8
Human Interface Devices (HID)

Games are interactive computer simulations, so the human player(s) need some way of providing inputs to the game. All sorts of *human interface devices* (HID) exist for gaming, including joysticks, joypads, keyboards and mice, track balls, the Wii remote, and specialized input devices like steering wheels, fishing rods, dance pads, and even electric guitars. In this chapter, we'll investigate how game engines typically read, process, and utilize the inputs from human interface devices. We'll also have a look at how outputs from these devices provide feedback to the human player.

8.1. Types of Human Interface Devices

A wide range of human interface devices are available for gaming purposes. Consoles like the Xbox 360 and PS3 come equipped with joypad controllers, as shown in Figure 8.1. Nintendo's Wii console is well known for its unique and innovative WiiMote controller, shown in Figure 8.2. PC games are generally either controlled via a keyboard and the mouse, or via a joypad. (Microsoft designed the Xbox 360 joypad so that it can be used both on the Xbox 360 and on Windows/DirectX PC platforms.) As shown in Figure 8.3, arcade machines have one or more built-in controllers, such as a joystick and various buttons, or a track ball, a steering wheel, etc. An arcade machine's input device is usually

Figure 8.1. Standard joypads for the Xbox 360 and PLAYSTATION 3 consoles.

Figure 8.2. The innovative WiiMote for the Nintendo Wii.

Figure 8.3. Various custom input devices for the arcade game *Mortal Kombat II* by Midway.

Figure 8.4. Many specialized input devices are available for use with consoles.

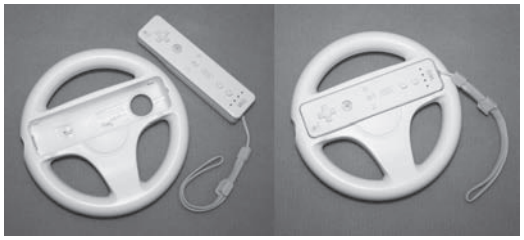

Figure 8.5. Steering wheel adapter for the Nintendo Wii.

somewhat customized to the game in question, although input hardware is often re-used among arcade machines produced by the same manufacturer.

On console platforms, specialized input devices and adapters are usually available, in addition to the "standard" input device such as the joypad. For example, guitar and drum devices are available for the *Guitar Hero* series of games, steering wheels can be purchased for driving games, and games like *Dance Dance Revolution* use a special dance pad device. Some of these devices are shown in Figure 8.4.

The Nintendo WiiMote is one of the most flexible input devices on the market today. As such, it is often adapted to new purposes, rather than replaced with an entirely new device. For example, *Mario Kart Wii* comes with a pastic steering wheel adapter into which the WiiMote can be inserted (see Figure 8.5).

8.2. Interfacing with a HID

All human interface devices provide input to the game software, and some also allow the software to provide feedback to the human player via various kinds of outputs as well. Game software reads and writes HID inputs and outputs in various ways, depending on the specific design of the device in question.

8.2.1. Polling

Some simple devices, like game pads and old-school joysticks, are read by *polling* the hardware periodically (usually once per iteration of the main game loop). This means explicitly querying the state of the device, either by reading hardware registers directly, reading a memory-mapped I/O port, or via a higher-level software interface (which, in turn, reads the appropriate registers or memory-mapped I/O ports). Likewise, outputs might be sent to the HID by

writing to special registers or memory-mapped I/O addresses, or via a higher-level API that does our dirty work for us.

Microsoft's XInput API, for use with Xbox 360 game pads on both the Xbox 360 and Windows PC platforms, is a good example of a simple polling mechanism. Every frame, the game calls the function `XInputGetState()`. This function communicates with the hardware and/or drivers, reads the data in the appropriate way, and packages it all up for convenient use by the software. It returns a pointer to an `XINPUT_STATE` struct, which in turn contains an embedded instance of a struct called `XINPUT_GAMEPAD`. This struct contains the current states of all of the controls (buttons, thumb sticks, and triggers) on the device.

8.2.2. Interrupts

Some HIDs only send data to the game engine when the state of the controller changes in some way. For example, a mouse spends a lot of its time just sitting still on the mouse pad. There's no reason to send a continuous stream of data between the mouse and the computer when the mouse isn't moving—we need only transmit information when it moves, or a button is pressed or released.

This kind of device usually communicates with the host computer via *hardware interrupts*. An interrupt is an electronic signal generated by the hardware, which causes the CPU to temporarily suspend execution of the main program and run a small chunk of code called an *interrupt service routine* (ISR). Interrupts are used for all sorts of things, but in the case of a HID, the ISR code will probably read the state of the device, store it off for later processing, and then relinquish the CPU back to the main program. The game engine can pick up the data the next time it is convenient to do so.

8.2.3. Wireless Devices

The inputs and outputs of a Bluetooth device, like the WiiMote, the DualShock 3 and the Xbox 360 wireless controller, cannot be read and written by simply accessing registers or memory-mapped I/O ports. Instead, the software must "talk" to the device via the Bluetooth protocol. The software can request the HID to send input data (such as the states of its buttons) back to the host, or it can send output data (such as rumble settings or a stream of audio data) to the device. This communication is often handled by a thread separate from the game engine's main loop, or at least encapsulated behind a relatively simple interface that can be called from the main loop. So from the point of view of the game programmer, the state of a Bluetooth device can be made to look pretty much indistinguishable from a traditional polled device.

8.3. Types of Inputs

Although human interface devices for games vary widely in terms of form factor and layout, most of the inputs they provide fall into one of a small number of categories. We'll investigate each category in depth below.

8.3.1. Digital Buttons

Almost every HID has at least a few *digital buttons*. These are buttons that can only be in one of two states: *pressed* and *not pressed*. Game programmers often refer to a pressed button as being *down* and a non-pressed button as being *up*.

Electrical engineers speak of a circuit containing a switch as being *closed* (meaning electricity is flowing through the circuit) or *open* (no electricity is flowing—the circuit has infinite *resistance*). Whether *closed* corresponds to *pressed* or *not pressed* depends on the hardware. If the switch is *normally open*, then when it is not pressed (up), the circuit is *open*, and when it is pressed (down), the circuit is *closed*. If the switch is *normally closed*, the reverse is true—the act of pressing the button opens the circuit.

In software, the state of a digital button (pressed or not pressed) is usually represented by a single bit. It's common for 0 to represent not pressed (up) and 1 to represent pressed (down). But again, depending on the nature of the circuitry, and the decisions made by the programmers who wrote the device driver, the sense of these values might be reversed.

It is quite common for the states of all of the buttons on a device to be packed into a single unsigned integer value. For example, in Microsoft's XInput API, the state of the Xbox 360 joypad is returned in a struct called `XINPUT_GAMEPAD`, shown below.

```
typedef struct _XINPUT_GAMEPAD {
    WORD  wButtons;
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;

    SHORT sThumbLX;

    SHORT sThumbLY;

    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

This struct contains a 16-bit unsigned integer (`WORD`) variable named `wButtons` that holds the state of all buttons. The following masks define

which physical button corresponds to each bit in the word. (Note that bits 10 and 11 are unused.)

```
#define XINPUT_GAMEPAD_DPAD_UP          0x0001 // bit 0
#define XINPUT_GAMEPAD_DPAD_DOWN        0x0002 // bit 1
#define XINPUT_GAMEPAD_DPAD_LEFT        0x0004 // bit 2
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x0008 // bit 3
#define XINPUT_GAMEPAD_START            0x0010 // bit 4
#define XINPUT_GAMEPAD_BACK             0x0020 // bit 5
#define XINPUT_GAMEPAD_LEFT_THUMB       0x0040 // bit 6
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x0080 // bit 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100 // bit 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200 // bit 9
#define XINPUT_GAMEPAD_A                0x1000 // bit 12
#define XINPUT_GAMEPAD_B                0x2000 // bit 13
#define XINPUT_GAMEPAD_X                0x4000 // bit 14
#define XINPUT_GAMEPAD_Y                0x8000 // bit 15
```

An individual button's state can be read by masking the `wButtons` word with the appropriate bit mask via C/C++'s bitwise AND operator (`&`) and then checking if the result is non-zero. For example, to determine if the A button is pressed (down), we would write:

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
    // Mask off all bits but bit 12 (the A button).
    return ((pad.wButtons & XINPUT_GAMEPAD_A) != 0);
}
```

8.3.2. Analog Axes and Buttons

An *analog input* is one that can take on a range of values (rather than just 0 or 1). These kinds of inputs are often used to represent the degree to which a trigger is pressed, or the two-dimensional position of a joystick (which is represented using two analog inputs, one for the x-axis and one for the y-axis,

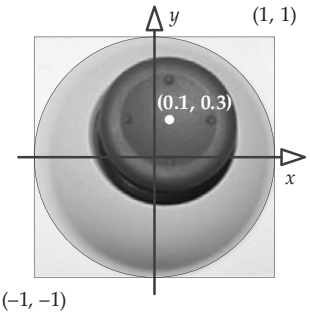

Figure 8.6. Two analog inputs can be used to represent the *x* and *y* deflection of a joystick.

as shown in Figure 8.6). Because of this common usage, analog inputs are sometimes called *analog axes*, or just *axes*.

On some devices, certain buttons are analog as well, meaning that the game can actually detect how hard the player is pressing on them. However, the signals produced by analog buttons are usually too noisy to be particularly usable. I have yet to see a game that uses analog button inputs effectively (although some may very well exist!)

Strictly speaking, analog inputs are not really analog by the time they make it to the game engine. An analog input signal is usually *digitized*, meaning it is quantized and represented using an integer in software. For example, an analog input might range from –32,768 to 32,767 if represented by a 16-bit signed integer. Sometimes analog inputs are converted to floating-point—the values might range from –1 to 1, for instance. But as we know from Section 3.2.1.3, floating-point numbers are really just quantized digital values as well.

Reviewing the definition of `XINPUT_GAMEPAD` (repeated below), we can see that Microsoft chose to represent the deflections of the left and right thumb sticks on the Xbox 360 gamepad using 16-bit signed integers (`sThumbLX` and `sThumbLY` for the left stick and `sThumbRX` and `sThumbRY` for the right). Hence, these values range from –32,768 (left or down) to 32,767 (right or up). However, to represent the positions of the left and right shoulder triggers, Microsoft chose to use 8-bit unsigned integers (`bLeftTrigger` and `bRightTrigger` respectively). These input values range from 0 (not pressed) to 255 (fully pressed). Different game machines use different digital representions for their analog axes.

```
typedef struct _XINPUT_GAMEPAD {
    WORD  wButtons;
```

```
    // 8-bit unsigned
    BYTE  bLeftTrigger;
    BYTE  bRightTrigger;

    // 16-bit signed
    SHORT sThumbLX;
    SHORT sThumbLY;

    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

8.3.3. Relative Axes

The position of an analog button, trigger, joystick, or thumb stick is *absolute*, meaning that there is a clear understanding of where zero lies. However, the inputs of some devices are *relative*. For these devices, there is no clear location at which the input value should be zero. Instead, a zero input indicates that the position of the device has not changed, while non-zero values represent a delta from the last time the input value was read. Examples include mice, mouse wheels, and track balls.

8.3.4. Accelerometers

The PLAYSTATION 3's Sixaxis and DualShock 3 joypads, and the Nintendo WiiMote, all contain acceleration sensors (accelerometers). These devices can detect acceleration along the three principle axes (x, y, and z), as shown in Figure 8.7. These are *relative* analog inputs, much like a mouse's two-dimensional axes. When the controller is not accelerating these inputs are zero, but when the controller is accelerating, they measure the acceleration up to ±3 g along each axis, quantized into three signed 8-bit integers, one for each of x, y, and z.

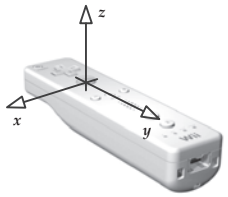

Figure 8.7. Accelerometer axes for the WiiMote.

8.3.5. 3D Orientation with the WiiMote or Sixaxis

Some Wii and PS3 games make use of the three accelerometers in the WiiMote or Sixaxis joypad to estimate the orientation of the controller in the player's

hand. For example, in *Super Mario Galaxy*, Mario hops onto a large ball and rolls it around with his feet. To control Mario in this mode, the WiiMote is held with the IR sensor facing the ceiling. Tilting the WiiMote left, right, forward, or back causes the ball to accelerate in the corresponding direction.

A trio of accelerometers can be used to detect the orientation of the WiiMote or Sixaxis joypad, because of the fact that we are playing these games on the surface of the Earth where there is a constant downward acceleration due to gravity of 1g (≈ 9.8 m/s^2). If the controller is held perfectly level, with the IR sensor pointing toward your TV set, the vertical (z) acceleration should be approximately –1 g.

If the controller is held upright, with the IR sensor pointing toward the ceiling, we would expect to see a 0 g acceleration on the z sensor, and +1 g on the y sensor (because it is now experiencing the full gravitational effect). Holding the WiiMote at a 45-degree angle should produce roughly sin(45°) = cos(45°) = 0.707 g on both the y and z inputs. Once we've calibrated the accelerometer inputs to find the zero points along each axis, we can calculate pitch, yaw, and roll easily, using inverse sine and cosine operations.

Two caveats here: First, if the person holding the WiiMote is not holding it still, the accelerometer inputs will include this acceleration in their values, invalidating our math. Second, the z-axis of the accelerometer has been calibrated to account for gravity, but the other two axes have not. This means that the z-axis has less precision available for detecting orientation. Many Wii games request that the user hold the WiiMote in a non-standard orientation, such as with the buttons facing the player's chest, or with the IR sensor pointing toward the ceiling. This maximizes the precision of the orientation reading, by placing the x- or y-accelerometer axis in line with gravity, instead of the gravity-calibrated z- axis. For more information on this topic, see http://druid.caughq.org/presentations/turbo/Wiimote-Hacking.pdf and http://www.wiili.org/index.php/Motion_analysis.

8.3.6. Cameras

The WiiMote has a unique feature not found on any other standard console HID—an infrared (IR) sensor. This sensor is essentially a low-resolution camera that records a two-dimension infrared image of whatever the WiiMote is pointed at. The Wii comes with a "sensor bar" that sits on top of your television set and contains two infrared light emitting diodes (LEDs). In the image recorded by the IR camera, these LEDs appear as two bright dots on an otherwise dark background. Image processing software in the WiiMote analyzes the image and isolates the location and size of the two dots. (Actually, it can detect and transmit the locations and sizes of up to four dots.) This position

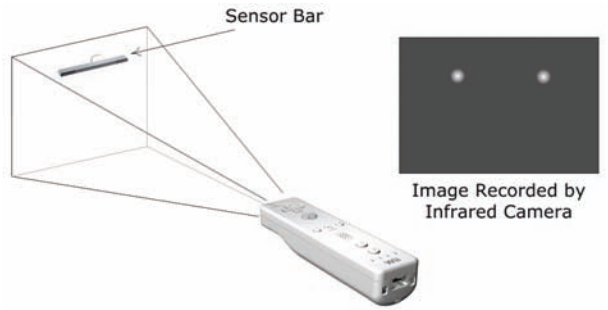

Figure 8.8. The Wii sensor bar houses two infrared LEDs which produce two bright spots on the image recorded by the WiiMote's IR camera.

and size information can be read by the console via a Bluetooth wireless connection.

The position and orientation of the line segment formed by the two dots can be used to determine the pitch, yaw, and roll of the WiiMote (as long as it is being pointed toward the sensor bar). By looking at the separation between the dots, software can also determine how close or far away the WiiMote is from the TV. Some software also makes use of the sizes of the dots. This is illustrated in Figure 8.8.

Figure 8.9. Sony's EyeToy for the PlayStation3.

Another popular camera device is Sony's EyeToy for the PlayStation line of consoles, shown in Figure 8.9. This device is basically a high quality color camera, which can be used for a wide range of applications. It can be used for simple video conferencing, like any web cam. It could also conceivably be used much like the WiiMote's IR camera, for position, orientation, and depth sensing. The gamut of possibilities for these kinds of advanced input devices has only begun to be tapped by the gaming community.

8.4. Types of Outputs

Human interface devices are primarily used to transmit inputs from the player to the game software. However, some HIDs can also provide feedback to the human player via various kinds of outputs.

8.4.1. Rumble

Game pads like the PlayStation's DualShock line of controllers and the Xbox and Xbox 360 controllers have a *rumble* feature. This allows the controller to vibrate in the player's hands, simulating the turbulence or impacts that the

character in the game world might be experiencing. Vibrations are usually produced by one or more motors, each of which rotates a slightly unbalanced weight at various speeds. The game can turn these motors on and off, and control their speeds to produce different tactile effects in the player's hands.

8.4.2. Force-Feedback

Force feedback is a technique in which an actuator on the HID is driven by a motor in order to slightly resist the motion the human operator is trying to impart to it. It is common in arcade driving games, where the steering wheel resists the player's attempt to turn it, simulating difficult driving conditions or tight turns. As with rumble, the game software can typically turn the motor(s) on and off, and can also control the strength and direction of the forces applied to the actuator.

8.4.3. Audio

Audio is usually a stand-alone engine system. However, some HIDs provide outputs that can be utilized by the audio system. For example, the WiiMote contains a small, low-quality speaker. The Xbox 360 controller has a headset jack and can be used just like any USB audio device for both output (speakers) and input (microphone). One common use of USB headsets is for multiplayer games, in which human players can communicate with one another via a voice over IP (VOIP) connection.

8.4.4. Other Inputs and Outputs

Human interface devices may of course support many other kinds of inputs and outputs. On some older consoles like the Sega Dreamcast, the memory card slots were located on the game pad. The Xbox 360 game pad, the Sixaxis and DualShock 3, and the WiiMote all have four LEDs which can be illuminated by game software if desired. And of course specialized devices like musical instruments, dance pads, etc. have their own particular kinds of inputs and outputs.

Innovation is actively taking place in the field of human interfaces. Some of the most interesting areas today are gestural interfaces and thought-controlled devices. We can certainly expect more innovation from console and HID manufacturers in years to come.

8.5. Game Engine HID Systems

Most game engines don't use "raw" HID inputs directly. The data is usually massaged in various ways to ensure that the inputs coming from the HID

translate into smooth, pleasing, intuitive behaviors in-game. In addition, most engines introduce at least one additional level of indirection between the HID and the game in order to abstract HID inputs in various ways. For example, a button-mapping table might be used to translate raw button inputs into logical game actions, so that human players can re-assign the buttons' functions as they see fit. In this section, we'll outline the typical requirements of a game engine HID system and then explore each one in some depth.

8.5.1. Typical Requirements

A game engine's HID system usually provides some or all of the following features:

- dead zones,
- analog signal filtering,
- event detection (e.g., button up, button down),
- detection of button *sequences* and multibutton combinations (known as *chords*),
- gesture detection,
- management of multiple HIDs for multiple players,
- multiplatform HID support,
- controller input re-mapping,
- context-sensitive inputs,
- the ability to temporarily disable certain inputs.

8.5.2. Dead Zone

A joystick, thumb stick, shoulder trigger, or any other analog axis produces input values that range between a predefined minimum and maximum value, which we'll call I_{min} and I_{max}. When the control is not being touched, we would expect it to produce a steady and clear "undisturbed" value, which we'll call I_0. The undisturbed value is usually numerically equal to zero, and it either lies half-way between I_{min} and I_{max} for a centered, two-way control like a joystick axis, or it coincides with I_{min} for a one-way control like a trigger.

Unfortunately, because HIDs are analog devices by nature, the voltage produced by the device is noisy, and the actual inputs we observe may fluctuate slightly around I_0. The most common solution to this problem is to introduce a small *dead zone* around I_0. The dead zone might be defined as $[I_0 - \delta, I_0 + \delta]$ for a joy stick, or $[I_0, I_0 + \delta]$ for a trigger. Any input values that are within the dead zone are simply clamped to I_0. The dead zone must be wide enough to account

for the noisiest inputs generated by an undisturbed control, but small enough not to interfere with the player's sense of the HID's responsiveness.

8.5.3. Analog Signal Filtering

Signal noise is a problem even when the controls are not within their dead zones. This noise can sometimes cause the in-game behaviors controlled by the HID to appear jerky or unnatural. For this reason, many games *filter* the raw inputs coming from the HID. A noise signal is usually of a high-frequency, relative to the signal produced by the human player. Therefore, one solution is to pass the raw input data through a simple *low-pass filter*, prior to it being used by the game.

A discrete first-order low-pass filter can be implemented by combining the current unfiltered input value with last frame's filtered input. If we denote the sequence of unfiltered inputs by the time-varying function $u(t)$ and the filtered inputs by $f(t)$, where t denotes time, then we can write

$$f(t) = (1-a)f(t-\Delta t) + au(t), \tag{8.1}$$

where the parameter a is determined by the frame duration Δt and a filtering constant RC (which is just the product of the resistance and the capacitance in a traditional analog RC low-pass filter circuit):

$$a = \frac{\Delta t}{RC + \Delta t}. \tag{8.2}$$

This can be implemented trivially in C or C++ as follows, where it is assumed the calling code will keep track of last frame's filtered input for use on the subsequent frame. For more information, see http://en.wikipedia.org/wiki/Low-pass_filter.

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{

    F32 a = dt / (rc + dt);

    return (1 - a) * lastFramesFilteredInput
          + a * unfilteredInput;
}
```

Another way to filter HID input data is to calculate a simple moving average. For example, if we wish to average the input data over a 3/30 second (3 frame) interval, we simply store the raw input values in a 3-element circular

buffer. The filtered input value is then the sum of the values in this array at any moment, divided by 3. There are a few minor details to account for when implementing such a filter. For example, we need to properly handle the first two frames of input, during which the 3-element array has not yet been filled with valid data. However, the implementation is not particularly complicated. The code below shows one way to properly implement an *N*-element moving average.

```
template< typename TYPE, int SIZE >
class MovingAverage
{
 TYPE      m_samples[SIZE];
 TYPE      m_sum;
 U32     m_curSample;
 U32     m_sampleCount;

public:
   MovingAverage() :
       m_sum(static_cast<TYPE>(0)),
       m_curSample(0),
       m_sampleCount(0)
 {
 }

 void    addSample(TYPE data)
 {
       if (m_sampleCount == SIZE)
       {
          m_sum -= m_samples[m_curSample];
       }
       else
       {
        ++m_sampleCount;
       }

       m_samples[m_curSample] = data;
       m_sum += data;
       ++m_curSample;
       if (m_curSample >= SIZE)
       {

          m_curSample = 0;
       }
    }

 F32    getCurrentAverage() const
 {
```

```
        if (m_sampleCount != 0)
        {
         return    static_cast<F32>(m_sum)
               / static_cast<F32>(m_sampleCount);
        }
        return 0.0f;
    }
};
```

8.5.4. Detecting Input Events

The low-level HID interface typically provides the game with the current states of the device's various inputs. However, games are often interested in detecting *events*, such as changes in state, rather than just inspecting the current state each frame. The most common HID events are probably button down (pressed) and button up (released), but of course we can detect other kinds of events as well.

8.5.4.1. Button Up and Button Down

Let's assume for the moment that our buttons' input bits are 0 when not pressed and 1 when pressed. The easiest way to detect a change in button state is to keep track of the buttons' state bits as observed last frame and compare them to the state bits observed this frame. If they differ, we know an event occurred. The current state of each button tells us whether the event is a button-up or a button-down.

We can use simple bit-wise operators to detect button-down and button-up events. Given a 32-bit word `buttonStates`, containing the current state bits of up to 32 buttons, we want to generate two new 32-bit words: one for button-down events which we'll call `buttonDowns` and one for button-up events which we'll call `buttonUps`. In both cases, the bit corresponding to each button will be 0 if the event has not occurred this frame and 1 if it has. To implement this, we also need last frame's button states, `prevButtonStates`.

The exclusive OR (XOR) operator produces a 0 if its two inputs are identical and a 1 if they differ. So if we apply the XOR operator to the previous and current button state words, we'll get 1's only for buttons whose states have changed between last frame and this frame. To determine whether the event is a button-up or a button-down, we need to look at the current state of each button. Any button whose state has changed that is currently down generates a button-down event, and vice-versa for button-up events. The following code applies these ideas in order to generate our two button event words:

```
class ButtonState
{

   U32 m_buttonStates;     // current frame's button
         //                   states
   U32 m_prevButtonStates; // previous frame's states

 U32   m_buttonDowns;      // 1 = button pressed this
         //                   frame
 U32   m_buttonUps;        // 1 = button released this
         //                   frame

 void   DetectButtonUpDownEvents()
 {
       // Assuming that m_buttonStates and
       // m_prevButtonStates are valid, generate
       // m_buttonDowns and m_buttonUps.

       // First determine which bits have changed via
       // XOR.
       U32 buttonChanges = m_buttonStates
                         ^ m_prevButtonStates;

       // Now use AND to mask off only the bits that are
       // DOWN.
       m_buttonDowns = buttonChanges & m_buttonStates;

       // Use AND-NOT to mask off only the bits that are
       // UP.
       m_buttonUps = buttonChanges & (~m_buttonStates);
 }

   // ...
};
```

8.5.4.2. Chords

A *chord* is a group of buttons that, when pressed at the same time, produce a unique behavior in the game. Here are a few examples:

- *Super Mario Galaxy*'s start-up screen requires you to press the A and B buttons on the WiiMote together in order to start a new game.
- Pressing the 1 and 2 buttons on the WiiMote at the same time put it into Bluetooth discovery mode (no matter what game you're playing).
- The "grapple" move in many fighting games is triggered by a two-button combination.

- For development purposes, holding down both the left and right triggers on the DualShock 3 in *Uncharted: Drake's Fortune* allows the player character to fly anywhere in the game world, with collisions turned off. (Sorry, this doesn't work in the shipping game!) Many games have a cheat like this to make development easier. (It may or may not be triggered by a chord, of course.) It is called *no-clip mode* in the Quake engine, because the character's collision volume is not *clipped* to the valid playable area of the world. Other engines use different terminology.

Detecting chords is quite simple in principle: We merely watch the states of two or more buttons and only perform the requested operation when *all* of them are down.

There are some subtleties to account for, however. For one thing, if the chord includes a button or buttons that have other purposes in the game, we must take care not to perform *both* the actions of the individual buttons and the action of chord when it is pressed. This is usually done by including a check that the other buttons in the chord are *not* down when detecting the individual button-presses.

Another fly in the ointment is that humans aren't perfect, and they often press one or more of the buttons in the chord slightly earlier than the rest. So our chord-detection code must be robust to the possibility that we'll observe one or more individual buttons on frame i and the rest of the chord on frame $i + 1$ (or even multiple frames later). There are a number of ways to handle this:

- You can design your button inputs such that a chord always does the actions of the individual buttons *plus* some additional action. For example, if pressing L1 fires the primary weapon and L2 lobs a grenade, perhaps the L1 + L2 chord could fire the primary weapon, lob a grenade, *and* send out an energy wave that doubles the damage done by these weapons. That way, whether or not the individual buttons are detected before the chord or not, the behavior will be identical from the point of view of the player.
- You can introduce a delay between when an individual button-down event is seen and when it "counts" as a valid game event. During the delay period (say 2 or 3 frames), if a chord is detected, then it takes precedence over the individual button-down events. This gives the human player some leeway in performing the chord.
- You can detect the chord when the buttons are pressed, but wait to trigger the effect until the buttons are released again.
- You can begin the single-button move immediately and allow it to be preempted by the chord move.

8.5.4.3. Sequences and Gesture Detection

The idea of introducing a delay between when a button actually goes down and when it really "counts" as down is a special case of *gesture detection*. A gesture is a sequence of actions performed via a HID by the human player over a period of time. For example, in a fighting game or brawler, we might want to detect a *sequence* of button presses, such as A-B-A. We can extend this idea to non-button inputs as well. For example, A-B-A-Left-Right-Left, where the latter three actions are side-to-side motions of one of the thumb sticks on the game pad. Usually a sequence or gesture is only considered to be valid if it is performed within some maximum time-frame. So a rapid A-B-A within a quarter of a second might "count," but a slow A-B-A performed over a second or two might not.

Gesture detection is generally implemented by keeping a brief history of the HID actions performed by the player. When the first component of the gesture is detected, it is stored in the history buffer, along with a time stamp indicating when it occurred. As each subsequent component is detected, the time between it and the previous component is checked. If it is within the allowable time window, it too is added to the history buffer. If the entire sequence is completed within the allotted time (i.e., the history buffer is filled), an event is generated telling the rest of the game engine that the gesture has occurred. However, if any non-valid intervening inputs are detected, or if any component of the gesture occurs outside of its valid time window, the entire history buffer is reset, and the player must start the gesture over again.

Let's look at three concrete examples, so we can really understand how this works.

Rapid Button Tapping

Many games require the user to tap a button rapidly in order to perform an action. The frequency of the button presses may or may not translate into some quantity in the game, such as the speed with which the player character runs or performs some other action. The frequency is usually also used to define the validity of the gesture—if the frequency drops below some minimum value, the gesture is no longer considered valid.

We can detect the frequency of a button press by simply keeping track of the last time we saw a button-down event for the button in question. We'll call this T_{last}. The frequency f is then just the inverse of the time interval between presses ($\Delta T = T_{\text{cur}} - T_{\text{last}}$ and $f = 1/\Delta T$). Every time we detect a new button-down event, we calculate a new frequency f. To implement a minimum valid frequency, we simply check f against the minimum frequency f_{min} (or we can just

check ΔT against the maximum period $\Delta T_{max} = 1/f_{min}$ directly). If this threshold is satisified, we update the value of T_{last}, and the gesture is considered to be on-going. If the threshold is not satisfied, we simply don't update T_{last}. The gesture will be considered invalid until a new pair of rapid-enough button-down events occurs. This is illustrated by the following pseudocode:

```
class ButtonTapDetector
{
 U32      m_buttonMask; // which button to observe (bit
                        // mask)
 F32      m_dtMax;      // max allowed time between
                        // presses
 F32      m_tLast;      // last button-down event, in
                        // seconds

public:
   // Construct an object that detects rapid tapping of
   // the given button (identified by an index).
   ButtonTapDetector(U32 buttonId, F32 dtMax) :
      m_buttonMask(1U << buttonId),
  m_dtMax(dtMax),
      m_tLast(CurrentTime() - dtMax) // start out
                                     // invalid
 {
 }

   // Call this at any time to query whether or not the
   // gesture is currently being performed.
 void   IsGestureValid() const
 {
      F32 t = CurrentTime();
      F32 dt = t - m_tLast;
      return (dt < m_dtMax);
 }

   // Call this once per frame.
 void   Update()
 {

  if      (ButtonsJustWentDown(m_buttonMask))
  {
   m_tLast        = CurrentTime();
  }
 }
};
```

In the above code excerpt, we assume that each button is identified by a unique id. The id is really just an index, ranging from 0 to $N - 1$ (where N is the number of buttons on the HID in question). We convert the button id to a

bit mask by shifting an unsigned 1 bit to the left by an amount equaling the button's index (`1U << buttonId`). The function `ButtonsJustWentDown()` returns a non-zero value if *any one* of the buttons specified by the given bit mask just went down this frame. Here, we're only checking for a single button-down event, but we can and will use this same function later to check for multiple simultaneous button-down events.

Multibutton Sequence

Let's say we want to detect the sequence A-B-A, performed within at most one second. We can detect this button sequence as follows: We maintain a variable that tracks which button in the sequence we're currently looking for. If we define the sequence with an array of button ids (e.g., `aButtons[3] = {A, B, A}`), then our variable is just an index i into this array. It starts out initialized to the first button in the sequence, $i = 0$. We also maintain a start time for the entire sequence, T_{start}, much as we did in the rapid button-pressing example.

The logic goes like this: Whenever we see a button-down event that matches the button we're currently looking for, we check its time stamp against the start time of the entire sequence, T_{start}. If it occurred within the valid time window, we advance the current button to the next button in the sequence; for the first button in the sequence only ($i = 0$), we also update T_{start}. If we see a button-down event that doesn't match the next button in the sequence, or if the time delta has grown too large, we reset the button index i back to the beginning of the sequence and set T_{start} to some invalid value (such as 0). This is illustrated by the code below.

```
class ButtonSequenceDetector
{
   U32*   m_aButtonIds;  // sequence of buttons to watch for
 U32    m_buttonCount;   // number of buttons in sequence
 F32      m_dtMax;       // max time for entire sequence
 U32      m_iButton;     // next button to watch for in seq.
 F32      m_tStart;      // start time of sequence, in
                            seconds

public:
   // Construct an object that detects the given button
   // sequence. When the sequence is successfully
   // detected, the given event is broadcast, so the
   // rest of the game can respond in an appropriate way.
   ButtonSequenceDetector(U32* aButtonIds,
                          U32 buttonCount,
                          F32 dtMax,
                          EventId eventIdToSend) :
  m_aButtonIds(aButtonIds),
  m_buttonCount(buttonCount),
```

```
 m_dtMax(dtMax),
      m_eventId(eventIdToSend), // event to send when
                                 //complete
      m_iButton(0),             // start of sequence
      m_tStart(0)               // initial value
                                 //irrelevant
{
}

  // Call this once per frame.
void   Update()
{
      ASSERT(m_iButton < m_buttonCount);

      // Determine which button we're expecting next, as
      // a bit mask (shift a 1 up to the correct bit
//       index).
U32      buttonMask = (1U << m_aButtonId[m_iButton]);

      // If any button OTHER than the expected button
      // just went down, invalidate the sequence. (Use
      // the bitwise NOT operator to check for all other
//       buttons.)
if       (ButtonsJustWentDown(~buttonMask))
{

         m_iButton = 0; // reset
}

      // Otherwise, if the expected button just went
      // down, check dt and update our state appropriately.
      else if (ButtonsJustWentDown(buttonMask))
{
         if (m_iButton == 0)
 {
            // This is the first button in the
  //           sequence.
  m_tStart           = CurrentTime();
            ++m_iButton; // advance to next button
 }

 else
 {
            F32 dt = CurrentTime() - m_tStart;

  if           (dt < m_dtMax)
  {
   //             Sequence is still valid.
```

```
                    ++m_iButton; // advance to next button

      //              Is the sequence complete?
      if              (m_iButton == m_buttonCount)
      {
                      BroadcastEvent(m_eventId);
       m_iButton                = 0; // reset
      }
     }

     else
     {
      //              Sorry, not fast enough.
      m_iButton              = 0; // reset
     }

    }
   }

  }
};
```

Thumb Stick Rotation

As an example of a more-complex gesture, let's see how we might detect when the player is rotating the left thumb stick in a clockwise circle. We can detect this quite easily by dividing the two-dimensional range of possible stick positions into quadrants, as shown in Figure 8.10. In a clockwise rotation, the stick passes through the upper-left quadrant, then the upper-right, then the lower-right, and finally the lower-left. We can treat each of these cases like a button press and detect a full rotation with a slightly modified version of the sequence detection code shown above. We'll leave this one as an exercise for the reader. Try it!

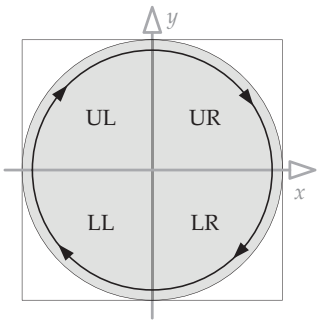

Figure 8.10. Detecting circular rotations of the stick by dividing the 2D range of stick inputs into quadrants.

8.5.5. Managing Multiple HIDs for Multiple Players

Most game machines allow two or more HIDs to be attached for multiplayer games. The engine must keep track of which devices are currently attached and route each one's inputs to the appropriate player in the game. This implies that we need some way of mapping controllers to players. This might be as simple as a one-to-one mapping between controller index and player index, or it might be something more sophisticated, such as assigning controllers to players at the time the user hits the Start button.

Even in a single-player game with only one HID, the engine needs to be robust to various exceptional conditions, such as the controller being accidentally unplugged or running out of batteries. When a controller's connection is lost, most games pause gameplay, display a message, and wait for the controller to be reconnected. Some multiplayer games suspend or temporarily remove the avatar corresponding to a removed controller, but allow the other players to continue playing the game; the removed/suspended avatar might reactivate when the controller is reconnected.

On systems with battery-operated HIDs, the game or the operating system is responsible for detecting low-battery conditions. In response, the player is usually warned in some way, for example via an unobtrusive on-screen message and/or a sound effect.

8.5.6. Cross-Platform HID Systems

Many game engines are cross-platform. One way to handle HID inputs and outputs in such an engine would be to sprinkle conditional compilation directives all over the code, wherever interactions with the HID take place, as shown below. This is clearly not an ideal solution, but it does work.

```
#if TARGET_XBOX360
    if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
#elif TARGET_PS3
    if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
    if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
    // do something...
}
```

A better solution is to provide some kind of hardware abstraction layer, thereby insulating the game code from hardware-specific details.

If we're lucky, we can abstract most of the differences beween the HIDs on the different platforms by a judicious choice of abstract button and axis

ids. For example, if our game is to ship on Xbox 360 and PS3, the layout of the controls (buttons, axes and triggers) on these two joypads are almost identical. The controls have different ids on each platform, but we can come up with generic control ids that cover both types of joypad quite easily. For example:

```
enum AbstractControlIndex
{
   // Start and back buttons
 AINDEX_START,             // Xbox 360 Start, PS3 Start
   AINDEX_BACK_PAUSE,      // Xbox 360 Back, PS3 Pause

   // Left D-pad
 AINDEX_LPAD_DOWN,
 AINDEX_LPAD_UP,
 AINDEX_LPAD_LEFT,
 AINDEX_LPAD_RIGHT,

   // Right "pad" of four buttons
   AINDEX_RPAD_DOWN,       // Xbox 360 A, PS3 X
   AINDEX_RPAD_UP,         // Xbox 360 Y, PS3 Triangle
   AINDEX_RPAD_LEFT,       // Xbox 360 X, PS3 Square
   AINDEX_RPAD_RIGHT,      // Xbox 360 B, PS3 Circle

   // Left and right thumb stick buttons
   AINDEX_LSTICK_BUTTON,   // Xbox 360 LThumb, PS3 L3,
                           // Xbox white
   AINDEX_RSTICK_BUTTON,   // Xbox 360 RThumb, PS3 R3,
                           // Xbox black

   // Left and right shoulder buttons
   AINDEX_LSHOULDER,       // Xbox 360 L shoulder, PS3 L1
   AINDEX_RSHOULDER,       // Xbox 360 R shoulder, PS3 R1

   // Left thumb stick axes
 AINDEX_LSTICK_X,
 AINDEX_LSTICK_Y,

   // Right thumb stick axes
 AINDEX_RSTICK_X,
 AINDEX_RSTICK_Y,

   // Left and right trigger axes
   AINDEX_LTRIGGER,        // Xbox 360 -Z, PS3 L2
   AINDEX_RTRIGGER,        // Xbox 360 +Z, PS3 R2
};
```

Our abstraction layer can translate between the raw control ids on the current target hardware into our abstract control indices. For example, whenever we read the state of the buttons into a 32-bit word, we can perform a bit-swizzling operation that rearranges the bits into the proper order to correspond to our abstract indices. Analog inputs can likewise be shuffled around into the proper order.

In performing the mapping between physical and abstract controls, we'll sometimes need to get a bit clever. For example, on the Xbox, the left and right triggers act as a single axis, producing negative values when the left trigger is pressed, zero when neither is trigger is pressed, and positive values when the right trigger is pressed. To match the behavior of the PlayStation's DualShock controller, we might want to separate this axis into two distinct axes on the Xbox, scaling the values appropriately so the range of valid values is the same on all platforms.

This is certainly not the only way to handle HID I/O in a multiplatform engine. We might want to take a more functional approach, for example, by naming our abstract controls according to their function in the game, rather than their physical locations on the joypad. We might introduce higher-level functions that detect abstract gestures, with custom detection code on each platform, or we might just bite the bullet and write platform-specific versions of all of the game code that requires HID I/O. The possibilities are numerous, but virtually all cross-platform game engines insulate the game from hardware details in *some* manner.

8.5.7. Input Re-Mapping

Many games allow the player some degree of choice with regard to the functionality of the various controls on the physical HID. A common option is the sense of the vertical axis of the right thumb stick for camera control in a console game. Some folks like to push forward on the stick to angle the camera up, while others like an inverted control scheme, where pulling back on the stick angles the camera up (much like an airplane control stick). Other games allow the player to select between two or more predefined button mappings. Some PC games allow the user full control over the functions of individual keys on the keyboard, the mouse buttons, and the mouse wheel, plus a choice between various control schemes for the two mouse axes.

To implement this, we turn to a favorite saying of an old professor of mine, Professor Jay Black of the University of Waterloo, "Every problem in computer science can be solved with a level of indirection." We assign each function in the game a unique id and then provide a simple table which maps each physical or abstract control index to a logical function in the game. When-

ever the game wishes to determine whether a particular logical game function should be activated, it looks up the corresponding abstract or physical control id in the table and then reads the state of that control. To change the mapping, we can either swap out the entire table wholesale, or we can allow the user to edit individual entries in the table.

We're glossing over a few details here. For one thing, different controls produce different kinds of inputs. Analog axes may produce values ranging from –32,768 to 32,767, or from 0 to 255, or some other range. The states of all the digital buttons on a HID are usually packed into a single machine word. Therefore, we must be careful to only permit control mappings that make sense. We cannot use a button as the control for a logical game function that requires an axis, for example. One way around this problem is to normalize all of the inputs. For example, we could re-scale the inputs from all analog axes and buttons into the range [0, 1]. This isn't quite as helpful as you might at first think, because some axes are inherently bidirectional (like a joy stick) while others are unidirectional (like a trigger). But if we group our controls into a few classes, we can normalize the inputs within those classes, and permit remapping only within compatible classes. A reasonable set of classes for a standard console joypad and their normalized input values might be:

- *Digital buttons.* States are packed into a 32-bit word, one bit per button.
- *Unidirectional absolute axes (e.g., triggers, analog buttons).* Produce floating-point input values in the range [0, 1].
- *Bidirectional absolute axes (e.g., joy sticks).* Produce floating-point input values in the range [–1, 1].
- *Relative axes (e.g., mouse axes, wheels, track balls).* Produce floating-point input values in the range [–1, 1], where ±1 represents the maximum relative offset possible within a single game frame (i.e., during a period of 1/30 or 1/60 of a second).

8.5.8. Context-Sensitive Controls

In many games, a single physical control can have different functions, depending on context. A simple example is the ubiquitous "use" button. If pressed while standing in front of a door, the "use" button might cause the character to open the door. If it is pressed while standing near an object, it might cause the player character to pick up the object, and so on. Another common example is a modal control scheme. When the player is walking around, the controls are used to navigate and control the camera. When the player is riding a vehicle, the controls are used to steer the vehicle, and the camera controls might be different as well.

Context-sensitive controls are reasonably straightforward to implement via a state machine. Depending on what state we're in, a particular HID control may have a different purpose. The tricky part is deciding what state to be in. For example, when the context-sensitive "use" button is pressed, the player might be standing at a point equidistant between a weapon and a health pack, facing the center point between them. Which object do we use in this case? Some games implement a priority system to break ties like this. Perhaps the weapon has a higher weight than the health pack, so it would "win" in this example. Implementing context-sensitive controls isn't rocket science, but it invariably requires lots of trial-and-error to get it feeling and behaving just right. Plan on lots of iteration and focus testing!

Another related concept is that of *control ownership*. Certain controls on the HID might be "owned" by different parts of the game. For example, some inputs are for player control, some for camera control, and still others are for use by the game's wrapper and menu system (pausing the game, etc.) Some game engines introduce the concept of a logical device, which is composed of only a subset of the inputs on the physical device. One logical device might be used for player control, while another is used by the camera system, and another by the menu system.

8.5.9. Disabling Inputs

In most games, it is sometimes necessary to disallow the player from controlling his or her character. For example, when the player character is involved in an in-game cinematic, we might want to disable all player controls temporarily; or when the player is walking through a narrow doorway, we might want to temporarily disable free camera rotation.

One rather heavy-handed approach is to use a bit mask to disable individual controls on the input device itself. Whenever the control is read, the disable mask is checked, and if the corresponding bit is set, a neutral or zero value is returned instead of the actual value read from the device. We must be particularly cautious when disabling controls, however. If we forget to reset the disable mask, the game can get itself into a state where the player looses all control forever, and must restart the game. It's important to check our logic carefully, and it's also a good idea to put in some fail-safe mechanisms to ensure that the disable mask is cleared at certain key times, such as whenever the player dies and re-spawns.

Disabling a HID input masks it for all possible clients, which can be overly limiting. A better approach is probably to put the logic for disabling specific player actions or camera behaviors directly into the

player or camera code itself. That way, if the camera decides to ignore the deflection of the right thumb stick, for example, other game engine systems still have the freedom to read the state of that stick for other purposes.

8.6. Human Interface Devices in Practice

Correct and smooth handling of human interface devices is an important part of any good game. Conceptually speaking, HIDs may seem quite straightforward. However, there can be quite a few "gotchas" to deal with, including variations between different physical input devices, proper implementation of low-pass filtering, bug-free handling of control scheme mappings, achieving just the right "feel" in your joypad rumble, limitations imposed by console manufacturers via their technical requirements checklists (TRCs), and the list goes on. A game team should expect to devote a non-trivial amount of time and engineering bandwidth to a careful and complete implementation of the human interface device system. This is extremely important because the HID system forms the underpinnings of your game's most precious resource—its player mechanics.