minimal keyboard handler that can detect and track multiple keypresses in real-time. The function that does all this is shown in Listing 5-6.

Listing 5-6 The new keyboard driver

```
void _interrupt _far Keyboard_Driver()
{
// this function is used as the new keyboard driver. once it is installed
// it will continually update a keyboard state table that has an entry for
// every key on the keyboard, when a key is down the appropriate entry will
be
// set to 1, when released the entry will be reset. any key can be queried by
// accessing the keyboard_state[] table with the make code of the key as the
// index

// need to use assembly for speed since this is an interrupt

_asm
   {
   sti ; re-enable interrupts
   in al, KEY_BUFFER ; get the key that was pressed from the keyboard
   xor ah,ah ; zero out upper 8 bits of AX
   mov raw_scan_code, ax ; store the key in global variable
   in al, KEY_CONTROL ; set the control register to reflect key was read
   or al, 82h ; set the proper bits to reset the keyboard flip flop
   out KEY_CONTROL,al ; send the new data back to the control register
   and al,7fh ; mask off high bit
   out KEY_CONTROL,al ; complete the reset
   mov al,20h ; reset command
   out PIC_PORT,al ; tell PIC to re-enable interrupts

   } // end inline assembly

// update the keyboard table

// test if the scan code was a make code or a break code
if (raw_scan_code <128)
   {
   // index into table and set this key to the "on" state if it already isn't
   // on

   if (keyboard_state[raw_scan_code]==KEY_UP)
      {
      // there is one more active key
      keys_active++;

      // update the state table
      keyboard_state[raw_scan_code] = KEY_DOWN;
```

```
      } // end if key wasn't already pressed

   } // end if a make code
else
   {
   // must be a break code, therefore turn the key "off"
   // note that 128 must be subtracted from the raw key to access the correct
   // element of the state table

   if (keyboard_state[raw_scan_code-128]==KEY_DOWN)
      {
      // there is one less active key
      keys_active--;

      // update the state table
      keyboard_state[raw_scan_code-128] = KEY_UP;

      } // end if key wasn't already pressed

   } // end else

} // end Keyboard_Driver
```

The *Keyboard_Driver( )* works more or less as we have described except for a little twist at the end. When a scan code has been determined to be either a make code or a break code, the keyboard_state[ ] table is referred to before any change is made to see if the change is needed at all. Furthermore, if a key is pressed or released, this is tracked in the global variable *keys_active*. Hence, by looking at *keys_active*, a program can determine whether any keys are down, and if not, the program can completely jump over the keyboard logic.

Now that we have all the functions to implement the new keyboard driver, let's rewrite our original program that tested to see if the [R] or [L] keys were down. The code would look like this:

```
// first install our new driver

Keyboard_Install_Driver();

while(!done)
      {
      // test if any keys are active

      if (keys_active)
          {
```

```
          // test for the 'R' key
          if (keyboard_state[MAKE_R])
             x++;

          // test for the 'L' key
          if (keyboard_state[MAKE_L])
             x--;
          } // end if keys active
      } // end main event loop

// remove the driver when done

Keyboard_Remove_Driver();
```

The beauty of the above fragment is that it can handle multiple keypresses at once. In this case, the player can press both [R] and [L] simultaneously. If this occurs, the result will be that x doesn't change, since it will be both incremented and decremented! This is exactly what we're trying to achieve, and we have done it with a few dozen lines of code and three relatively simple functions. In general, the new keyboard system is practically transparent to use. We load it, use it, and then remove it when we are done.

There is one drawback, and that's the fact that none of the BIOS or C functions will work anymore since we have taken over the keyboard completely and thrown out the old driver. But that's life. Although it's possible to chain interrupts and link the DOS keyboard driver on top of ours, we won't be chaining the keyboard at this time. We'll learn how to do that later in the book.

Keying in the Right Combination

As a demo of the keyboard driver, I have put together a program that loads a PCX file with the image of a numeric keypad. When you run the program, press the keys 1 through 9 on the keyboard and the virtual keys on the screen will light up. The main point of the program is to demonstrate that multiple, simultaneous keypresses can be detected with the new keyboard driver. The name of the demo is KEYTEST.EXE. If you wish to make your own executable from the source KEYTEST.C, you will have to link it to all the previous library modules, including BLACK5.C. Listing 5-7 shows the source for your review.

Listing 5-7 A program to demonstrate the keyboard driver

```
// KEYTEST.C - A demo of the keyboard driver

// I N C L U D E S
////////////////////////////////////////////////////////////
```

```
#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

// include our graphics library

#include "black3.h"
#include "black4.h"
#include "black5.h"

// D E F I N E S
///////////////////////////////////////////////////////////////

#define START_NUMERIC_COLOR 6*16    // start of color register bank that
                                    // keys are drawn with

// G L O B A L S
//////////////////////////////////////////////////////////////

pcx_picture image_pcx;  // general PCX image used to load background and
imagery

// M A I N
////////////////////////////////////////////////////////////////////

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

RGB_color on  = {0,63,0},   // the color values for the on and off buttons
          off = {0,20,0};

// set the graphics mode to mode 13h

Set_Graphics_Mode(GRAPHICS_MODE13);

// load the screen image

PCX_Init((pcx_picture_ptr)&image_pcx);

// load a PCX file (make sure it's there)
```

```
if (PCX_Load("keypad.pcx", (pcx_picture_ptr)&image_pcx,1))
   {
   // copy the image to the display buffer

   PCX_Show_Buffer((pcx_picture_ptr)&image_pcx);

   // delete the PCX buffer

   PCX_Delete((pcx_picture_ptr)&image_pcx);

   // install the keyboard driver

   Keyboard_Install_Driver();

   // enter main evet loop

   while(!keyboard_state[MAKE_ESC])
        {

        // to avoid video snow wait for vertical retrace

        Wait_For_Vertical_Retrace();

        // now test all the keys to see if they are pressed or released
        // based on this turn the virtual light that illuminates each
        // button on or off

        if (keyboard_state[MAKE_1])
           Write_Color_Reg(START_NUMERIC_COLOR+0, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+0, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_2])
           Write_Color_Reg(START_NUMERIC_COLOR+1, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+1, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_3])
           Write_Color_Reg(START_NUMERIC_COLOR+2, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+2, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_4])
           Write_Color_Reg(START_NUMERIC_COLOR+3, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+3, (RGB_color_ptr)&off);
```

```
        if (keyboard_state[MAKE_5])
           Write_Color_Reg(START_NUMERIC_COLOR+4, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+4, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_6])
           Write_Color_Reg(START_NUMERIC_COLOR+5, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+5, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_7])
           Write_Color_Reg(START_NUMERIC_COLOR+6, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+6, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_8])
           Write_Color_Reg(START_NUMERIC_COLOR+7, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+7, (RGB_color_ptr)&off);

        if (keyboard_state[MAKE_9])
           Write_Color_Reg(START_NUMERIC_COLOR+8, (RGB_color_ptr)&on);
        else
           Write_Color_Reg(START_NUMERIC_COLOR+8, (RGB_color_ptr)&off);

        } // end main event loop

   // remove the keyboard driver

   Keyboard_Remove_Driver();

   // use a screen transition to exit

   Screen_Transition(SCREEN_WHITENESS);

   } // end if pcx file found

// reset graphics to text mode

Set_Graphics_Mode(TEXT_MODE);

} // end main
```

The program starts by loading the PCX image of the keypad and displaying it on the screen. Then the keyboard driver is installed and the main event loop is entered. The event loop contains a set of if statements that test whether each of the keys 1 through 9 are being pressed or not. If a key is being pressed, the virtual key on the screen is illuminated. This illumination is accomplished by using color

register animation. Each one of the keys on the keypad was drawn using a different color register; then, by changing the values of the color registers, the keys look like they light up. Also, there is an interesting line of code that calls the *Wait_For_Vertical_Retrace( )* function. This is necessary to avoid screen snow due to the continual updating of the color registers.

On some VGA cards, when the color registers are changed, the video image "snows." This problem can be minimized by making infrequent changes to the color registers or by updating them during the vertical retrace, as we have done here with the *Wait_For_Vertical_Retrace( )* function. Finally, we exit the program by pressing [ESC], at which time the old keyboard driver is replaced and the program exits.

Joysticks

To evoke the strongest feeling of a true video game input device, you can't beat the joystick. The joystick has always been a common input device on home video game consoles and arcade machines alike. However, it's only been in the last few years that the PC has adopted the joystick as a main video game input device. Although the PC always had hardware to support joysticks, many people didn't like them because of their crude design and poor response. Today, things are different–joysticks are futuristic, with lights and rapid fire buttons that make them look like they're from a movie set!

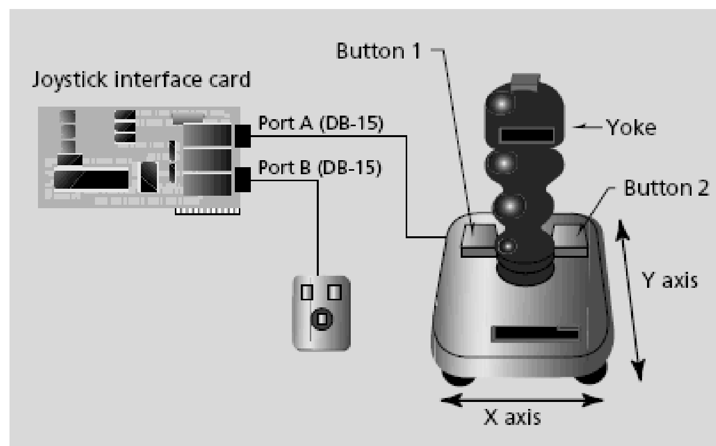

Figure 5 - A joystick and interface card.

Although there have been many changes in its design and construction, the basic operation of the PC joystick has remained the same. PC joysticks are controlled and interfaced by a joystick card. These days, most I/O cards or sound cards have a joystick port or two built in. Otherwise, you can still buy a separate joystick card and plug it in your PC. This isn't practical, however, since it's a bit of a waste using a whole slot for a joystick card. In any case, the interface card plugs into the PC and the joystick(s) plugs into the card.

The joysticks on the PC are of course analog, in contrast to the

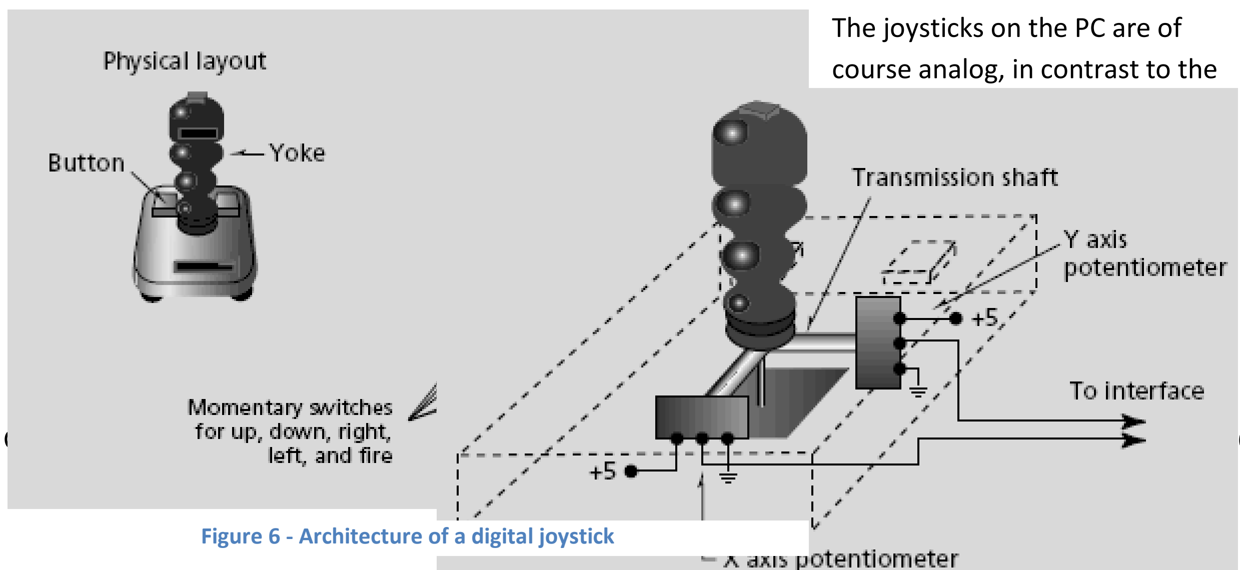

Figure 6 - Architecture of a digital joystick

simpler digital models on video game consoles and arcade machines. This means that the joysticks can detect a range of values in both the X and Y directions. Figure 5-5 shows a typical joystick setup on the PC. Notice also that the joystick has two buttons. These buttons are placed in different positions depending on the joystick, but in general, the PC joystick design only supports two buttons. So how does the PC's analog joystick work? Well, first let's take a look at how a digital joystick works for some insight.

Basically, digital joysticks are a set of switches placed under some interface. When a digital joystick yoke is moved up, down, right, or left, one of the switches is activated. Figure 5-6 shows a digital joystick and its electrical configuration. The problem with a digital joystick is that it can only detect extreme values. It can't be used to go a little to the left or a lot to the right. It's either all or nothing! The analog joystick, on the other hand, uses a device called a potentiometer (a mechanically variable resistor) to measure the amount the joystick is deflected in either direction. Take a look at Figure 5-7. We see that each axis of the joystick is connected to a potentiometer, and these potentiometers are then used as the resistive elements in a timing circuit on the joystick card. Also, there are a total of two joysticks, therefore, four timing circuits.

You might recall from high school physics, if you have a voltage source and you apply it to a resistor in series with a capacitor, the capacitor will take a certain amount of time to charge. This time is proportional to both the capacitor and the resistor. In the case of the PC's joystick design, the capacitors are a fixed value, as are the voltages. The only variables are the resistors. Hence, if we count the amount of time it takes for the charging circuits to charge the capacitors, we can compute the proportional value of the resistors, which in essence are the positions of the joystick potentiometers. The joystick cards don't have real A/D converters. And that's why it's necessary to use this timing method to compute the value of the joystick potentiometers (which are proportional to the X and Y axes deflection of the joystick yoke). In reality, the joystick potentiometers are part of the timing circuits of standard 555 timers–in case you're interested. But if you're not an electrical engineer, don't worry about it!

Figure 7 - The electronics of an analog joystick

Reading the Buttons

Now that we know the basic premise behind the PC's joysticks, let's work our way through writing software to read the buttons and position of the sticks. Reading the buttons is the easy part, so we'll start there. A single port called the joystick port communicates with the joystick interface card. It is located at I/O port 201h and has been defined in BLACK5.H as:

```
#define JOYPORT          0x201  // joyport is at 201 hex
```

The joystick port is used to read both the buttons and the position of each joystick. Amazingly enough, all this information is somehow encoded in this

Bit Number	Meaning
0	Joystick 1 X axis strobe
1	Joystick 1 Y axis strobe
2	Joystick 2 X axis strobe
3	Joystick 2 Y axis strobe
4	Joystick 1 Button 1
5	Joystick 1 Button 2
6	Joystick 2 Button 1

Table 2 – Bit assignments for the joystick port at 201h

single port. Table 5-2 shows the bit assignments for the port.

Let's ignore bits 0 through 3 for a moment, since they are used to read the positions of the joysticks, and focus on bits 4 through 7, which are used to read the buttons of the joysticks. They work in a straightforward manner. If a button is pressed on joystick 1 or 2, the appropriate bit will be set. The only glitch is that the bits are active low, meaning if no buttons are down, then bits 4 through 7 will all be 1's. Then when a button is pressed, the appropriate bit will revert to 0. This isn't a problem since we can invert the bits after we have read them so that a pressed button is a 1 and a released button is a 0.

Now let's write a function that can read either joystick's buttons. The function will take a single parameter, which is the bit mask to extract the proper bit from the joystick port. Also, the function will invert the result as we have discussed. Here are the defines for all the different buttons to make the function easier to call:

```
#define JOYSTICK_BUTTON_1_1    0x10 // joystick 1, button 1
#define JOYSTICK_BUTTON_1_2    0x20 // joystick 1, button 2
#define JOYSTICK_BUTTON_2_1    0x40 // joystick 2, button 1
#define JOYSTICK_BUTTON_2_2    0x80 // joystick 2, button 2
```

Before we write the function, there is one detail that I must tell you: the joystick port must be strobed before we can access it. This strobe tells the joystick port to sample the data and latch it into the joystick port buffer at 201h. To accomplish this strobe, all we have to do is write any data to the port itself. In essence, by writing a value to the port, the hardware is notified that it should take a sample of the joystick buttons. This is a bit crude, but it's not a big deal. However, if you didn't know this little fact, you could read the joystick port until you're blue in the face with no results! Listing 5-8 contains the final function that does it all.

Listing 5-8 Function to read the buttons of the joysticks

```
unsigned char Joystick_Buttons(unsigned char button)
{

// this function reads the state of the joystick buttons by retrieving the
// appropriate bit in the joystick port

outp(JOYPORT,0); // clear the joystick port and request a sample

// invert buttons then mask with request so that a button that is pressed
// returns a "1" instead of a "0"

return( (unsigned char)(~inp(JOYPORT) & button));
```

```
} // end Joystick_Buttons
```

Notice that the first line of the function strobes the port and directs it to sample; then the port is read and masked with the requested button, and the result is inverted and returned to the caller. Hence, if we wished to test if button 1 of joystick 1 was pressed, we could write:

```
if (Joystick_Buttons(JOYSTICK_BUTTON_1_1))
    {
     // fire photon torpedoes!
     } // end if joystick 1's first button is pressed
```

Pretty easy, huh? That's very important in video games. Make all your functions simple and to the point. Let's move on to reading the position of the joystick(s).

Computing the Position of the Stick

As we learned, each joystick consists of two potentiometers, which are each mechanically connected to both the X and Y axes of the joystick yoke. The potentiometers are in turn electrically connected to timing circuits within the joystick card and, based on their positions, change the charging time of the circuits. The value of the potentiometers (which is really the position of the joystick) can then be found by taking note of how long the charging took place. The question is, how do we make this whole process happen? Well, it's actually very easy. This is what we do:

1. Reset a counter.
2. Strobe the joystick port with any value. This starts the charging process and sets the joystick position bits all to 1's.
3. Increment the counter.
4. Test if the desired joystick bit has reverted to 0, and if so, the value in the counter is proportional to the position of the joystick axis; if not, return to step 3.

Basically, we start the charging process for all the joysticks by strobing the joystick port. Then we count how many cycles it takes for the desired bit to revert back to 0 in the joystick port. Of course the bits we are testing are the joystick strobe bits 0 through 3. The only problem with this method of testing the joystick position is that it's highly machine dependent. If we time how many cycles a joystick circuit takes to charge on an XT, it will be very different from how many cycles the same circuit will charge on a 586. Therefore, the values we retrieve must somehow be scaled or interpreted from a calibrated

baseline. This is why many games have a joystick calibration phase. During this phase, the maximum and minimum values of the joystick's X and Y axes are computed, which can then be used to interpret the values retrieved from the joystick port during the game. Later we'll cover this, but keep the timing problem in mind.

We have all the information we need to read the position of either stick's X or Y axis. As usual, I've defined a few constants to make the calls to the function more readable:

```
#define JOYSTICK_1          0x01  // joystick 1
#define JOYSTICK_2          0x02  // joystick 2
#define JOYSTICK_1_X        0x01  // joystick 1, x axis
#define JOYSTICK_1_Y        0x02  // joystick 1, y axis
#define JOYSTICK_2_X        0x04  // joystick 2, x axis
#define JOYSTICK_2_Y        0x08  // joystick 2, y axis
```

There are defines for each axis of each joystick and a couple defines to represent either stick generically. Listing 5-9 contains the code to read a joystick on the PC.

Listing 5-9 Reading a joystick on the PC

```
unsigned int Joystick(unsigned char stick)
{
// this function will read a joystick by starting the timing circuits
connected
// to each joystick port, when the timing circuit has charged the joystick
// bit will revert to 0, the time this process takes is proportional to
// the joystick position and is returned as the result

_asm
   {
   cli                    ; disable interrupts for timing purposes
   mov ah, byte ptr stick  ; select joystick to read with bitmask
   xor al,al              ; zero out al
   xor cx,cx              ; clear cx i.e. set it to 0
   mov dx,JOYPORT         ; point dx to the joystick port
   out dx,al              ; start the 555 timers charging charged:
   in al,dx               ; read the joystick port and test if the bit
   test al,ah             ; has reverted back to 0
   loopne charged         ; if the joystick circuit isn't charged then
                          ; decrement cx and loop

   xor ax,ax              ; zero out ax
   sub ax,cx              ; subtract cx from ax to get a number that
increases
                          ; as the joystick position is moved away from
neutral
```

```
   sti                     ; re-enable interrupts

   } // end asm

// ax has the 16 result, so no need for an explicit return

} // end Joystick
```

The function is obviously in assembly language, but this is due to the tight timing constraints of the problem. I admit that I've never thought of writing one in pure C, but I bet a C version would work just as well on today's machines. Anyway, the function reads the requested joystick axis and returns the number of clicks or cycles that the charging took. The caller then must interpret this value. Remember, the same joystick on two machines may return very different values for the same position. The values may even be an entire order of magnitude off if the speed of the two machines is different enough.

The function literally implements the four steps we outlined at the beginning of this section except for a single change. The counter is decremented instead of incremented. This is done so the function will "time out" if it takes too long. This will occur if there isn't a joystick present. In any case, the final value returned to the caller is computed by subtracting the value in CX from 0, which will result in a number that becomes larger as the joystick is moved away from its neutral position and smaller as the joystick is moved toward its neutral position.

As an example of reading the joystick, we could write something like this:

```
x_pos = Joystick(JOYSTICK_1_X);
y_pos = Joystick(JOYSTICK_1_Y);
```

This fragment would read the X and Y axes of joystick 1 and place the values into the variables x_pos and y_pos. You might be wondering if BIOS has any joystick support? The answer is yes, but the functions are very slow. However, the BIOS routines are very accurate and they prescale the values of the joystick timings so that the results will always be from 0 to 512. This makes the calibration process almost unnecessary if the BIOS routines are used to read the joystick. The BIOS joystick interface is supported under universal interrupt *INT 15h*. The joystick function is 84h and has the subfunctions listed in Table 5-3.

Function 84h

Subfunction 00h: Read buttons.

Entry: AH=84h
DX=00h

Exit: AL=Buttons as defined below:

Bit Number	*Meaning*
0	Don't care
1	Don't care
2	Don't care
3	Don't care
4	Joystick 1 button 1
5	Joystick 1 button 2
6	Joystick 2 button 1
7	Joystick 2 button 2

Subfunction 01h: Read joystick positions.

Entry: AH=84h
DX=01h

Exit: AX = Joystick 1 X axis
BX = Joystick 1 Y axis
CX = Joystick 2 X axis
DX = Joystick 2 Y axis

Table 3 - The BIOS joystick interface INT 15h.

Using the information in Table 5-3, we can create BIOS versions of the button function and the joystick function; however, it only makes sense to write a joystick position function since the BIOS button function does the same thing ours does. However, the BIOS version of the joystick position not only reads the desired joystick position but it autoscales it to a range of 0 to 255. This added functionality isn't free (it's very slow), but it's a quick way to write a very robust joystick interface if speed isn't an issue. Anyway, Listing 5-10 contains yet another version of the joystick position function that is based on BIOS. Notice that it uses the same interface as the previous low-level *Joystick( )* function.

Listing 5-10 A joystick position function based on BIOS

```
unsigned int Joystick_Bios(unsigned char stick)
{
// read the joystick using bios interrupt 15h with the joystick function 84h

union _REGS inregs, outregs; // used to hold CPU registers
inregs.h.ah = 0x84; // joystick function 84h
inregs.x.dx = 0x01; // read joysticks subfunction 1h

// call the BIOS joystick interrupt

_int86(0x15,&inregs, &outregs);

// return requested joystick

switch(stick)
      {
      case JOYSTICK_1_X:  // ax has joystick 1's X axis
           {
           return(outregs.x.ax);
           } break;

      case JOYSTICK_1_Y:  // bx has joystick 1's Y axis
           {
           return(outregs.x.bx);
           } break;

      case JOYSTICK_2_X:  // cx has joystick 2's X axis
           {
           return(outregs.x.cx);
           } break;

      case JOYSTICK_2_Y:  // dx has joystick 2's Y axis
           {
           return(outregs.x.dx);
           } break;
      default:break;
```

```
      } // end switch stick

} // end Joystick_Bios
```

The function doesn’t do much more than respond to the request (which is the desired joystick and axis?) and return their proper value. We can literally replace any reference to the standard joystick function *Joystick( )* with *Joystick_Bios( )* in any of our programs if we wish to use the BIOS version. For example, the small fragment we wrote awhile back to compute the x and y positions of joystick 1 can be rewritten as:

```
x_pos = Joystick_Bios(JOYSTICK_1_X);
y_pos = Joystick_Bios(JOYSTICK_1_Y);
```

Remember though that the BIOS version is very slow and shouldn't be used in time-critical input processing. With that in mind, let's talk about calibration.

Calibrating the Joystick

If you’ve ever played a PC game, you’ve undoubtedly calibrated a joystick or two in your lifetime. The calibration process consists of moving the joystick to the extremes of its working envelope and pressing the fire button (or hitting a key) at different times. The purpose of this is to create a table that records the maximum and minimum values that the joystick can output. Take a look at Figure 5-8, which depicts a typical stick and some of the values that might be obtained during a calibration. Notice that the minimum and maximum X and Y values are recorded along with the neutral position.

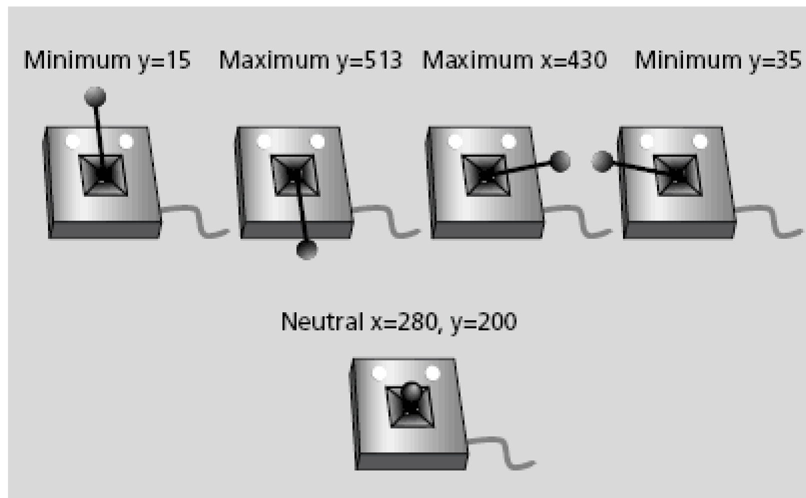

Figure 8 - Typical calibration values for an analog joystick.

The neutral position is needed since the center of the joystick may not be (0,0). It may be non-zero, such as (50,80); hence, the values retrieved by the *Joystick( )* or *Joystick_Bios( )* functions have to consider that (50,80) is the neutral position and not (0,0), as you might think. Anyway, we have everything we need to write a calibrator. All the calibrator needs to do is query the player to move the stick all around

and record the maximum and minimum values in a table along with the neutral position. I have found that asking the user to swirl or move the joystick in a circle is a good method to obtain a full range of motion. We will record the data in the variables defined below:

```
unsigned int joystick_1_max_x,
             joystick_1_max_y,
             joystick_1_min_x,
             joystick_1_min_y,
             joystick_1_neutral_x,
             joystick_1_neutral_y,
             joystick_2_max_x,
             joystick_2_max_y,
             joystick_2_min_x,
             joystick_2_min_y,
             joystick_2_neutral_x,
             joystick_2_neutral_y;
```

The names of the variables should give away each of their meanings, so let's write the function that does the actual calibration. The function will use standard I/O to print to the screen and query the user, but later you may want to make a more advanced joystick calibrator that has a graphical image of the joystick and prints out the values in real-time. Whatever you decide to do, the basic task of the function will be the same as our simple version. Tell the user to swirl the stick around, press fire after a cycle or two, center the stick, and press fire again. As the process takes place, the variables above are updated, and when the function exits, the calibration will be complete.

Also, we're going to add one more feature to our calibration function. We're going to allow it to use either the BIOS version of the joystick function or the lowlevel assembly version for the calibration process. This allows a little more flexibility because the method used to calibrate the joystick must be the same used to read the joystick during the game or else the data will be scaled incorrectly. We'll talk about how the calibration data is used in a moment, but let's take a look at the function in Listing 5-11.

Listing 5-11 A joystick calibration function

```
void Joystick_Calibrate(int stick,int method)
{

// this function is used to calibrate a joystick. the function will
// query the user to move the joystick in circular motion and then release
// the stick back to the neutral position and press the fire button. using
// this information the function will compute the max,min and neutral
// values for both the X and Y axis of the joystick. these values will
// then be stored in global variables so they can be used by other
// functions, note the function can use either the BIOS joystick call
```

```
// or the low level one we made

unsigned int x_value, // used to read values of X and Y axis in real-time
             y_value;

// which stick does caller want to calibrate?

if (stick==JOYSTICK_1)
   {
   printf("\nCalibrating Joystick #1: Move the joystick in a circle, then");
   printf("\nplace the stick into its neutral position and press fire.");

   // set calibrations values to extremes

   joystick_1_max_x = 0;
   joystick_1_max_y = 0;
   joystick_1_min_x = 32000;
   joystick_1_min_y = 32000;

   // process X and Y values in real time

   while(!Joystick_Buttons(JOYSTICK_BUTTON_1_1 | JOYSTICK_BUTTON_1_2))
        {
        // get the new values and try to update calibration

        // test if user wants to use bios or low level
        if (method==USE_BIOS)
           {
           x_value = Joystick_Bios(JOYSTICK_1_X);
           y_value = Joystick_Bios(JOYSTICK_1_Y);
           }
        else
           {
           x_value = Joystick(JOYSTICK_1_X);
           y_value = Joystick(JOYSTICK_1_Y);
           }
        // update globals with new extremes

        // process X - axis
        if (x_value >= joystick_1_max_x)
            joystick_1_max_x = x_value;

        if (x_value <= joystick_1_min_x)
            joystick_1_min_x = x_value;

        // process Y - axis
        if (y_value >= joystick_1_max_y)
            joystick_1_max_y = y_value;
```

```
        if (y_value <= joystick_1_min_y)
            joystick_1_min_y = y_value;

        printf("\nx value = %d, y value = %d",x_value, y_value);

        } // end while

        // stick is now in neutral position so record the values here also
        joystick_1_neutral_x = x_value;
        joystick_1_neutral_y = y_value;

   // notify user process is done
   printf("\nJoystick #1 Calibrated. Press the fire button to exit.");
   while(Joystick_Buttons(JOYSTICK_BUTTON_1_1 | JOYSTICK_BUTTON_1_2));
   while(!Joystick_Buttons(JOYSTICK_BUTTON_1_1 | JOYSTICK_BUTTON_1_2));

   } // end calibrate joystick #1

else
if (stick==JOYSTICK_2)
   {
   printf("\nCalibrating Joystick #2: Move the joystick in a circle, then");
   printf("\nplace the stick into its neutral position and press fire.");

   // set calibrations values to extremes
   joystick_2_max_x = 0;
   joystick_2_max_y = 0;
   joystick_2_min_x = 32000;
   joystick_2_min_y = 32000;

   // process X and Y axis values in real time
   while(!Joystick_Buttons(JOYSTICK_BUTTON_2_1 | JOYSTICK_BUTTON_2_2))
      {
      // get the new values and try to update calibration

      // test if user wants to use bios or low level
      if (method==USE_BIOS)
        {
        x_value = Joystick_Bios(JOYSTICK_1_X);
        y_value = Joystick_Bios(JOYSTICK_1_Y);
        }
      else
        {
        x_value = Joystick(JOYSTICK_1_X);
        y_value = Joystick(JOYSTICK_1_Y);
        }
```

```
        // update globals with new extremes

        // process X - axis
        if (x_value >= joystick_2_max_x)
            joystick_2_max_x = x_value;
        else
        if (x_value <= joystick_2_min_x)
            joystick_2_min_x = x_value;

        // process Y - axis
        if (y_value >= joystick_2_max_y)
            joystick_2_max_y = y_value;
        else
        if (y_value <= joystick_2_min_y)
           joystick_2_min_y = y_value;

        } // end while

        // stick is now in neutral position so record the values here also
        joystick_2_neutral_x = x_value;
        joystick_2_neutral_y = y_value;

   // notify user process is done
   printf("\nJoystick #2 Calibrated. Press the fire button to exit.");

   while(Joystick_Buttons(JOYSTICK_BUTTON_1_1 | JOYSTICK_BUTTON_1_2));
   while(!Joystick_Buttons(JOYSTICK_BUTTON_1_1 | JOYSTICK_BUTTON_1_2));

   } // end calibrate joystick #2

} // end Joystick_Calibrate
```

The joystick calibration function supports both joysticks and both low-level and BIOS reading methods. The function is self-contained but must be executed before the graphics of the game have started since the function uses standard *printf( )* for output. The function uses a very simple algorithm to compute the maximum and minimum joystick values: first, the calibration variables are set to impossibly large and small values; then for each cycle, the new joystick values are tested against the current minimum and maximum values; then if the new values are smaller or larger, the last calibration values are overwritten with the new ones. Hence, by the end of the process, the variables “converge” to the actual minimum and maximum values for the joystick motion. The function itself is called with parameters representing the joystick to calibrate along with the method of calibration. The two methods are defined in BLACK5.H as:

1. define USE_BIOS 0 // command to use BIOS version of something

2. define USE_LOW_LEVEL 1 // command to use our own low level version

As an example, say we wanted to calibrate joystick 1 using the low-level method. We would call the calibration function like this:

```
Joystick_Calibrate(JOYSTICK_1, USE_LOW_LEVEL);
```

Of course, our entire game or application would have to use the joystick function called *Joystick( )* for the calibration information to be valid.

The next burning question should be, how do we use this calibration information? The answer is, however we like! The calibration information simply gives us upper and lower bounds on the values we expect from the joystick function. Using these bounds, we can scale the joystick values to anything we wish, test if the joystick is in the neutral position, and so forth. A typical use of the calibration values is to test if the joystick is in the neutral position. For example, as a player moves his body around and gets excited, the joystick sometimes moves slightly, enough to make a change of values that wasn't really intended by the player. Hence, the software should implement a method that uses the possible range of values along with the neutral position to compute a "dead band" around the neutral position that is all considered neutral. As an example, let's use a joystick model that has only one axis, the X axis, and we have done the calibration and recorded the following:

The maximum X is *max_x*.
The minimum X is *min_x*.
The neutral or center position is *neutral_x*.

We could then write some pseudocode that wouldn't do anything if the joystick hadn't been moved enough to be considered an intentional movement. Here's the code:

```
// compute the possible range
range = max_x - min_x

// compute 10% of this range
delta = .1 * range;

// obtain the joystick position
value = Get_Joystick();

// test if the user really moved the joystick

if (value > neutral_x + delta || value<neutral_x - delta)
   {
   // process motion
   } // end if user really moved the stick
```

```
else
   {
   // user must have bumped the stick a little, do nothing
   } // end else
```

The code fragment basically implements the following logic: if the joystick is moved farther than 10 percent of its range, then the movement was intended; otherwise, it was probably an accident. The value of 10 percent might be too high or too low, but this depends on the situation.

Now let's talk about a little detail that could really throw a monkey wrench in our whole program. What if there isn't a joystick connected and we execute these functions? This is definitely something to consider, so we should figure out a way to detect if a joystick(s) is present.

Joystick Detection Techniques

The joystick interface card has no clean way of detecting if a joystick is present that isn't hardware dependent. In the worst case, not only is the joystick missing, but there isn't even a joystick card! The best way to test for the existence of a joystick is to use the BIOS joystick routines to read the position of the stick. If both positions are 0, there's a 99 percent chance that the system is missing the joystick. You see, a value of (0,0) for the X and Y axes basically means infinite resistance, and that's definitely the case when the only thing connected to the joystick port is air! Therefore, we simply need to write a function that reads the X and Y *axes* of a joystick, adds the values together, and if the result is 0, there is no joystick attached. The function is shown in Listing 5-12.

Listing 5-12 Detecting the presence of a joystick

int Joystick_Available(int stick_num) { // test if the joystick that the user is requesting is plugged in // note the use of the BIOS joystick function, it is very reliable

if (stick_num == JOYSTICK_1)

```
{
  // test if joystick 1 is plugged in by testing the port values
  // they will be 0,0 if there is no stick
  return(Joystick_Bios(JOYSTICK_1_X)+Joystick_Bios(JOYSTICK_1_Y));
  } // end if joystick 1
```

else

```
  {
```

```
// test if joystick 2 is plugged in by testing the port values
// they will be 0,0 if there is no stick
return(Joystick_Bios(JOYSTICK_2_X)+Joystick_Bios(JOYSTICK_2_Y));
} // end else joystick 2
```

} // end Joystick_Available

The function takes a single parameter, which is the joystick to try and detect. As an example, if we wanted to test if joystick 2 was available, we could write:

if (Joystick_Available(JOYSTICK_2))

```
{
// stick is connected
} // end if</pre>
```

As a rule of thumb, it's always a good idea to first test if the joystick(s) is attached to the PC before allowing the user to calibrate it. Since the calibration function can only exit when the fire button is pressed, the lack of a joystick could present a problem, and an infinite loop would be the result!

Driving the Stick Crazy

For the joystick demo things have become a little more exciting. The joystick is used to control an alien ship on a starfield background. Not only do I want you to study the use of the joystick, but this demo is another good example of an event loop. The alien ship is erased, moved, and then drawn over and over. Remember, this is the standard technique we will be using throughout the book to create game loops and real-time event handling. In any case, the name of the demo is JOYTEST.EXE and the source is JOYTEST.C. Use the joystick to rotate the ship and move forward and backward. When the program starts, it will try to detect a joystick and calibrate it, so be sure to have one plugged in. Listing 5-13 contains the source.

Listing 5-13 A joystick demonstration program

```
// JOYTEST.C - A demo of the joystick driver

// I N C L U D E S
///////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "black3.h"
#include "black4.h"
#include "black5.h"

// D E F I N E S
/////////////////////////////////////////////////////////////

#define SHIP_FRAMES 16  // number of animaton frames of ship

// G L O B A L S
////////////////////////////////////////////////////////////

pcx_picture image_pcx;  // general PCX image used to load background and
imagery

sprite ship;            // the players ship

// these are the velocity lookup tables, they have pre-computed velocities
// for each of the 16 directions the ship can be pointing in

int x_velocity[SHIP_FRAMES] = {0,2,4,4,6,4,4,2,0,-2,-4,-4,-6,-4,-4,-2};
int y_velocity[SHIP_FRAMES] = {-6,-4,-4,-2,0,2,4,4,6,4,4,2,0,-2,-4,-4};

// M A I N /////////////////////////////////////////////////////////////////

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

int index,     // loop variable
    dx,        // use to hold roughly 15% of the range of each
    dy,        // joystick axis
    joy_x,     // the final normalized joystick position values
    joy_y;

// test if there is a joystick

if (!Joystick_Available(JOYSTICK_1))
   {
   printf("\nJoystick 1 not connected. Exiting.");
   return;
```

```
   } // end if

printf("\nJoystick 1 detected...");

// calibrate the stick

Joystick_Calibrate(JOYSTICK_1,USE_LOW_LEVEL);

// compute 15% of range

dx = (int)( (float).5 + (float).15 * (float)(joystick_1_max_x-
joystick_1_min_x));
dy = (int)( (float).5 + (float).15 * (float)(joystick_1_max_y-
joystick_1_min_y));

// set the graphics mode to mode 13h

Set_Graphics_Mode(GRAPHICS_MODE13);

// create the double buffer

Create_Double_Buffer(200);

// load the imagery for ship

PCX_Init((pcx_picture_ptr)&image_pcx);

PCX_Load("falcon.pcx", (pcx_picture_ptr)&image_pcx,1);

// intialize the ship sprite

Sprite_Init((sprite_ptr)&ship,160,100,24,20,0,0,0,0,0,0);

// extract the bitmaps for the ship, there are 16 of them, one for each
// pre-rotated angle

// get images off first row of template 0-11

for (index=0; index<12; index++)

PCX_Get_Sprite((pcx_picture_ptr)&image_pcx,(sprite_ptr)&ship,index,index,0);

// get images off second row of template 12-15

for (index=12; index<16; index++)
    PCX_Get_Sprite((pcx_picture_ptr)&image_pcx,(sprite_ptr)&ship,index,index-
12,1);
```

```
// done with this PCX file so delete memory associated with it

PCX_Delete((pcx_picture_ptr)&image_pcx);

// now load the background startfield

PCX_Init((pcx_picture_ptr)&image_pcx);

PCX_Load("frontier.pcx",(pcx_picture_ptr)&image_pcx,1);

// copy PCX image to double buffer

PCX_Copy_To_Buffer((pcx_picture_ptr)&image_pcx,double_buffer);

// delete the pcx image

PCX_Delete((pcx_picture_ptr)&image_pcx);

// scan under the ship, so the first time through the event loop
// whas something to replace

Sprite_Under_Clip((sprite_ptr)&ship,double_buffer);

// main event loop, process until keyboard hit

while(!kbhit())
     {

     // do animation cycle: 1. erase, 2.game logic, 3. scan, 4. draw

     // erase the ship

     Sprite_Erase_Clip((sprite_ptr)&ship,double_buffer);

// PLAYERS SHIP LOGIC

     // get joystick position and subtract away center to
     // compute delta from center

     joy_x = Joystick(JOYSTICK_1_X) - joystick_1_neutral_x;
     joy_y = Joystick(JOYSTICK_1_Y) - joystick_1_neutral_y;

     // test if player has moved stick past the 10% mark, if so transform
     // ship

     if (joy_x > dx)
        {
        // rotate ship right
```

```
      if (++ship.curr_frame==SHIP_FRAMES)
         ship.curr_frame = 0;
      }
   else
   if (joy_x < -dx)
      {
      // rotate ship left

      if (--ship.curr_frame == -1)
         ship.curr_frame = SHIP_FRAMES-1;

      } // end if rotating left

   // test if player is movinh ship foward or backward

   if (joy_y > dy)
      {
      // move ship backward

      // index into velocity table and translate ship with values

      ship.x -= x_velocity[ship.curr_frame];
      ship.y -= y_velocity[ship.curr_frame];

      }
   else
   if (joy_y < -dy)
      {
      // move ship foward

      ship.x += x_velocity[ship.curr_frame];
      ship.y += y_velocity[ship.curr_frame];

      } // end if foward

   // clip ship to screen universe

   if (ship.x > 319)
       ship.x = -24;
   else
   if (ship.x < -24)
       ship.x = 319;

   if (ship.y > 199)
       ship.y = -20;
   else
   if (ship.y < -20)
```

```
        ship.y = 199;

    // ready to draw ship, but first scan background under them

    Sprite_Under_Clip((sprite_ptr)&ship,double_buffer);

    Sprite_Draw_Clip((sprite_ptr)&ship,double_buffer,1);

    // display double buffer

    Display_Double_Buffer(double_buffer,0);

    // lock onto 18 frames per second max

    Time_Delay(1);

    } // end while

// exit in a very cool way

Screen_Transition(SCREEN_DARKNESS);

// free up all resources

Sprite_Delete((sprite_ptr)&ship);
Delete_Double_Buffer();

Set_Graphics_Mode(TEXT_MODE);

} // end main
```

The demo JOYTEST.EXE is one of the most complex we have seen thus far (although the layer demo is probably the coolest!). The program begins by testing for a joystick and exiting if one isn't found. If a joystick is available, it is calibrated by the user and the main portion of initialization begins. First, the double buffer is allocated and the animation cells for the player's ship are loaded into a sprite. Notice that there are 16 frames of animation for the player's ship. This is because the ship can face one of the 16 directions. The 16 animation cells were predrawn using Deluxe Animation. This is a standard technique in 2D bitmapped games; that is, to prerotate the images and then load them into the game. Next, the background starfield frontier is copied into the double buffer.

The program is now ready to enter into the main event loop. At the mouth to the event loop is a call to *Sprite_Under_Clip( )*. This is needed to scan the background under the player's ship so that when the event loop is entered for the first time, the call to *Sprite_Erase_Clip( )* has an image to replace. Moving on, the event loop is entered and the ship is erased. Then the joystick's position is read and the deltas relative to the neutral position are calculated (this uses the precomputed calibration data). Then the joystick position is tested to see if the player is trying to rotate the ship or move. The logic will filter out

any movements that don't exceed at least 15 percent of the joystick travel. This is to minimize the amount of accidental motion that is common with analog joysticks. Actually, your joystick may need a little trimming even after the demo starts. If you notice the ship drifting or rotating, adjust the joystick trimmers until the ship is motionless.

Rotation of the ship is performed by changing the *curr_frame* field of the ship sprite and using it as an index into the sprite animation cells, which are in 22.5 degree rotations. Hence, by incrementing or decrementing the *curr_frame* field of the sprite structure, it makes the ship look like it's rotating. However, if the player presses the joystick forward or backward, the ship will fly in the direction the ship is facing. This is accomplished by using the *curr_frame* field as an index in a pair of tables that contain precomputed velocities. Of course, we could have used the direction of the ship and sin and cos to compute the velocity for any given direction, but using look-up tables allows us to do computations primarily with integers instead of floating point numbers (which are slower).

The next portion of the ship logic tests whether the ship has flown off the screen; if so, the ship is warped back to the opposite edge. Finally, the ship is drawn and the double buffer is copied to the screen. We exit the demo by pressing any key, after which all the resources are released back to DOS and the demo exits. If you're feeling motivated, add some weapons and gravity to the ship!

The Mouse

Last but not least, it's time to mickey with the mouse (I couldn't resist). The mouse is a simple 2D input device that functions as a pointer. The user moves the mouse around on a flat surface, and the motion is tracked and sent to the PC via a serial transmission (although some mice, such as bus mice, have dedicated cards). This serial transmission is decoded by the mouse driver software, and the resulting information is used by the game or application to control something. Figure 5-9 shows a representation of a typical mechanical mouse. As the mouse moves in either the X or Y axis, the optical encoders are interrupted. This process is recorded and used to calculate the position and velocity of the mouse. We aren't interested in exactly how the mouse works and sends its information because we will be using a marvelous invention called a driver!

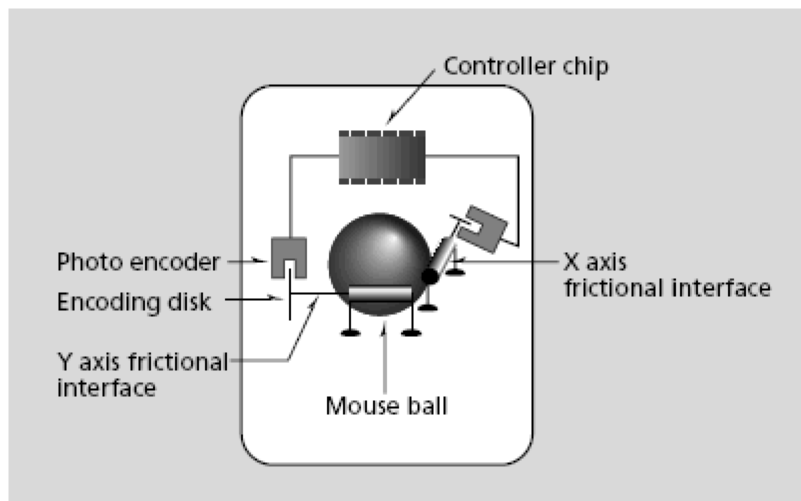

Figure 9 - The insides of a trackball mouse.

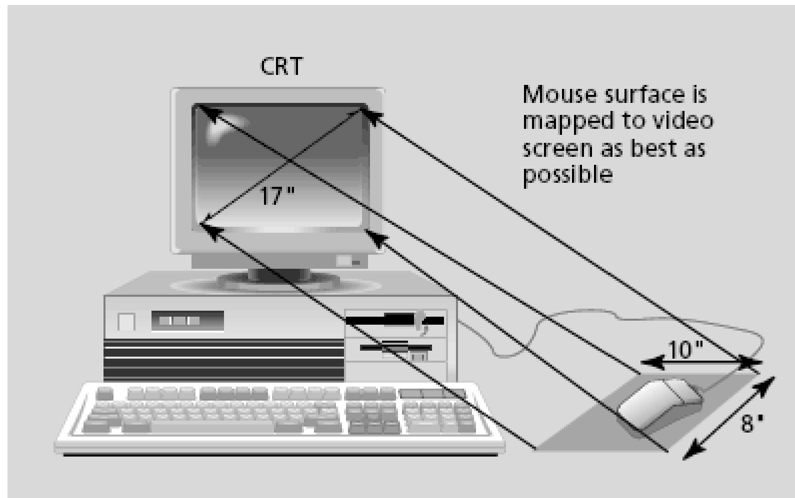

Figure 10 - Mapping the mouse surface to the screen

The mouse driver is a standard interface on PCs that is linked into interrupt INT 33h. We can query the state of the mouse and set its configuration using this interrupt. However, before we can do anything, the mouse driver must be installed during the game's startup process. Typically, the Microsoft mouse driver is called MOUSE.COM and is installed by entering MOUSE.COM at the command line, or as one of the programs loaded during the AUTOEXEC.BAT process. Similarly, there is another version of the driver called MOUSE.SYS, which can be loaded as one of the drivers in CONFIG.SYS. In any case, be sure you have a Microsoft-compatible mouse driver and mouse for the following demos and code.

Communicating with the mouse is very easy and can be done in a single function. Even though the mouse interrupt 33h supports a lot of functions (enough to write a whole book on), we will only concern ourselves with a few. So, what do we want out of a mouse? We want to know its position, we want to be able to show or hide the mouse pointer, determine the state of the buttons, and finally, it would be nice to be able to set the sensitivity. The first items on our wish list are easy enough to understand, but what is sensitivity?

Sensitivity is a measure of how much physical mouse motion it takes to equal one virtual pixel on the screen. Take a look at Figure 5-10. Here we see a mouse, its work area, and the video screen. In most cases, we would like the mouse pointer to be able to move over the entire screen surface by moving the mouse within the bounds of its work area. This means that the 10x8-inch area (in this case) must be able to represent the entire screen without the user lifting up the mouse. This is achieved by setting the mouse sensitivity. The mouse has a finite resolution called a mickey, and each mickey is 1/200th of an inch, or 1/400th, depending on the mouse. Usually, the sensitivity of the mouse is set up to work with the surface area of most mouse pads quite well. However, in the case that you have a really small work area (or really large one), you may want to reprogram the sensitivity of the mouse. That's why we need the sensitivity setting. Personally, I have never in my life changed the sensitivity of a mouse except for the exercise of doing it, but you never know when you'll need it, so we'll support it.

Talking to the Mouse

The mouse functions through INT 33h can be called by setting up the appropriate registers and then making the call to interrupt 33h. The functions we're going to support are listed in Table 5-4.

Function 00h: Mouse reset and status.
Entry: AX=0000h
Exit: AX = FFFFh for success, 0000h is failure.
BX = Number of buttons on mouse.

Function 01h: Show mouse cursor.
Entry: AX=0001h
Exit: Nothing

Function 02h: Hide mouse cursor.
Entry: AX=0002h
Exit: Nothing

Function 03h: Get button status and mouse position.
Entry: AX=0003h
Exit: BX = Button status as defined below:

Bit Number	*Meaning*
7-3	Don't care
2	Center mouse button, 1 if pressed, 0 otherwise.
1	Right mouse button, 1 if pressed, 0 otherwise.
0	Left mouse button, 1 if pressed, 0 otherwise.

CX = X absolute mouse position.
DX = Y absolute mouse position.

Function 0Bh: Read relative motion counters.
Entry: AX=000Bh
Exit: CX = Relative X axis motion in mickey units.
DX = Relative Y axis motion in mickey units.

Function 1Ah: Set sensitivity.
Entry: AX=001Ah
BX = X axis sensitivity from 0–100.
CX = Y axis sensitivity from 0–100.
DX = Mouse double speed value from 0–100.

Table 4 - Mouse interrupt 33h functions

Let's talk about what each function does before we write our mouse interface. The first function, 00h, is simple enough: it is used to reset the mouse driver. The next pair of functions, show mouse and hide mouse, are used to make the mouse pointer visible and invisible; however, they work in a funny way. When the system starts out, the mouse is visible. If we call the hide mouse function, the mouse pointer will become invisible. Then if we call the hide mouse function again, the mouse will still be invisible. But at this point, if we call the show mouse function, we would think the mouse would become visible, right? But it won't! The reason for this is that the mouse has a little internal counter that counts the number of times the mouse has been hidden and shown. When the count is positive, the mouse is visible; when the count is negative, the mouse is invisible. Hence, when writing routines, if you make ten calls to hide mouse, one after another, you had better make ten calls to show mouse if you want the mouse to become visible.

The next function, 03h, is the most important and most frequently used. It is responsible for tracking the mouse position and button state. The position values returned by the function are in screen pixels, not mickeys. However, sometimes (depending on the video mode) the mouse may have a different physical resolution, so you'll have to divide or multiply the mouse position by constants to arrive at a one-to-one mapping of mouse coordinates to screen coordinates.

The next function is useful in flight games and returns the relative motion of the mouse since the last call or, in other words, the delta of the mouse position. Unfortunately, these values are in mickeys, so there isn't an easy relationship between the motion and the screen. But relative motion detection is useful in many cases.

The last mouse function is the sensitivity and double speed control function. Three parameters are sent to the function, one for each setting. By changing these parameters from their default values, you can

change the size of the physical-virtual mouse surface. Now let's take all this information and write a mouse interface function.

A Mouse Interface Function

Unlike the other interface devices such as the keyboard and joystick, the mouse interface consists of a single function that can be called upon to do different operations depending on a command code. These command codes direct the mouse driver to do one of the operations in Table 5-4. To make writing the function easier, here are some defines for the mouse interface to make it more readable:

```
// defines for mouse interface

#define MOUSE_INTERRUPT              0x33 // mouse interrupt number
#define MOUSE_RESET                  0x00 // reset the mouse
#define MOUSE_SHOW                   0x01 // show the mouse
#define MOUSE_HIDE                   0x02 // hide the mouse
#define MOUSE_POSITION_BUTTONS       0x03 // get buttons and position
#define MOUSE_MOTION_REL             0x0B // query motion counters
                                          // to compute relative motion
#define MOUSE_SET_SENSITIVITY        0x1A // set the sensitivity of mouse

// mouse button bitmasks

#define MOUSE_LEFT_BUTTON            0x01 // left mouse button mask
#define MOUSE_MIDDLE_BUTTON          0x04 // middle mouse button mask
#define MOUSE_RIGHT_BUTTON           0x02 // right mouse button mask
```

The function we'll write in a moment should take only a few parameters, such as the command, parameters representing the X and Y axes, and the buttons. These latter variables will also be used as outputs when the function is requested to compute the position of the mouse or its status. Hence, some of the variables should be passed by reference so they can be changed by the function. Listing 5-14 contains the final mouse function that takes all these points into consideration.

Listing 5-14 A mouse interface function

```
int Mouse_Control(int command, int *x, int *y,int *buttons)
{
union _REGS inregs,  // CPU register unions to be used by interrupts
            outregs;

// what is caller asking function to do?
switch(command)
      {
```

```
     case MOUSE_RESET: // this resets the mouse
          {
          // mouse subfunction 0: reset
          inregs.x.ax = 0x00;

          // call the mouse interrupt
          _int86(MOUSE_INTERRUPT, &inregs, &outregs);

          // return number of buttons on this mouse
          *buttons = outregs.x.bx;

          // return success/failure of function
          return(outregs.x.ax);
          } break;

     case MOUSE_SHOW: // this shows the mouse
          {
          // this function increments the internal mouse visibility counter.
          // when it is equal to 0 then the mouse will be displayed.

          // mouse subfunction 1: increment show flag
          inregs.x.ax = 0x01;

          // call the mouse interrupt
          _int86(MOUSE_INTERRUPT, &inregs, &outregs);

          // return success always
          return(1);

          } break;

     case MOUSE_HIDE: // this hides the mouse
          {
          // this function decrements the internal mouse visibility counter.
          // when it is equal to -1 then the mouse will be hidden.

          // mouse subfunction 2: decrement show flag
          inregs.x.ax = 0x02;

          // call the interrupt
          _int86(MOUSE_INTERRUPT, &inregs, &outregs);

          // return success
          return(1);

          } break;

     case MOUSE_POSITION_BUTTONS: // this gets both the position and
```

```
                                // state of buttons
          {
          // this function computes the absolute position of the mouse
          // and the state of the mouse buttons

          // mouse subfunction 3: get position and buttons
          inregs.x.ax = 0x03;

          // call the mouse interrupt
          _int86(MOUSE_INTERRUPT, &inregs, &outregs);

          // extract the info and send back to caller via pointers
          *x       = outregs.x.cx;
          *y       = outregs.x.dx;
          *buttons = outregs.x.bx;

          // return success always
          return(1);
          } break;

     case MOUSE_MOTION_REL: // this gets the relative motion of mouse
          {
          // this function gets the relative mouse motions from the last
          // call, these values will range from -32768 to +32767 and
          // be in mickeys which are 1/200 of inch or 1/400 of inch
          // depending on the resolution of your mouse

          // subfunction 11: get relative motion
          inregs.x.ax = 0x0B;

          // call the interrupt
          _int86(MOUSE_INTERRUPT, &inregs, &outregs);

          // extract the info and send back to caller via pointers
          *x       = outregs.x.cx;
          *y       = outregs.x.dx;

          // return success
          return(1);

          } break;

     case MOUSE_SET_SENSITIVITY:
          {
          // subfunction 26: set sensitivity
          inregs.x.ax = 0x1A;

          // place the desired sensitivity and double speed values in place
```

```
            inregs.x.bx = *x;
            inregs.x.cx = *y;
            inregs.x.dx = *buttons;

            // call the interrupt
            _int86(MOUSE_INTERRUPT, &inregs, &outregs);

            // always return success
            return(1);

            } break;
       default:break;

       } // end switch

} // end Mouse_Control
```

The function simply cases on the input command and calls the mouse interrupt with the proper parameters. If the function is retrieving data such as mouse position, the input variables x, y, and buttons are overwritten. For example, to reset the mouse and print out its x and y position, we could write the following little program:

```
int x,y,buttons;

// reset the mouse

Mouse_Control(MOUSE_RESET, &x,&y,&buttons);

// main event loop

while(true)
        {
        // obtain the state of the mouse
        Mouse_Control(MOUSE_POSITION_BUTTONS,&x,&y,&buttons);

        // print results
        printf("\nMouse is at [%d,%d]",x,y);

        } // end infinite while
```

Point and Squash!

The final mouse demo is probably the closest thing to an actual game we have seen. A grassy area with a picnic blanket is being attacked by ants. Your mission is to use the mouse (which has a hammer attached to it) to squash the ants before they carry away all the food! If you smash an ant, you will see some

graphic feedback that the ant has been exterminated. The name of the program is MOUSETST.EXE and the source code is MOUSETST.C. As usual, to create an executable, you will have to link all the library modules thus far to the final EXE file. The source code is shown in Listing 5-15.

Listing 5-15 A demo that shows a good use of the mouse!

```
// MOUSETST.C - A demo of the mouse driver with some added fun

// I N C L U D E S
////////////////////////////////////////////////////////////

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>
#include <memory.h>
#include <malloc.h>
#include <math.h>
#include <string.h>

#include "black3.h"
#include "black4.h"
#include "black5.h"

// D E F I N E S
//////////////////////////////////////////////////////////////

#define ANT_DEAD       0  // the ant is dead (smashed)
#define ANT_EAST       1  // the ant is moving east
#define ANT_WEST       2  // the ant is moving west

#define HAMMER_UP      0  // the hammer is in it's resting position
#define HAMMER_MOVING  1  // the hammer is hammering!!!

// G L O B A L S
/////////////////////////////////////////////////////////////

pcx_picture image_pcx;  // general PCX image used to load background and
imagery

sprite ant,             // the ant
       hammer;          // the players hammer

// M A I N ///////////////////////////////////////////////////////////////
```

```
void main(int argc, char **argv)
{
int index,      // loop variable
    mouse_x,    // mouse status
    mouse_y,
    buttons;

// set the graphics mode to mode 13h

Set_Graphics_Mode(GRAPHICS_MODE13);

// create the double buffer

Create_Double_Buffer(200);

// load the imagery for ant

PCX_Init((pcx_picture_ptr)&image_pcx);

PCX_Load("moreants.pcx", (pcx_picture_ptr)&image_pcx,1);

// intialize the ant sprite

Sprite_Init((sprite_ptr)&ant,0,0,12,6,0,0,0,0,0,0);

// extract the bitmaps for the ant, there are 3 animation cells for each
// direction thus 6 cells

for (index=0; index<6; index++)

PCX_Get_Sprite((pcx_picture_ptr)&image_pcx,(sprite_ptr)&ant,index,index,0);

// done with this PCX file so delete memory associated with it

PCX_Delete((pcx_picture_ptr)&image_pcx);

// load the imagery for players hammer

PCX_Init((pcx_picture_ptr)&image_pcx);

PCX_Load("hammer.pcx", (pcx_picture_ptr)&image_pcx,1);

// intialize the hammer sprite that will take the place of the mouse
// pointer

Sprite_Init((sprite_ptr)&hammer,0,0,22,20,0,0,0,0,0,0);

// extract the bitmaps for the hammer there are 5
```

```
for (index=0; index<5; index++)

PCX_Get_Sprite((pcx_picture_ptr)&image_pcx,(sprite_ptr)&hammer,index,index,0)
;

// done with this PCX file so delete memory associated with it

PCX_Delete((pcx_picture_ptr)&image_pcx);

// now load the picnik area background

PCX_Init((pcx_picture_ptr)&image_pcx);

PCX_Load("grass.pcx",(pcx_picture_ptr)&image_pcx,1);

// copy PCX image to double buffer

PCX_Copy_To_Buffer((pcx_picture_ptr)&image_pcx,double_buffer);

// delete the pcx image

PCX_Delete((pcx_picture_ptr)&image_pcx);

// scan under ant and hammer before entering the event loop, this must be
// done or else on the first cycle the "erase" function will draw garbage

Sprite_Under_Clip((sprite_ptr)&ant,double_buffer);
Sprite_Under_Clip((sprite_ptr)&hammer,double_buffer);

// reset the mouse and hide the pointer

Mouse_Control(MOUSE_RESET,NULL,NULL,&buttons);
Mouse_Control(MOUSE_HIDE,NULL,NULL,NULL);

// main event loop, process until keyboard hit

while(!kbhit())
     {

     // do animation cycle: 1. erase, 2.game logic, 3. scan, 4. draw

     // erase all objects by replacing what was under them

     if (ant.state!=ANT_DEAD)
         Sprite_Erase_Clip((sprite_ptr)&ant,double_buffer);

     Sprite_Erase_Clip((sprite_ptr)&hammer,double_buffer);
```

```
// PLAYERS HAMMER LOGIC

      // obtain the new position of mouse and state of buttons

      Mouse_Control(MOUSE_POSITION_BUTTONS,&mouse_x,&mouse_y,&buttons);

      // map the mouse position to the screen and assign it to hammer

      hammer.x = (mouse_x >> 1)-16;
      hammer.y = mouse_y;

      // test if player is trying to use hammer

      if (buttons==MOUSE_LEFT_BUTTON && hammer.state==HAMMER_UP)
         {
         // set state of hammer to moving

         hammer.state = HAMMER_MOVING;

         } // end if player trying to strike

      // test if hammer is animating

      if (hammer.state==HAMMER_MOVING)
         {
         // test if sequence complete

         if (++hammer.curr_frame==4)
            {
            hammer.state      = HAMMER_UP;
            hammer.curr_frame = 0;
            } // end if done

         // test if hammer is hitting ant

         if (hammer.curr_frame==3 && ant.state!=ANT_DEAD)
            {
            // do a collision test between the hammer and the ant

            if (ant.x > hammer.x && ant.x+12 < hammer.x+22 &&
                ant.y > hammer.y && ant.y+6 < hammer.y+20)
               {
               // kill ant

               ant.state = ANT_DEAD;

               // draw a smashed ant, use frame 5 of hammer
```

```
              // set current frame to blood splat

              hammer.curr_frame = 4;

              // draw the splat

              Sprite_Draw_Clip((sprite_ptr)&hammer,double_buffer,1);

              // restore the hammer frame

              hammer.curr_frame = 3;

              } // end if hammer hit ant

           } // end if smash test

        } // end if hammer moving

// ANT LOGIC

    // test if it's time to start an ant

    if (ant.state == ANT_DEAD && ((rand()%10)==0))
       {

       // which direction will ant move in

       if ((rand()%2)==0)
          {
          // move ant east

          ant.y          = rand()%200;    // starting y position
          ant.x          = 0;             // starting x position
          ant.counter_1 = 2 + rand()%10; // ant speed
          ant.state     = ANT_EAST;      // ant direction
          ant.curr_frame= 0;             // starting animation frame

          }
       else
          {
          // move ant west

          ant.y          = rand()%200;     // starting y position
          ant.x          = 320;            // starting x position
          ant.counter_1 = -2 - rand()%10; // ant speed
          ant.state     = ANT_WEST;       // ant direction
          ant.curr_frame= 0;              // starting animation frame
```

```
          } // end else west

       } // end if an ant is started

    // test if ant is alive

    if (ant.state!=ANT_DEAD)
       {
       // process ant

       // move the ant

       ant.x+=ant.counter_1;

       // is ant off screen?

       if (ant.x <0 || ant.x > 320)
          ant.state = ANT_DEAD;

       // animate the antm use proper animation cells based on direction
       // cells 0-2 are for eastward motion, cells 3-5 are for westward
motion

       if (ant.state==ANT_EAST)
          if (++ant.curr_frame>2)
             ant.curr_frame=0;

       if (ant.state==ANT_WEST)
          if (++ant.curr_frame>5)
             ant.curr_frame=3;

       } // end ant is alive

     // ready to draw objects, but first scan background under them

     if (ant.state!=ANT_DEAD)
         Sprite_Under_Clip((sprite_ptr)&ant,double_buffer);

     Sprite_Under_Clip((sprite_ptr)&hammer,double_buffer);

     if (ant.state!=ANT_DEAD)
         Sprite_Draw_Clip((sprite_ptr)&ant,double_buffer,1);

     Sprite_Draw_Clip((sprite_ptr)&hammer,double_buffer,1);

     // display double buffer
```

```
        Display_Double_Buffer(double_buffer,0);

        // lock onto 18 frames per second max

        Time_Delay(1);

        } // end while

// exit in a very cool way

Screen_Transition(SCREEN_SWIPE_X);

// free up all resources

Sprite_Delete((sprite_ptr)&ant);
Sprite_Delete((sprite_ptr)&hammer);
Delete_Double_Buffer();

Set_Graphics_Mode(TEXT_MODE);

} // end main
```

The logic of the mouse demo is somewhat similar to that of the joystick demo except that a couple of elements have been added. There is an ant that moves by itself, and there is more animation. The program begins by allocating the double buffer and loading the imagery for the ant and player's hammer. Notice that there are multiple animation frames for both. Furthermore, the ant has animation cells for westward and eastward motion. After the imagery for the ant and hammer are loaded, the background picnic area is loaded and copied to the double buffer.

Once the graphics system is initialized, there are calls to the *Mouse_Control( )* function before the main event loop. These calls are needed to reset the mouse and hide the mouse pointer (the mouse pointer is hidden so that it can be represented by a hammer instead of a pointer). Once we enter the main event loop, the standard animation cycle is followed, starting with the erasure of the ant and hammer. Notice that a test is made to see if the ant is dead or alive. This is because at times the ant won't be visible on the screen, and hence shouldn't be erased. The ant.state field is used to track this fact.

Anyway, the mouse position and buttons are read, and the mouse position is mapped to the screen. This is necessary because the mouse is in a different resolution than mode 13h, so the two lines,

```
hammer.x = (mouse_x >> 1)-16;
hammer.y = mouse_y;
```

make sure that the mouse pointer stays on the screen. Next, the logic for the hammer begins. The hammer can only do one thing and that's strike! Hence, if the player presses the left mouse button, the hammer will start its animation sequence for a strike. This will only occur if the sequence isn't already

running. At the end of the hammer's logic is a collision test to see if the hammer has struck an ant. If so, the ant is killed–that is, its state field is set to *ANT_DEAD*, and a splat is drawn on the screen.

The next portion of code controls the ant and is fairly complex. The ant is tested to see if it's dead or alive. If the ant is dead, the program tries to start another ant. If an ant is started, the program selects a random position and velocity for the ant (west or east). On the other hand, if the ant is alive, it is moved, animated, and tested for collision with the screen boundaries. If the ant has walked off an edge of the screen, it is killed.

Finally, the ant and hammer are drawn and the double buffer is displayed. If the player presses a key, the program exits and the resources are released as usual. The game needs a score and more ants. See if you can add these features. The score is easy, but adding more ants is a bit harder. Here's a hint: use an array of ants …

Creating an Input System

Now that we know more about the keyboard, joystick, and mouse than we ever wanted to, let's talk about a higher level layer of software that can be created. This layer is called an input event system. In general, most games allow the user to select an input device, and the input device's data is filtered through an input handler. This input handler converts the messages or events into one common form and then passes them to the game logic. Figure 5-11 shows this process in action.

As game programmers, we don't want to write special code for each input device; we would much rather have all the input devices filtered through a single function that outputs a sort of generic structure that's the same for any input device. This level of indirection makes it easy to add input devices and debug the code since the input logic is not strewn all over, but is located in a single place.

Typically, the input event systems for each game are slightly different, so I don't have any code for you. The whole point is that the game logic should obtain its input from a virtual input source that could possibly be connected to a joystick, mouse, keyboard, or plug in the back of your head.

Input Device Conditioning

There is one final pseudophilosophical note that we should pay attention to: many input devices don't do what the player is trying to make them do, or in some cases, just aren't designed to do what we as Cybersorcerers are trying to make them do. Therefore, a bit of input conditioning and logic can sometimes make a crude input device into a good one. As an example, say we have written a game of shoot-'em-up and the

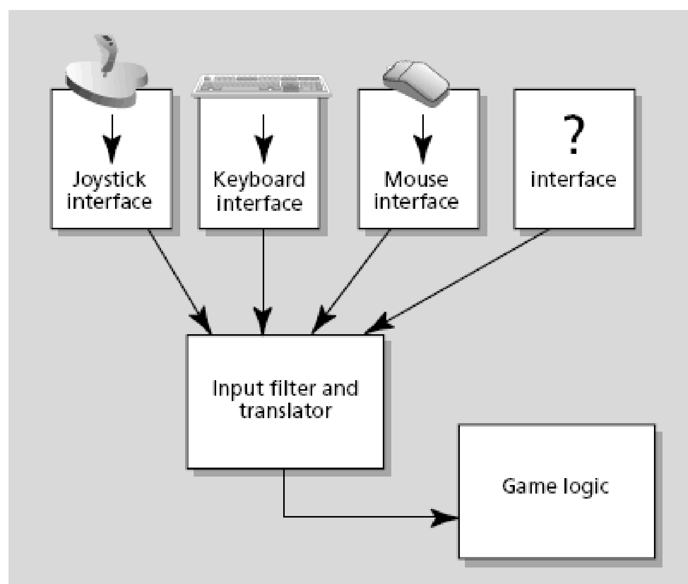

Figure 11- Filtering all inputs to a single function

Copyright 2006 Andre LaMothe

mouse is used as a target control. Imagine that it’s so hard for the player to select a target when a number of them are in a cluster that the player gets frustrated.

A nice feature might be to track the target nearest to the mouse pointer and give the player some help. In this case, he could just place the mouse near the object and the computer would lock onto it. That’s the idea–give the player some help with the input devices to make them do what he is thinking, not what he’s telling them to do!

Summary

In this chapter we covered the most popular input devices supported by the PC and Cybersorcerers alike. We covered the keyboard, joystick, and mouse. And along the way, we created quite an extensive software library to read all of the input devices. The most complex of these is the keyboard driver that can track all the keys in the keyboard at once! Anyway, let me ask you a question: You like to dance, don’t you? Well, it’s time for a bit of music. Read on to see how to make the PC rock!

Copyright 2006 Andre LaMothe