Accessing Microsoft Windows* 8 Desktop Sensors

Abstract

This sample demonstrates how Windows 8 Desktop applications can use the sensors that are available in Intel® Ultrabook™ PCs and tablets. The sample consists of two projects, the Sensor Manager and Windows Sensors. The Sensor Manager library is an interface to get the required data from the sensors. It was designed to be standalone, and has already been integrated into 3rd party game titles. The Windows Sensors project is a very simple 3D "game" demonstrating how to use the library.

The Sensor Manager Library

The Sensor Manager library was created as an experiment in creating a standalone module that would provide an extensible sensor manager that could be used with DirectX applications, especially full screen games. The sensor manager needed to be able to support the full range of sensors available on Ultrabooks, and due to the need for most games to be backward-compatible with Windows 7 the sensor manager had to work with both Windows 7* and Windows 8 Desktop. Sensors on Windows are treated as Plug and Play devices; one of the goals of the Sensor Manager library was to make something robust enough to cope with the full life cycle of a potentially detachable sensor.

Games that treat sensors as always available, ignoring these events risk providing an unsatisfactory experience as a lost device might render the game unplayable. The inability to use future control devices limits the game's future appeal. The sensor manager is designed to cope with the loss of a previously detected device and report its status and provide sensible default data to the host

application allowing for a seamless user experience.

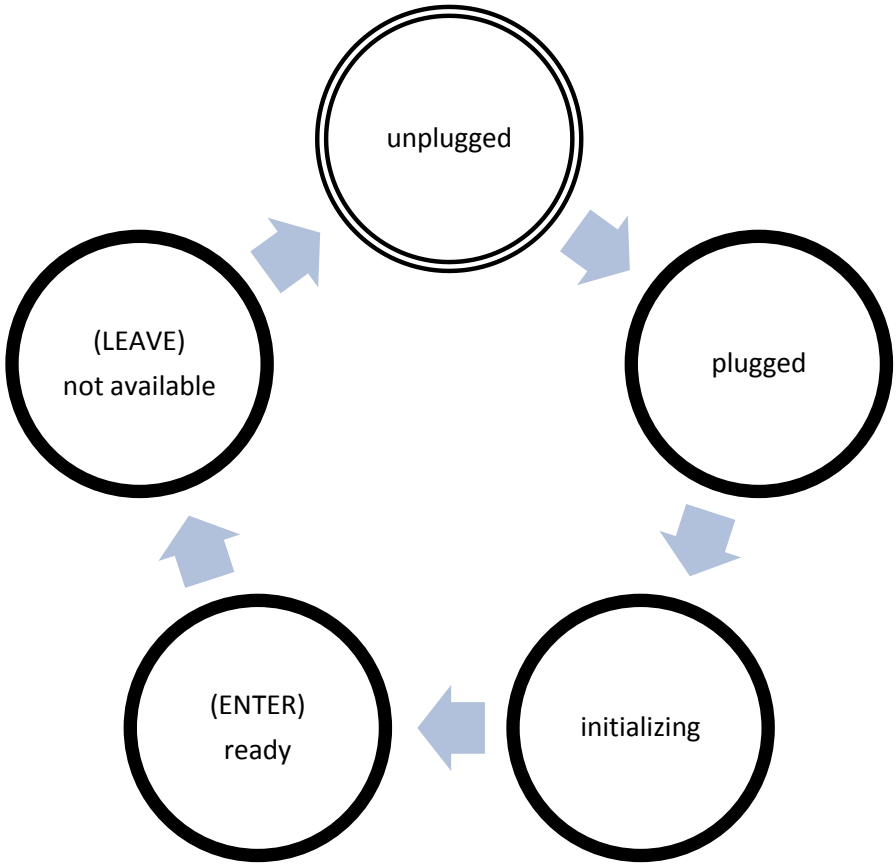

Figure 1: Life cycle of a sensor

Figure 1 shows the life cycle of a sensor. Plug and Play connect is called an Enter event, and disconnect is called a Leave event.

At first this might seem excessive, as the assumption most people make is sensors will be built into an Ultrabook and therefore always available. In reality there are a few different scenarios where sensors can be connected/disconnected:

1. It is possible to have USB-based sensors external to the system and plugged in to a USB port.
2. It is conceivable to have sensors that are attached by an unreliable wireless interface (such as Bluetooth*) or wired interface (such as Ethernet), where connects and disconnects are common.
3. If a Windows Update upgrades the device driver for the sensors, they will appear to disconnect and then reconnect.
4. When Windows shuts down (to S4 or S5), the sensors appear to disconnect.

The sensor manager encapsulates the native Win32*/COM API within a core manager that enumerates the attached sensors, and manages the life time of any sensors that receive the plug and play Enter event when a new sensor is attached. The first task on creation of the sensor manager is to register itself to receive callbacks when new sensors are detected. In C++/COM, you

use the *SetEventSink* method to hook the callback. The sensor manager inherits from *ISensorManagerEvents* and implements *IUnknown* and a callback for *OnSensorEnter(ISensor *pSensor, SensorState state);* The sensor manager then enumerates all currently attached sensors using the code in Figure 2.

```
HRESULT CSensorManagerEvents::Initialize(int NumSensorTypes, ...)
{
	ISensorCollection* spSensors;
	ISensorCollection *pSensorCollection = NULL;
	ULONG ulCount = 0;
	hr = ::CoCreateInstance(CLSID_SensorManager, NULL, CLSCTX_INPROC_SERVER,
IID_PPV_ARGS(&m_spISensorManager));
	hr = m_spISensorManager->SetEventSink(this);
	hr = m_spISensorManager->GetSensorsByCategory(SENSOR_CATEGORY_ALL,
&spSensors);
	hr = spSensors->GetCount(&ulCount);
	for(ULONG i=0; i < ulCount; i++)
		{
			ISensor* spSensor;
			hr = spSensors->GetAt(i, &spSensor);
		// Check Permissions
		// OnSensorEnter (spSensor,state)
	}
}
```

Figure 2: Initialization code

The application passes in a list of all the sensors it is actually interested in using. This avoids creating handlers for unneeded attached sensors. Note the need to check for permissions on the requested sensors; under Windows 7, the user can disable the sensors from the Control Panel and any application should honor this and give the user the opportunity to enable the sensors when first run.

Sensors that are needed are registered with the manager using the code in Figure 3.

```
HRESULT CSensorManagerEvents::OnSensorEnter(ISensor* pSensor, SensorState state)
{
    if (NULL != pSensor)
    {
       hr = AddSensor(pSensor);
    }
}

HRESULT CSensorManagerEvents::AddSensor(ISensor *pSensor)
{
	if (NULL != pSensor)
	 {
		pSensor->GetState(&state);
		hr = pSensor->GetType(&idType);
		if (state == SENSOR_STATE_ACCESS_DENIED)
			return E_FAIL;
```

```
            if (IsEqualIID(idType, SENSOR_TYPE_INCLINOMETER_3D))
            {
                  if(CInclinometer::ValidateOutput(pSensor) != S_OK)
                        return E_FAIL;
            }
            hr = pSensor->SetEventSink(m_SensorEvents);
            GUID pguid[1] = {SENSOR_EVENT_STATE_CHANGED};
            hr = pSensor->SetEventInterest(pguid,1);

            //  Create Internal class to track this sensor
      }
}
```

Figure 3: Registering sensors

Before sensors are registered, the code validates the type of output they provide since a sensor might provide data in a different format to what the calling application requests. One example is a device orientation sensor that could provide either a quaternion or a matrix as its output or both. *SetEventSink* is used to assign a callback to an object to receive any windows messages relating to a particular sensor such as *OnLeave*. The code then uses *SetEventInterest* to limit the number of messages received. This was done to remove the data changed events that would come from asynchronously reading the sensors. The sensor manager implements a direct polling of the sensors to reduce lag, on the assumption most games would need to read the sensors at regular intervals.

Removal of a sensor is either done directly from the manager, as shown in Figure 4, or from a call-back responding to a OnLeave message when a sensor is lost, as shown in Figure 5.

```
HRESULT CSensorManagerEvents::RemoveSensor(ISensor* pSensor)
{
      HRESULT hr = S_OK;
      if (NULL != pSensor)
      {
                   hr = pSensor->SetEventSink(NULL); // This also decreases the
ref count of the sink object.
                  SENSOR_ID idSensor = GUID_NULL;
                  hr = pSensor->GetID(&idSensor);
            RemoveSensor(idSensor);
       }
}
```

Figure 4: Removing a sensor

```
HRESULT CBaseSensorEvents::OnLeave(REFSENSOR_ID sensorID)
{
    HRESULT hr = S_OK;
    hr = m_pSensorManagerEvents->RemoveSensor(sensorID); // Callback into parent
    return hr;
```

```
}
```

Figure 5: Callback when losing a sensor

The polling of requested sensors is done by finding the requested sensor in the manager and passing a *ISensorDataReport* structure to be filled out as shown in Figure 6.

```
HRESULT COrientationDevice::OnDataUpdated(ISensor *pSensor, ISensorDataReport
*pDataReport, void* pData)
{
   PROPVARIANT pvRot;
    HRESULT hr = S_OK;
    If (NULL != pSensor && NULL != pDataReport)
    {
              SENSOR_ID idSensor = GUID_NULL;
              hr = pSensor->GetID(&idSensor);

      PropVariantInit(&pvRot);

      hr = pDataReport->GetSensorValue(SENSOR_DATA_TYPE_ROTATION_MATRIX, &pvRot);
      if (pvRot.vt == (VT_UI1|VT_VECTOR))
      {
             // Copy data to pData
      }
      PropVariantClear(&pvRot);
    }
    return hr;
}
```

Figure 6: Polling a sensor

The code directly polls the sensors, rather than using an asynchronous system similar to that found in most Microsoft Windows 7 API samples. In an asynchronous system, the driver submits new data into a queue for processing by various services running on the PC. This eventually results in the data changed message which is processed by the application's windows message loop. Between the driver submitting the data and the application receiving it the windows kernel will do various preprocessing of the data. In a typical game, the application's message loop would be processed once per frame, which at 30FPS is once every 33ms, using something similar to Figure 7.

```
// Main message loop
MSG msg = {0};
while( WM_QUIT != msg.message )
{
    if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
```

```
    else
    {
        Render();
    }
}
```

Figure 7: Windows Message Polling

If two or more sensors are used by an application, then the windows preprocessing can mean multiple messages from sensor 1 can arrive during one frame while multiple messages from sensor 2 might arrive a frame later. This is contrary to the expectation that the messages would arrive in the same order that they occurred, regardless of the sensor which sent them. The messages are time stamped but the delay in receiving them can lead to a perceived unresponsiveness.

The polling approach provided more consistent behavior with much reduced lag. While not as power efficient as an asynchronous system, the power impact in the context of a 3D game would be negligible, especially in a situation where you are expecting the sensor data to be changing continually, such as a game's control system.

The Sample Game

The Sample Game is a very simple application of the Sensor Manager library. A bike is driven around an arena by tilting a device with an inclinometer (tilt sensor). The code demonstrates how to properly initialize and poll the sensors, along with how to use the Sensor Manager to determine if a sensor has become disconnected. In the case of this particular sample, the sensor manager finds the first available 3D INCLINOMETER and uses this to control the bike, with any additional inclinometers found selectable from a drop down menu. The RAW inclinometer data is displayed as part of the UI. The RAW data varies between +/-180 on the X axis used to control the bike's direction and +/-90 around the Y-Axis used to control acceleration. More detail of how the inclinometer reports its data can be seen in Figure 8.

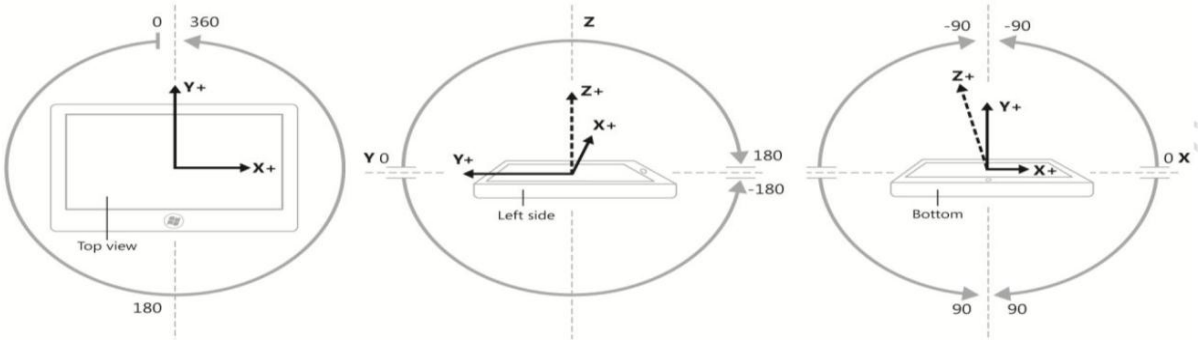

Figure 8: Return angles for inclinometer

The inclinometer data is read in the *void WindowsSensor::Update(double deltaSeconds)* function once per frame. Once read, it is applied to the bike's velocity and turning rate once adjusted for the amount of time elapsed between updates. Finally, a calibration tool has been added, where the user can "zero" the sensor, giving it a new resting point. This allows the user to re-calibrate the device for his or herown comfort.

Conclusion

The Windows Sensors API can seem daunting and complex. This sample demonstrates how to use the Microsoft-provided interfaces to use any and all sensors a device might have. The Sensor Manager framework is ready to be dropped into an existing application, or used as a starting point for implementing a custom solution.

About the Authors

Kyle Weicht is a software engineer in the Intel Software and Services Group, where he supports Intel graphics solutions in the Developer Relations Division. He holds a B.S. in Game Development from Full Sail University.

Leigh Davies is a senior application engineer at Intel with over 15 years' programming experience in the PC gaming industry. He is a member of the European Visual Computing Software Enabling Team providing technical support to game developers, areas of expertise include 3D graphics and recently touch and sensors.

References

http://software.intel.com/en-us/blogs/2012/07/26/detecting-ultrabook-sensors

http://software.intel.com/en-us/articles/ultrabook-and-tablet-windows-8-sensors-development-guide

http://msdn.microsoft.com/en-us/library/windows/desktop/dd318953(v=vs.85).aspx

http://msdn.microsoft.com/en-us/library/windows/hardware/hh406647(v=vs.85).aspx

http://blogs.msdn.com/b/jimtravis/

Notices