Lesson 5 - Using a PS3 Joypad

Summary

An introduction to libpad - The SDK library used to interface with input devices on the PlayStation 3.

New Concepts

Libpad, using digital and analogue pad input, relative and absolute movement, function pointers, static variables, static classes, enumerators

Introduction

In this lesson, we're going to take a look at how to receive joypad input, and what to do with the data once we have it. We'll be using Libpad to do this, the joypad interface library provided by Sony in the PS3 SDK. This lesson will also show you a few things you can do with the data you get that'll help you utilise joypad input more effectively.

Relative and absolute positions

If you've ever dealt with mouse input when coding a computer game, you might already be familiar with the concepts of relative and absolute positions. Basically, in general there are two ways of receiving movement updates from a mouse or joypad analogue input - either an *absolute* position in screen coordinates, or *relative* to some frame of reference, such as the previous frame's position, or zero. Absolute positions are useful for things like pointers in your game GUI, whereas relative movement is more useful for character movement in FPS games. The analogue inputs on the PS3 send their data as a single byte, with a value of '0' being the analogue stick fully to the left, and '255' being fully to the right. This is fine for tracking relative movement, but means we have to do a bit of extra work if we want to know where a 'mouse' pointer is on screen.

Function Pointers

If you've read the C++ lesson on threads, you'll already know a bit about function pointers. If not, here's a quick run-down. Functions are just data in memory like anything else, so you can create a pointer to the start of a function, just as you would create a pointer to the start of a variable or Class. You can then use that function pointer to call that function. What good is this? Well, it's a very easy way of encapsulating your classes better. In this lesson, we're going to hide all our input code in a class, imaginatively named *Input*. In this class, we're going to have a function pointer for each button on the joypad, and this pointer will be called if the button is pressed. We are going to have a public function in Input that will let us assign which function each of these function pointers points at. From this simple framework, we can trigger extensive gameplay functions without having to break the encapsulation of the Input class, and reduce the complexity of input handling code. Handy, yes?

Project Overview

For this lesson, we only need a PPU project, so make one called InputPPU. In the project, we're going to define one class called **Input**, and a cpp file called controller, to contain our main function and test out our Input class.

The Input Class

Input Header file

We begin our Input class header in the usual way, with some **#include** definitions. We need access to strings, and the libpad library, so we include them. Our Input class will also make use of the **fabs()** function, so on line 3 we include **stdlib.h**. After this we have some **#defines**, which will be explained as they are used in the class code.

```
1  #pragma once
2
3  #include <stdio.h>
4  #include <math.h>
5  #include <string>
6  #include <iostream>
7  #include <cell/pad.h>
8
9  #define SCREENX          720
10 #define SCREENY          480
11 #define DEADZONE         5.0f
12 #define SENSITIVITY      100.0f
13
14 typedef void(*InputFunction)();
```

input.h

On line 14 we have an unusual looking **typedef**. As described earlier we're going to use function pointers in the Input class, and we want the functions the Input class receives as function pointers to be of a very specific makeup - they return void and have no arguments. This typedef will create this function definition, and call it *InputFunction*. Using this typedef, we can use our function type as a variable data type, just as we would **int**, **float** or **char**.

After our typedef, we are going to define an **enumerator**, called *PadButtons*. If you've not come across enumerators before, don't worry, there's not much to them. All they are is a typedef that encapsulates a named set of constants. In this case, we're creating a type - *PadButtons*, and then creating a set of constants - one to represent each button on the joypad. So just how we have an *int* type that can have values of 1,2,3 etc, we have created a *PadButtons* type that can have values of INPUT_SQUARE, INPUT_CROSS etc. Behind the scenes, the constants are **integers**, beginning with 0 and incrementing with every additional constant in the enum - note how we can explicitly set what each constant is enumberated to using the = operator, just as we'd assign values to a variable.

```
15 enum PadButtons {
16    INPUT_SELECT   = 0,
17    INPUT_L3       = 1,
18    INPUT_R3       = 2,
19    INPUT_START    = 3,
20    INPUT_UP       = 4,INPUT_RIGHT      = 5,
21    INPUT_DOWN     = 6,INPUT_LEFT = 7,
22    INPUT_L2       = 8,
23    INPUT_R2       = 9,
24    INPUT_L1       = 10,
25    INPUT_R1       = 11,
26    INPUT_TRIANGLE = 12, INPUT_CIRCLE   = 13,
27    INPUT_CROSS    = 14, INPUT_SQUARE   = 15,
28    PADBUTTONS_MAX
29 };
```

input.h

Class declaration

Now for the actual Input class. Here you'll see how we use our newly defined *PadButtons* enum, and our *InputFunction* typedef.

```
30 class Input
31 {
32 public:
33     static void Initialise();
34     static void Destroy();
35     static void UpdateJoypad();
36 
37     static void SetPadFunction(PadButtons button, InputFunction fn);
38     static void GetPointerPosition(float &x, float &y);
39     static void GetJoypadMovement(float  &x, float &y);
40 protected:
41     static float pointerPosX;
42     static float pointerPosY;
43 
44     static float pointerRelativeX;
45     static float pointerRelativeY;
46 
47     static InputFunction functions[PADBUTTONS_MAX];
48 };
```

input.h

Notice something a bit different about this class? Everything is static! In fact, as everything is static, it is a *static class*. A class which we only need one instance of, and which doesn't have much in the way of member variables can be made static - making our Input class a perfect choice. Static classes have no constructors or destructors - they're never really instantiated. In many ways, static classes are very similar to just putting some related functions inside a namespace. As we have no constructor or destructor to do things for us automatically behind the scenes, we must manually define some functions to initialise and destroy the Input class, as seen on lines 33 and 34. On line 35 we have *UpdateJoypad*, which will poll the libpad library for the latest joypad state. On line 38 we declare *GetPointerPosition*, a function which will give us screen coordinates, handy for debug GUIs and so on. On line 37, we have *SetPadFunction*, which uses our *PadButtons* enum and our *InputFunction* typdefs. This makes sure that the function passed to *SetPadFunction*, and our target button are both valid. On lines 41 we have static member variables, to hold our relative and absolute joypad movement, and an array of *InputFunctions*. Note the use of the *PADBUTTONS_MAX* enum - as it's the last constant in the enum, it'll always equate to a value large enough to use as an index of an array large enough for every *PadButton*.

Input Class Definition

With our class declared in it's header file, we can flesh it out in its cpp file. On line 1, we include our header file, and then on lines 3-7 we initialise our static member variables. Wait, we initialise variables outside of a function? Remember, static member variables don't belong to any particular instance, so we can't initialise them inside a constructor - and our entire class is static, so we don't have a constructor, anyway! Instead we just initialise them at the top of the cpp file. Note the syntax for initialising these variables, it's almost the same as just declaring a variable, but with an additional class name qualifier so that the compiler knows you are initialising the Input class variables, and not declaring new global variables. It's conceptually similar to how you would declare a class function in a class header, then define it using the class name in another file.

```
#include "Input.h"

float Input::pointerPosX = 0.0f;
float Input::pointerPosY = 0.0f;

float Input::pointerRelativeX = 0.0f;
float Input::pointerRelativeY = 0.0f;

InputFunction Input::functions[PADBUTTONS_MAX];
```

input.cpp

After that, we declare our static *Initialise* and *Destroy* functions. Our *Initialise* function sets all of our function pointers to NULL, and calls the Sony SDK function **cellPadInit**, with an argument of 1. **cellPadInit** turns libpad on, and its argument sets the maximum number of controllers that the application will use. The *Destroy* function simply calls **cellPadEnd** - another Sony SDK function that turns off libpad.

```
void Input::Initialise() {
    if(cellPadInit(1) != CELL_PAD_OK) {
        std::cout << "cellPadInit failed!" << std::endl;
    }

    for(int i = 0; i < NUM_FUNCTIONS; ++i){
        functions[i] = NULL;
    }
}

void Input::Destroy() {
    cellPadEnd();
}
```

input.cpp

We then have some mutator methods - two to get the current state of the joypad analogue controller, and one to set a function pointer for a given joypad button. Note how we can use our PadButtons enum as an index value for an array - remember that enums are basically just named integers. Also, remember how we also explicitly set our enum to begin at 0 - the first index of an array!

```
void Input::GetPointerPosition(float &x, float &y) {
    x = pointerPosX;
    y = pointerPosY;
}

void Input::GetJoypadMovement(float &x, float &y) {
    x = pointerRelativeX;
    y = pointerRelativeY;
}

void Input::SetPadFunction(PadButtons button, InputFunction function){
    functions[button] = function;
}
```

input.cpp

The most important function in our little Input class is *UpdateJoypad*. This function is called whenever we want to get the latest joypad state. On line 38 we get that joypad state from libpad using the SDK function **cellPadGetData**. This function takes two parameters, an int and a reference to a special **CellPadData** struct, which we declare on line 37. The int simply determines which joypad to get the latest data for - we just want the first joypad, so we pass a value of 0, just like how arrays start with an index of 0. The **CellPadData** struct has two variables, an int called *len* and an array of unsigned shorts called *button*. *len* is the number of members of the button array that have changed since the last time **cellPadGetData** was called - so if it is 0, nothing has changed and we don't have to process anything.

```
36 void Input::UpdateJoypad() {
37    CellPadData data;
38    cellPadGetData(0,&data);
```

input.cpp

We use an **if** statement on line 39 to check for changes, and if there are any, start processing them. The joypad button states are kept in the third and fourth shorts of the button array, with each short being a bitmask - remember how we used bitmasks way back in lesson 2, when issuing DMA requests? So, for each of the two button shorts, we check to see if ANY of the bits have been set by using if statements on lines 40 and 49 - remember, a non-zero value equates to **TRUE**. Then, we use a **for** loop to cycle through each of the bits in the short and check to see if they are true. If they are, we check to see if a function pointer for the button exists (using the same non-zero equality as we used for the bitmask), and if it does exist, we call it, just like we would any other function.

```
39    if(data.len > 0) {            //Pad status has changed...
40       if(data.button[2])      {  //One of these buttons has changed...
41          for(int i = 0; i < NUM_FUNCTIONS / 2; ++i) {
42             if(data.button[2] & (1 << i)) {
43                if(functions[i]) {
44                   functions[i]();   //We have a function, call it!
45                }
46             }
47          }
48       }
49       if(data.button[3]) {
50          for(int i = 0; i < NUM_FUNCTIONS / 2; ++i) {
51             if(data.button[3] & (1 << i)) {
52                if(functions[NUM_FUNCTIONS / 2+i]) {
53                   functions[NUM_FUNCTIONS / 2+i]();
54                }
55             }
56          }
57       }
```

input.cpp

But wait, how exactly does this work, and get the correct function pointer for each button? If you look closely at the two loops, you'll see that one will call function pointers 0-7, and the other 8-15. Now, try having a look in the SDK documentation to see how the **CellPadData** bitmasks relate to the joypad buttons. You'll see that we've actually been a bit sneaky with the ordering in the *PadButtons* enum we use to assign function pointers to buttons, and they perfectly match up to the order of the button shorts in the *CellPadData* struct! This clever ordering allows us to collapse the entire button checking into two loops, rather than a series of if statements to check each button individually.

That's the buttons sorted out, but what about the analogue sticks? The data for the left analogue stick is held in the 7th and 8th short of the button array - one short for the X-axis, and one for the Y-Axis. For each axis, a short value of 0 represents full left/up and a short value of 255 represents full right/down. Yes, even though the array is of short type, and thus has a range of 0 to 65535, only half of the bits are used for each axis. It's much easier to think of movement as being relative to 0, so we transform our range of movement, shown on lines 59 and 63. on lines 60 and 64 we use one of our header file **#defines** - *DEADZONE*. What do these **if** statements do? The PS3 analogue joypads are quite sensitive to movement, and will sometimes not quite centre themselves when you let go of them, so we create a 'dead zone', where these slight movements don't count. we use the **fabs** math function to remove the sign bit from our relative movement value, and then compare it against our dead zone value, and zero out very small movements. By checking the absolute value of our movement, we don't have to do any extra logic to accommodate negative or positive movement - for example, both -5 and 5 have an *absolute* value of 5.

```
58        pointerRelativeX = data.button[6] * 2 - 255.0f;
59        if(fabs(pointerRelativeX) < DEADZONE) {
60           pointerRelativeX = 0;
61        }
62        pointerRelativeY = data.button[7] * 2 - 255.0f;
63        if(fabs(pointerRelativeY) < DEADZONE) {
64           pointerRelativeY = 0;
65        }
66     }
```

input.cpp

Nearly done! Our little Input class has the ability to return the current screen coordinates for a graphical pointer, for use in menu screens or debug purposes. So, every frame, we add our relative axis movements to the *pointerPos* variables. Note how we also divide our relative movement by another header macro - **SENSITIVITY**. This allows us to fine tune how fast our screen pointer moves, similar to how we can adust mouse sensitivity in games. As we don't want our screen pointer to go outside of the screen bounds, we check for negative values and zero them, and also check for movement too far the other way, using our **SCREENX** and **SCREENY** macros. These just simulate the size of the screen for the purposes of this lesson - in a proper game you'd get the screen dimensions from the renderer and pass it to your *Input* class for bounds checking.

```
67     pointerPosX  += (pointerRelativeX / SENSITIVITY);
68     pointerPosY  += (pointerRelativeY / SENSITIVITY);
69
70     if(pointerPosX < 0) {
71        pointerPosX = 0;
72     }
73     else if(pointerPosX > SCREENX) {
74        pointerPosX = SCREENX;
75     }
76
77     if(pointerPosY < 0) {
78        pointerPosY = 0;
79     }
80     else if(pointerPosY > SCREENY) {
81        pointerPosY = SCREENY;
82     }
83
84     cellPadClearBuf(0);
85  }
```

input.cpp

One last SDK function and we're done. **cellPadClearBuf** clears the input buffer for a given joypad after you've processed it's data, ready for the next time **cellPadGetData** is called.

Main

OK, so we've now made ourselves a simple little Input class for the PS3 joypads, let's try it out!

Includes and definitions

As ever, we start with our includes and our definitions. Not very much this lesson, we're keeping things simple. We has the usual includes to use strings, and we also include the header file for our Input class. On line 5, we have a boolean value **done** that will control when to break out of a loop in our main function. On lines 7 to 12, we declare some functions that will be sent to the input class as function pointers for calling when a button is pressed down.

```
1  #include <string>
2  #include <iostream>
3  #include "input.h"
4
5  static bool done = false;
6
7  void triangle_button();
8  void square_button();
9  void circle_button();
10 void cross_button();
11 void start_button();
12 void shoulder_button();
```

controller.cpp

Main Function

Now for our main function. It's really nothing more than a simple test harness around our new Input class. We initialise our Input class on line 15, set some function pointers on lines 16 to 26, then enter a loop that continously polls our Input class. Below the loop, we simply clean up and exit.

```
14 int main(void) {
15    Input::Initialise();
16    Input::SetPadFunction(INPUT_SQUARE,    square_button);
17    Input::SetPadFunction(INPUT_CROSS,     cross_button);
18    Input::SetPadFunction(INPUT_CIRCLE,    circle_button);
19    Input::SetPadFunction(INPUT_TRIANGLE,  triangle_button);
20    Input::SetPadFunction(INPUT_L1,  shoulder_button);
21    Input::SetPadFunction(INPUT_L2,  shoulder_button);
22    Input::SetPadFunction(INPUT_R1,  shoulder_button);
23    Input::SetPadFunction(INPUT_R2,  shoulder_button);
24    Input::SetPadFunction(INPUT_START,  start_button);
25    while(!done) {
26       Input::UpdateJoypad();
27    }
28    std::cout << "Start button pressed! Quitting..." << std::endl;
29    Input::Destroy();
30    return 0;
31 }
```

controller.cpp

As for the functions we are going to send to the Input class as function pointers, they're not really very complicated...

```
32 void triangle_button()  {
33    std::cout << "Calling the triangle function pointer!" << std::endl;
34 }
35
36 void square_button() {
37    std::cout << "Calling the square function pointer!" << std::endl;
38 }
39
40 void circle_button() {
41    std::cout << "Calling the circle function pointer!" << std::endl;
42 }
43
44 void cross_button()     {
45    std::cout << "Calling the cross function pointer!" << std::endl;
46 }
47
48 void shoulder_button()     {
49    std::cout << "Pressing a shoulder button!" << std::endl;
50 }
51
52 void start_button()     {
53    done = true;
54 }
```

controller.cpp

Note on line 53 that we set the **bool** that controls our main loop to true - this function will be called if the joypad **Start** button is pressed, quitting the program. If you run this program, you should be able to see how pressing the right side buttons outputs info to the TTY buffer, and pressing Start quits our program.

Tutorial Summary

Nice and easy! The libpad library makes it very easy to access incoming joypad data, and our new Input class hides all the implementation, making it really easy to make our program do something when a button has been pressed. Although this lesson uses the PlayStation 3 libraries for its implementation, you should be able to see how you could use the same basic design pattern to write an input class for windows, linux, or any other development environment you can think of! We do have one problem, though. At the moment our Joypad update function is framerate-dependant - the quicker our framerate, the quicker our pointer will move across the screen. We'll see how to fix that later, in the graphics lessons...

Further Work

1) So far we're only tracking the left analogue stick. Try adding support for the right analogue stick to your joypad.

2) We now know we can detect when a button has been pressed down, but what about when a button is released? If so, can we add function pointers to button releases? What about functions we want to continue being called for as long as a button is being held down?

3) So far we can only keep track of one controller, so try modifying the Input class to support multiple controllers.