- Contact

Type text to search here...

Home > DirectX > XInput Tutorial Part 1: Adding gamepad support to your Windows game

XInput Tutorial Part 1: Adding gamepad support to your Windows game

August 30, 2013 Leave a comment Go to comments
12 Votes

In this game development tutorial we will look at how to add support for gamepads to Windows games written in C++, using the XInput API introduced in DirectX 11. XInput replaces DirectInput from previous DirectX versions and greatly simplifies the task of working with game controllers, as long as they are XInput-compatible. If you don't normally use a gamepad on your PC but you own an Xbox 360 then you're in luck: Xbox 360 controllers support XInput, so just plug it into a USB port on your PC and you're good to go.

Our end goal is to develop a library which allows us to shoehorn in gamepad support to existing games which use the keyboard and mouse for input with almost no extra effort at all (and if you want to cut to the chase and just use such a library, my Simple2D library versions 1.11 and above include all the code in this article and more to do the job for you), To start with though, we'll have a quick crash course on the basic API.

In this article, you will learn:

- How to compile applications which use XInput
- How to check if a controller is connected and on which port
- How to check for digital button presses
- How to check the movement positions of the analog thumb sticks and rear triggers
- How to calibrate deadzones for the analog sticks
- How to make a class which wraps it all together

NOTE: *If you can't be bothered with the low-level details and just want to shoehorn gamepad support into a game really quickly, my Simple2D library (version 1.11 and above) includes gamepad support – see the Tetris gamepad support article for a quick example on how to use the library to add gamepad support in just a few minutes!*

Setting up your build environment

There are three versions of XInput at the time of writing; XInput 1.3 is bundled with Windows 7, XInput 1.4 is bundled with Windows 8 and XInput 9.1.0 is a generalized version with some features added and some removed, which can be used across platforms. We will be working with XInput 9.1.0 here. To use this, you need to *#include <xinput.h>* at the start of your source code, and include *Xinput9_1_0.lib* as a linker input (if you want to link against XInput 1.3/1.4, use *XInput.lib* as the linker input).

How to check whether a controller is connected

There are four virtual ports numbered 0-3 on which a controller can be connected (on an Xbox 360 controller, the connected port is highlighted by which quadrant of the green ring around the Xbox button is lit, with port 0 being the top-left quadrant, and the other ports proceeding in a clockwise direction around the ring). To see if a controller is connected on

a given port, we try to query the state of the controller on the port, and check for success or failure. If we succeeded in receiving the state, then the controller is connected on that port, otherwise it is disconnected.

The *XInputGetState* function retrieves the state of a controller. The first argument is the port number to query, and the second argument is a pointer to receive the controller state in the form of an *XINPUT_STATE* struct. The function itself returns an error/success code.

```
XINPUT_STATE state;
ZeroMemory(&state, sizeof(XINPUT_STATE));

if (XInputGetState(0, &state) == ERROR_SUCCESS)
{
// controller is connected on port 0
}
```

The code above shows how to check if a controller is connected on port 0. To cycle through all the ports and find the first connected controller, you can use:

```
int controllerId = -1;

for (DWORD i = 0; i < XUSER_MAX_COUNT && controllerId == -1; i++)
{
XINPUT_STATE state;
ZeroMemory(&state, sizeof(XINPUT_STATE));

if (XInputGetState(i, &state) == ERROR_SUCCESS)
controllerId = i;
}
```

When the *for* loop exits, *controllerId* will contain the port number of the first connected controller, or -1 if no controllers are connected.

How to test whether a particular button is pressed down

The *XINPUT_STATE* struct contains an *XINPUT_GAMEPAD* struct, which itself has a *wButtons* value that is a bitmask with one bit representing each button – 1 if the button is pressed, 0 otherwise. XInput defines values you can use to test against *wButtons* as follows:

Face buttons

```
XINPUT_GAMEPAD_A
XINPUT_GAMEPAD_B
XINPUT_GAMEPAD_X
XINPUT_GAMEPAD_Y
```

Directional pad arrows

```
XINPUT_GAMEPAD_DPAD_LEFT
XINPUT_GAMEPAD_DPAD_RIGHT
XINPUT_GAMEPAD_DPAD_UP
XINPUT_GAMEPAD_DPAD_DOWN
```

Shoulder buttons

```
XINPUT_GAMEPAD_LEFT_SHOULDER
XINPUT_GAMEPAD_RIGHT_SHOULDER
```

Analog thumb sticks (when pressed in and used as a button)

```
XINPUT_GAMEPAD_LEFT_THUMB
XINPUT_GAMEPAD_RIGHT_THUMB
```

Centre buttons

```
XINPUT_GAMEPAD_BACK
XINPUT_GAMEPAD_START
```

To test if a particular button is currently held down, we simply perform a logical AND with the *wButtons* value and the values above. For example, to test if the user is pressing A, we can do:

```
bool A_button_pressed = ((state.Gamepad.wButtons & XINPUT_GAMEPAD_A
```

How to test how much each trigger is depressed

The left and right triggers are the bottom two buttons on the back of the Xbox 360 controller and these return analog values (in comparison to the top back buttons – the shoulder buttons – which are digital and just return 'pressed' or 'not pressed'). The values returned by the triggers are unsigned 8-bit integers (range 0-255 where 0 is not pressed and 255 is fully pressed). That's a little inconvenient but we can easily convert them to percentages like this:

```
float leftTrigger = (float) state.Gamepad.bLeftTrigger / 255;
float rightTrigger = (float) state.Gamepad.bRightTrigger / 255;
```

This will give you decimal values from 0-1 for each trigger depending on how much the user is pressing them in. The raw trigger values can be found in *state.Gamepad.bLeftTrigger* and *state.Gamepad.bRightTrigger* respectively.

How to test the position of each analog thumb stick

The two analog thumb sticks have values each denoting their positions on the X and Y axis, where the centre point (when the stick is not being touched by the user) is (0,0). X increases to the right and Y increases upwards, so when the stick is pointed left or down, the X and Y axes have negative values respectively. The values returned by the thumb sticks are signed 16-bit integers (range -32768 to +32767 where -32768 is fully to the left or down, 0 is centered and 32767 is fully to the right or up, depending on the axis being queried). Once again we can convert these values to percentages:

```
float normLX = fmaxf(-1, (float) state.Gamepad.sThumbLX / 32767);
float normLY = fmaxf(-1, (float) state.Gamepad.sThumbLY / 32767);
```

(versions of Visual Studio prior to Visual Studio 2013 use a <math.h> header which isn't C99-compliant. If fmaxf() doesn't work in your compiler, try using max() instead)

The raw stick values can be found in *state.Gamepad.sThumbLX, state.Gamepad.sThumbLY* (left stick) and *state.Gamepad.sThumbRX, state.Gamepad.sThumbRY* (right stick). The example above processes the left stick.

There is a small twist here: we want values from -1 to 1. If we divide the returned values by 32768, we can never get a value of 1 when the stick is moved fully right or up as the maximum returned value is 32767. On the other hand, if we divide by 32767 as above, the lowest (most extreme) negative value will be slightly less than -1. We compensate by clamping the lowest possible value to -1 to ensure the returned range is between exactly -1 and +1.

If you try the code above, you will immediately spot a problem: even when the sticks are supposedly at rest, because they are so sensitive, they return various values (with my controller, values from -400 to 400 are common when the controller is just sitting still on the table). In a game which uses these analog values, this will cause strange jittery motion of the player and for them to slowly drift around seemingly randomly. To fix this, we have to introduce *deadzone* calculations.

How to calibrate deadzones for the analog sticks

A *deadzone* is an area of movement from the resting position of a gaming peripheral (for example a gamepad thumb stick, joystick or steering wheel) that is ignored and treated as if no movement occurred. This is handy to stop the controls from being over-sensitive to tiny movements relative to the resting position, for example when the user grips the controller but does not intend to actually move (if their thumb is resting on top of the thumb stick for example). The simple movement of air around a thumb stick can cause it to return non-zero values, so implementing a deadzone is essential to ensure such miniscule readings are not interpreted as player movements causing the drift effect mentioned above.

Taking the normalized -1 to +1 values of the sticks above, we may take a first stab at the problem like this:

```
float deadzoneX = 0.05f;
float deadzoneY = 0.02f;

float leftStickX = (abs(normLX) < deadzoneX ? 0 : normLX);
float leftStickY = (abs(normLY) < deadzoneY ? 0 : normLY);
```

This sets some hard-coded deadzones (5% on the X-axis and 2% on the Y-axis, which I found works well for my own Xbox 360 controllers – you may wish to check a range of controllers to be sure). If the amount of movement of the stick in either axis is below that axis's deadzone, the movement amount is set to zero, otherwise it is left untouched at its previous value.

This works, but there is a problem. Imagine the deadzones are set high at 80%. When the user moves the stick less than 80% from the center, a 0 movement value is returned as desired. But as soon as we hit 80%-100%, the movement value will be set to 80%-100%, potentially leading to a sudden very fast on-screen movement of the player once we leave the deadzone. What we would like to do instead is scale the portion of the stick outside the deadzone (where movement is allowed) so that movement in this region returns values from 0-1. For example, if the deadzone is 80%, then 80% movement should return 0, 85% movement should return 0.25, 90% movement should return 0.5 and so on.

Without boring you with the mathematical details, the following code does exactly that:

```
leftStickX = (abs(normLX) < deadzoneX ? 0 : (abs(normLX) - deadzone
leftStickY = (abs(normLY) < deadzoneY ? 0 : (abs(normLY) - deadzone

if (deadzoneX > 0) leftStickX /= 1 - deadzoneX;
if (deadzoneY > 0) leftStickY /= 1 - deadzoneY;
```

Essentially, if the stick axis movement is outside the deadzone, we first get the absolute (positive) movement amount and subtract the deadzone amount from it. If the deadzone is 80%, this leaves us with values from 0-0.2 (0%-20%). The multiplication restores the sign (positive or negative) of the movement, since *x / abs(x)* always gives 1 if x is positive and -1 if x is negative. Finally, the *if* statements scale up the movement to a value between 0 and 1. With an 80% deadzone, we will have values of 0-0.2 and they will be divided by 1 – 0.8 = 0.2, which is the same as multiplying by 5. Clearly, 0.2 (the maximum value of

movement with an 80% deadzone once the deadzone has been subtracted) * 5 = 1, leading to a correct linear scaling of all the possible movement values outside the deadzone.

If you don't understand the math, don't worry – just copy and paste the code.

Rolling it all together into a Gamepad class

Here is a simple class which puts together everything we have learned so far:

```
+ expand source
```

At the start of your application, create an instance of *Gamepad*. Once per frame (or when you check for keyboard events), call *Refresh()*. This will get the state of the currently connected controller, or check to see if a new controller has been connected if none was available on the previous call to *Refresh()*. It will also deal gracefully with disconnects. After you call *Refresh()*, you can use *GetPort(), GetState(), IsPressed(WORD button)* and the public fields *leftStickX, leftStickY, rightStickX, rightStickY, leftTrigger* and *rightTrigger* to get the analog stick/trigger values. The *Gamepad* constructor allows you to set the deadzones, however since different devices work differently, I recommend you provide an option in your game to allow the user to change the stick deadzones in a controller settings dialog.

The following very simple text-based example shows how to use the class:

```
#include <iostream>
#include <Windows.h>
#include <Xinput.h>

using std::cout;
using std::endl;

/* Insert the Gamepad class code here */

int main()
{
Gamepad gamepad;

bool wasConnected = true;

while (true)
{
Sleep(100);

if (!gamepad.Refresh())
{
if (wasConnected)
{
wasConnected = false;

cout << "Please connect an Xbox 360 controller." << endl;
}
}
else
{
if (!wasConnected)
{
wasConnected = true;

cout << "Controller connected on port " << gamepad.GetPort() << enc
}
```

```
cout << "Left thumb stick: (" << gamepad.leftStickX << ", " << game

cout << "Left analog trigger: " << gamepad.leftTrigger << "   Right

if (gamepad.IsPressed(XINPUT_GAMEPAD_A)) cout << "(A) button presse
}
}
}
```

Wait, is that it?!

Well, not quite. As you can see, XInput makes it very simple to add gamepad support to an existing game. If you are currently using *GetAsyncKeyState()* in your games to check for key presses, this may well be all you need. You can simply add calls to *Gamepad::IsPressed()* at the same point in the code to check for controller button presses. If you have some key which causes acceleration to the player, you can use the percentage returned by one of the analog sticks as a multiplier to the standard amount of acceleration.

Things get a bit more complicated when we are using Windows keyboard messages (*WM_KEYDOWN* and *WM_KEYUP*) to handle keyboard events. This method has the advantage that it is event-driven instead of requiring keyboard polling, but the downside is that keys are handled in the standard user interface way, ie. with a repeat timer and repeat count instead of showing the absolute down/up state of the key. The type of input mechanism to use depends somewhat on the game type, but if you are using Windows messages for keyboard input, you now have a problem, because XInput requires you to poll the controller at regular intervals as we have seen above. The solution is to write code which translates gamepad button presses and analog movement into Windows keyboard events, and is the subject of part 2 of this mini-series.

I hope you found the tutorial useful, and good luck with your game programming!

I'm a software developer with very limited work capacity due to having the debilitating illness M.E. – please read my article Dying with M.E. as a Software Developer and donate to the crowdfund to help me with my bucket list if you found this article useful. Thank you so much!

Useful Links

XInput Game Controller APIs @ MSDN – this is a reference to the entire XInput API including how to do vibration, check the controller's battery level and so on