Lesson 3
Basic Input with a PS3 controller

Playstation 3 Development

Sam Serrels and Benjamin Kenwright[1]*

Abstract
This article presents an uncomplicated introduction for processing analog and digital input using Sony's Playstation 3 controller. We employ Sony's *libpad* library and present a flexible input class that can be reused easily in other Playstation 3 projects.

Keywords
Sony, PS3, PlayStation, Setup, Windows, Target Manager, ELF, PPU, SPU, Programming, ProDG, Visual Studio

[1] *Edinburgh Napier University, School of Computer Science, United Kingdom*: b.kenwright@napier.ac.uk

Contents

Introduction

About the Edinburgh Napier University Game Technology Playstation 3 Development Lessons Edinburgh Napier University Game Technology Lab is one of the leading game teaching and research groups in the UK - offering students cutting edge facilities that include Sony's commercial development kits. Furthermore, within the Edinburgh Napier Game Technology group are experienced developers to assist those students aspiring to releasing their own games for PlayStation. Students have constant access to he Sony DevKits and encourage enthusiastic students to design and build their own games and applications during their spare time [4].

This tutorial Input is what makes applications interactive, it's what turns programs into games, and for a consoles like the PS3, the controller is the primary input device. This tutorial discusses the features of the PlayStation 3 controller. With the small input classes provided, you will be able to use the controller to trigger functions in your code. The focus is on speed, a small utility class that you can drop into future examples to get input functioning in seconds. Future tutorials will develop more advanced input handling code, while this tutorial shows you the basics for you to work up from.

Additional Reading In addition to the lesson tutorials, we would recommend reading a number of books on Playstation 3 development and cross-platform coding, such as, Cell Programming for the PS3 [3], Vector Maths and Optimisation for the PS3 [1], and Cross-Platform Development in C++ [2].

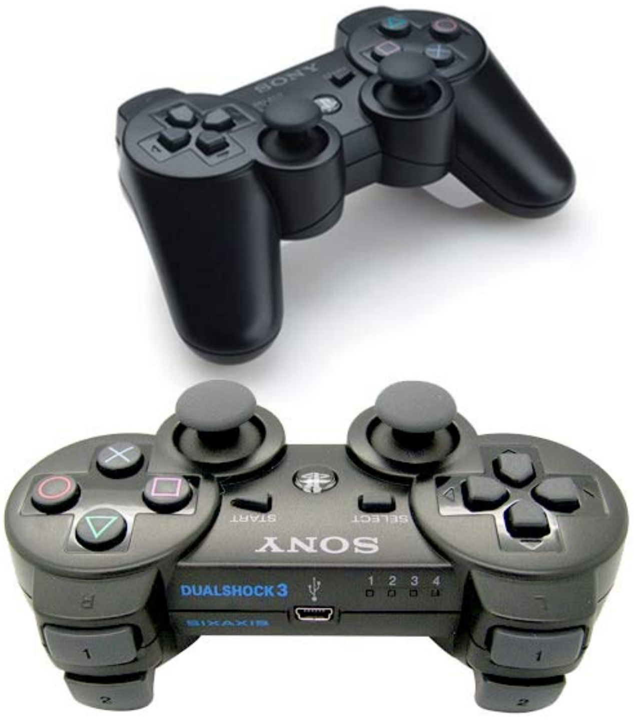

Figure 1. The DualShock 3 wireless controller (SCPH-98050) -

1. The PS3 controller hardware

The Sixaxis Wireless Controller was the official wireless controller for the PlayStation 3 until it was succeeded by the DualShock 3. "Sixaxis" also refers to the motion sensing technology used in both the Sixaxis and DualShock 3 controllers. Both controllers can also be used on the PSP Go via Bluetooth (requires a PlayStation 3 system for initial connection).

The DualShock 3 wireless controller Replacing the Sixaxis as the standard PlayStation 3 controller, the DualShock 3 features the same functions and design (including "Sixaxis" motion sensing), but with vibration feedback capability.

Design Bluetooth is used for the wireless confection (it will also function as a wired controller via USB). There is an internal battery that is charged via a usb cable. The design is an evolution of the DualShock 2 (DS2) controller, retaining its pressure-sensitive buttons, layout and basic shape. The L2 and R2 buttons were replaced with analogue triggers and the precision of the analogue sticks was increased from 8-bit to 10-bit. The center PlayStation button allows access to the system menu.

Inputs Motion sensing (X,Y,Z axis and angular rate sensors), 2 Analogue sticks (10-bit precision), 2 Analogue triggers (L2, R2), 6 Pressure-sensitive buttons(Triangle, Circle, Cross, Square, L1, R1), Pressure-sensitive 4-way D-Pad, 5 Digital buttons ("PS", Start, Select, L3, R3). L3 and R3 are accessed by pushing down on either analogue stick

Analogue buttons Most of the buttons on the controller are not actually digital, they are readable as analogue values (pressure-sensitive). To use this feature the controller has to be put into a certain mode, without doing so results in the controller treating the buttons like digital inputs.

2. The libpad Library

libpad is a library for controlling controllers from an application. Controllers supported by libpad are as follows.
- Game controllers with PlayStation3 specifications: Sixaxis and Dualshock3 wireless controllers, and other similar compatible third party controllers.
- Other game controllers pursuant to USB HID spec 1.1

The main function of libpad is to notify the application of information that is output from the controller, such as, button, analogue stick, and six-axis sensor information. Although the format of data output from a controller is unique to each controller model, libpad automatically converts data into a common button data format when notifying the application. Even for controllers with unique shapes – such as, those in the form of an instrument – the data will be converted to a common button data format per class (for example, guitar type, drum type, etc.) and be notified to the application.
In addition to this, libpad provides a feature to control the vibration of controllers with a rumble feature, a feature to notify the system software of button information output from a custom controller that is individually supported by the application, and an automatic input feature for debug support.

Initialize the library
Call ***cellPadInit()*** to initialize the library.

Get connection information
Call ***cellPadGetInfo2()*** to obtain controller connection information such as how many controllers are connected and to which ports.

Set up controller
Call ***cellPadSetPortSetting()*** as necessary and make settings to enable the pressure-sensitivity mode or the sensor mode for a controller. If necessary, clear the data accumulated in the controller data buffer using ***cellPadClearBuf()***.

Read controller data
Call ***cellPadGetData()*** every 1 to 4 frames and read the controller data. If this read process is repeated before the data gets updated, an empty data (CellPadData.len = 0) meaning "same as before" will return. Update intervals differ by controller. This interval is approximately 10msec with a USB connection, and 11.25msec for a Bluetooth connection. To respond to the disconnection or connection of a controller, periodically call ***cellPadGetInfo2()*** and monitor the connection status.

Terminate the library
To terminate libpad, call ***cellPadEnd()***.

3. Function pointers

In this tutorial, the input code we will be as basic as possible, we need a simple design to manage the multitude of inputs on a controller. Input handling lies in the realms of things like UI code, I.e Event Driven Code, and things can get messy quickly. So for our simple little code, how do we manage this? If you are thinking about arrays, enumerators and bit-masks you are on the right track, the special ingredient we will use is function pointers.

Callbacks 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. This is commonly also referred to as a "callback function" and is used heavily in javascript programming.

Why? In the past you may have done something like: every frame, poll input values from a library, store into an array, and then when needed, looked up that array for certain keys. In production you would want need to do this, but in this simple example we just need the quickest and smallest solution. We let libpad do most of the array work and just call a function when something changes.

4. Code

The code is rather self explanatory, so it is listed here in full.

Listing 1. Input.h

```
#define HALT {__asm volatile( "trap" );}
#define ASSERT(exp) {if(!(exp)){DBG_HALT;}}
#define ASSERT_M(exp,msg){if(!(exp)){puts(msg); HALT}}
#define ASSERT_F(exp,func) {if(!(exp)) {func; HALT}}

# pragma once
# include <cell/pad.h>

//! port_no Controller number(0−6), code currently only supports 1
#define PORT_NO 0

//! The type of function to be used as a pointer for inputs,
typedef void (∗ InputFunction )();

//! Enums of all the buttons on a ps3 controller
enum PadButtons {
  INPUT_SELECT  = 0,
  INPUT_L3    = 1,
  INPUT_R3    = 2,
  INPUT_START   = 3,
  INPUT_UP    = 4,
  INPUT_RIGHT   = 5,
  INPUT_DOWN    = 6,
  INPUT_LEFT    = 7,
  INPUT_L2    = 8,
  INPUT_R2    = 9,
  INPUT_L1    = 10,
  INPUT_R1    = 11,
  INPUT_TRIANGLE  = 12,
  INPUT_CIRCLE  = 13,
  INPUT_CROSS   = 14,
  INPUT_SQUARE  = 15,
  PADBUTTONS_MAX
};

class Input
{
  public :

    //! Call cellPadInit() and initialize arrays.
    static void Initialise();

    //! Release control of controller.
    static void Destroy();

    //! Retrieve and parse controller info, call funcs if needed.
    static void Update();

    //! Register a callback function to a certain button.
    static void SetPadFunction(PadButtons button,InputFunction fn);

    //! Clear buffers and reset pointer position.
    static void Refresh();

  protected :

    //! Array of input callback functions
    static InputFunction functions [ PADBUTTONS_MAX ];
};
```

Listing 2. Input.cpp

```
#include "Input.h"

InputFunction Input :: functions [ PADBUTTONS_MAX ];

// Call cellPadInit() and initialize arrays.
void Input::Initialise ()
{
  int err = cellPadInit (1);
  ASSERT_M( err==CELL_PAD_OK , "cellPadInit failed !" );

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

// Release control of controller.
void Input::Destroy ()
{
  cellPadEnd ();
}

// Clear buffers and reset pointer position.
void Input::Refresh ()
{
  int err = cellPadClearBuf(PORT_NO);
  ASSERT_M(err==CELL_PAD_OK , "cellPadClearBuf failed !");
}

// Register a callback function to a certain button.
void Input::SetPadFunction(PadButtons button, InputFunction fn)
{
  functions [ button ] = fn;
}

// Retrieve and parse controller info, call funcs if needed.
void Input::Update()
{
  CellPadData data ;
  cellPadGetData (PORT_NO  ,& data );
  if(data.len > 0)  // Has Pad status changed ?
  {
    if(data.button[2]) //data.button[2] : Digital button statuses
    {
      for (int i = 0; i < PADBUTTONS_MAX / 2; ++i)
      {
        if( data.button [2] & (1 << i))
        {
          if(functions[i])
          {
            functions[i](); // We have a function , call it!
          }
        }
      }
    }
    if(data.button [3]) // data.button [3]: More Digital button statuses
    {
      for (int i = 0; i < PADBUTTONS_MAX / 2; ++i)
      {
        if( data . button [3] & (1 << i))
        {
          if( functions [PADBUTTONS_MAX / 2+i])
          {
            functions [PADBUTTONS_MAX / 2+i ]();
          }
        }
      }
    }
    //data.button[4]: Right stick (X direction)
    //data.button[5]: Right stick (Y direction)
    //data.button[6]: Left stick (X direction)
    //data.button[7]: Left stick (Y direction)
    //data.button[8−>19]: pressure info, needs cellPadSetPortSetting
    //data.button[20−>23]: Sixaxis info, needs cellPadSetPortSetting
  }
}
```

Listing 3. main.cpp

```
#include <iostream>
#include "Input.h"

static bool done = false ;
void triangle_button();
void square_button();
void circle_button();
void cross_button();
void start_button();
void shoulder_button();

int main(void)
{
 Input :: Initialise ();

 /* We are passing the memory location of the functions to the
   input class, in this case the ampersand(&) is actually optional */
 Input::SetPadFunction( INPUT_SQUARE, &square_button);
 Input::SetPadFunction( INPUT_CROSS, cross_button);
 Input::SetPadFunction( INPUT_CIRCLE, circle_button);
 Input::SetPadFunction( INPUT_TRIANGLE, triangle_button);
 Input::SetPadFunction( INPUT_L1, shoulder_button);
 Input::SetPadFunction( INPUT_L2, shoulder_button);
 Input::SetPadFunction( INPUT_R1, shoulder_button);
 Input::SetPadFunction( INPUT_R2, shoulder_button);
 Input::SetPadFunction( INPUT_START, start_button);

 std::cout << "Starting: Basic input program " << std :: endl ;

 while (! done )
 {
  Input::Update();
 }

 std::cout << " Start button pressed ! Quitting ... " << std :: endl ;
 Input::Destroy ();
 return EXIT_SUCCESS;
}

void triangle_button () {
 std::cout << "Calling the triangle function pointer !" << std::endl;
}

void square_button () {
 std::cout << "Calling the square function pointer !" << std::endl;
}

void circle_button () {
 std::cout << "Calling the circle function pointer !" << std::endl;
}

void cross_button () {
 std::cout << "Calling the cross function pointer !" << std::endl;
}

void shoulder_button () {
 std::cout << "Pressing a shoulder button !" << std::endl;
}

void start_button () {
 done = true;
}
```

4.1 Explanation

Lines 40 to 66, Input.cpp Data.len returns 0 if nothing has changed since the last cellPadGetData().
Data.button[2] and Data.button[3] hold a uint16 of various digital button states, each bit is a seperate button.
So, for each of the two button uint16's, we check to see if ANY of the bits have been set by using if statements on lines 42 and 55 - a non-zero value equates to TRUE. Then, we use a for loop to cycle through each of the bits in the and check to see if they are true. If they are, we check to see if a function pointer for the button exists, if it does, we call it.

5. Conclusion

This example is basic, but it gets the job done and is self contained. In future implementations you will want to implement support for things like multiple controllers, detecting disconnects and reconnects etc...
Making a more fully featured input system is covered in the next input tutorial. Where features like handling of analogue data and other features will be implemented.

Recommended Reading

Programming the Cell Processor: For Games, Graphics, and Computation, Matthew Scarpino, ISBN: 978-0136008866
Vector Games Math Processors (Wordware Game Math Library), James Leiterman, ISBN: 978-1556229213
Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

References

[1] James Leiterman. Vector games math processors (wordware game math library) (isbn:978-1556229213), 2011. 1

[2] Syd Logan. Cross-platform development in c++: Building mac os x, linux, and windows applications (isbn:978-0321246424), 2007. 1

[3] Matthew Scarpino. Programming the cell processor: For games, graphics, and computational proccessing (isbn: 978-0136008866), 2011. 1

[4] Edinburgh Napier Game Technology Website. www.napier.ac.uk/games/. *Accessed: Feb 2014*, 2014. 1