Getting Started With XInput in Windows applications

Article • 10/23/2023

XInput enables Windows applications to process controller interactions (including controller rumble effects and voice input and output).

This topic provides a brief overview of the capabilities of XInput and how to set it up in an application. It includes the following:

Introduction to XInput

Applications can use the XInput API to communicate with gaming controllers when they are plugged into a Windows PC (up to four unique controllers can be plugged in at a time).

Using this API, any compatible connected controller can be queried for its state, and vibration effects can be set. Controllers that have the headset attached can also be queried for sound input and output devices that can be used with the headset for voice processing.

Controller Layout

Compatible controllers have two analog directional sticks, each with a digital button, two analog triggers, a digital directional pad with four directions, and eight digital buttons. The states of each of these inputs are returned in the **XINPUT_GAMEPAD** structure when the **XInputGetState** function is called.

The controller also has two vibration motors to supply force feedback effects to the user. The speeds of these motors are specified in the **XINPUT_VIBRATION** structure that is passed to the XInputSetState function to set vibration effects.

Optionally, a headset can be connected to the controller. The headset has a microphone for voice input, and a headphone for sound output. You can call the XInputGetAudioDeviceIds or legacy XInputGetDSoundAudioDeviceGuids function to obtain the device identifiers that correspond to the devices for the microphone and headphone. You can then use the Core Audio APIs to receive voice input and send sound output.

Using XInput

Using XInput is as simple as calling the XInput functions as required. Using the XInput functions, you can retrieve controller state, get headset audio IDs, and set controller rumble effects.

Multiple Controllers

The XInput API supports up to four controllers connected at any time. The XInput functions all require a *dwUserIndex* parameter that is passed in to identify the controller being set or queried. This ID will be in the range of 0-3 and is set automatically by XInput. The number corresponds to the port that the controller is plugged into, and is not modifiable.

Each controller displays which ID it is using by lighting up a quadrant on the "ring of light" in the center of the controller. A *dwUserIndex* value of 0 corresponds to the top-left quadrant; the numbering proceeds around the ring in clockwise order.

Applications should support multiple controllers.

Getting Controller State

Throughout the duration of an application, getting state from a controller will probably be done most often. From frame to frame in a game application, state should be retrieved and game information updated to reflect the controller changes.

To retrieve state, use the XInputGetState function:

C++

```
DWORD dwResult;
for (DWORD i=0; i< XUSER_MAX_COUNT; i++ )
{
    XINPUT_STATE state;
    ZeroMemory( &state, sizeof(XINPUT_STATE) );
```

```
        // Simply get the state of the controller from XInput.
        dwResult = XInputGetState( i, &state );

        if( dwResult == ERROR_SUCCESS )
        {
            // Controller is connected
        }
        else
        {
            // Controller is not connected
        }
    }
```

Note that the return value of XInputGetState can be used to determine if the controller is connected. Applications should define a structure to hold internal controller information; this information should be compared against the results of **XInputGetState** to determine what changes, such as button presses or analog controller deltas, were made that frame. In the above example, *g_Controllers* represents such a structure.

Once the state has been retrieved in a XINPUT_STATE structure, you can check it for changes and get specific information about controller state.

The *dwPacketNumber* member of the XINPUT_STATE structure can be used to check if the state of the controller has changed since the last call to XInputGetState. If *dwPacketNumber* does not change between two sequential calls to **XInputGetState**, then there has been no change in state. If it differs, then the application should check the *Gamepad* member of the **XINPUT_STATE** structure to get more detailed state information.

For performance reasons, don't call XInputGetState for an 'empty' user slot every frame. We recommend that you space out checks for new controllers every few seconds instead.

Dead Zone

In order for users to have a consistent gameplay experience, your game must implement dead zone correctly. The dead zone is "movement" values reported by the controller even when the analog thumbsticks are untouched and centered. There is also a dead zone for the 2 analog triggers.

ⓘ Note

Games that use XInput that do not filter dead zone at all will experience poor gameplay. Please note that some controllers are more sensitive than others, thus the dead zone may vary from unit to unit. It is recommended that you test your games with several Xbox controllers on different systems.

Applications should use "dead zones" on analog inputs (triggers, sticks) to indicate when a movement has been made sufficiently on the stick or trigger to be considered valid.

Your application should check for dead zones and respond appopriately, as in this example:

C++

```cpp
XINPUT_STATE state = g_Controllers[i].state;

float LX = state.Gamepad.sThumbLX;
float LY = state.Gamepad.sThumbLY;

//determine how far the controller is pushed
float magnitude = sqrt(LX*LX + LY*LY);

//determine the direction the controller is pushed
float normalizedLX = LX / magnitude;
float normalizedLY = LY / magnitude;

float normalizedMagnitude = 0;

//check if the controller is outside a circular dead zone
if (magnitude > INPUT_DEADZONE)
{
    //clip the magnitude at its expected maximum value
    if (magnitude > 32767) magnitude = 32767;

    //adjust magnitude relative to the end of the dead zone
    magnitude -= INPUT_DEADZONE;

    //optionally normalize the magnitude with respect to its expected
range
    //giving a magnitude value of 0.0 to 1.0
    normalizedMagnitude = magnitude / (32767 - INPUT_DEADZONE);
}
else //if the controller is in the deadzone zero out the magnitude
{
    magnitude = 0.0;
    normalizedMagnitude = 0.0;
}

//repeat for right thumb stick
```

This example calculates the controller's direction vector and how far along the vector the controller has been pushed. This allows enforcement of a circular deadzone by simply checking whether the controller's magnitude is greater than the deadzone value. In addition the code normalizes the controller's magnitude which can then be multiplied by a game specific factor to convert the controller's position to units relevant to the game.

Note that you may define your own dead zones for the sticks and triggers (anywhere from 0-65534), or you may use the provided deadzones defined as XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE, XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE, and XINPUT_GAMEPAD_TRIGGER_THRESHOLD in XInput.h:

C++

```
#define XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE  7849
#define XINPUT_GAMEPAD_RIGHT_THUMB_DEADZONE 8689
#define XINPUT_GAMEPAD_TRIGGER_THRESHOLD    30
```

Once the deadzone is enforced, you may find it useful to scale the resulting range [0.0..1.0] floating point (as in the example above), and optionally apply a non-linear transform.

For example, with driving games, it may be helpful to cube the result to provide a better feel to driving the cars using a gamepad, as cubing the result gives you more precision in the lower ranges, which is desirable, since gamers typically either apply soft force to get subtle movement or apply hard force all the way in one direction to get rd response.

Setting Vibration Effects

In addition to getting the state of the controller, you may also send vibration data to the controller to alter the feedback provided to the user of the controller. The controller contains two rumble motors that can be independently controlled by passing values to the XInputSetState function.

The speed of each motor can be specified using a WORD value in the XINPUT_VIBRATION structure that is passed to the XInputSetState function as follows:

C++

```
XINPUT_VIBRATION vibration;
ZeroMemory( &vibration, sizeof(XINPUT_VIBRATION) );
vibration.wLeftMotorSpeed = 32000; // use any value between 0-65535 here
vibration.wRightMotorSpeed = 16000; // use any value between 0-65535 here
XInputSetState( i, &vibration );
```

Note that the right motor is the high-frequency motor, the left motor is the low-frequency motor. They do not always need to be set to the same amount, as they provide different effects.

Getting Audio Device Identifiers

The headset for a controller has these functions:

- Record sound using a microphone
- Play back sound using a headphone

Use this code to obtain the device identifiers for the headset:

C++

```
WCHAR renderId[ 256 ] = {0};
WCHAR captureId[ 256 ] = {0};
UINT rcount = 256;
UINT ccount = 256;

XInputGetAudioDeviceIds( i, renderId, &rcount, captureId, &ccount );
```

After you obtain the device identifiers, you can create the appropriate interfaces. For example, if you use XAudio 2.8, use this code to create a mastering voice for this device:

C++

```
IXAudio2* pXAudio2 = NULL;
HRESULT hr;
if ( FAILED(hr = XAudio2Create( &pXAudio2, 0, XAUDIO2_DEFAULT_PROCESSOR )
) )
    return hr;

IXAudio2MasteringVoice* pMasterVoice = NULL;
if ( FAILED(hr = pXAudio2->CreateMasteringVoice( &pMasterVoice,
XAUDIO2_DEFAULT_CHANNELS, XAUDIO2_DEFAULT_SAMPLERATE, 0, renderId, NULL,
AudioCategory_Communications ) ) )
    return hr;
```

For info about how to use the captureId device identifier, see Capturing a Stream.

Getting DirectSound GUIDs (legacy DirectX SDK only)

The headset that can be connected to a controller has two functions: it can record sound using a microphone, and it can play back sound using a headphone. In the XInput API, these functions are accomplished through DirectSound, using the **IDirectSound8** and **IDirectSoundCapture8** interfaces.

To associate the headset microphone and headphone with their appropriate DirectSound interfaces, you must get the DirectSoundGUIDs for the capture and render devices by calling XInputGetDSoundAudioDeviceGuids.

> **Note**
>
> Use of the legacy DirectSound is not recommended, and is not available in Windows Store apps. The info in this section only applies to the DirectX SDK version of XInput (XInput 1.3). The Windows 8 version of XInput (XInput 1.4) exclusively uses Windows Audio Session API (WASAPI) device identifiers that are obtained through XInputGetAudioDeviceIds.

C++

```cpp
XInputGetDSoundAudioDeviceGuids( i, &dsRenderGuid, &dsCaptureGuid );
```

Once you have retrieved the GUIDs you can create the appropriate interfaces by calling DirectSoundCreate8 and DirectSoundCaptureCreate8 like this:

C++

```cpp
// Create IDirectSound8 using the controller's render device
if( FAILED( hr = DirectSoundCreate8( &dsRenderGuid, &pDS, NULL ) ) )
   return hr;

// Set coop level to DSSCL_PRIORITY
if( FAILED( hr = pDS->SetCooperativeLevel( hWnd, DSSCL_NORMAL ) ) )
   return hr;

// Create IDirectSoundCapture using the controller's capture device
if( FAILED( hr = DirectSoundCaptureCreate8( &dsCaptureGuid, &pDSCapture,
NULL ) ) )
   return hr;
```

Related topics

Programming Reference

Feedback

Was this page helpful? Yes No