Windows Phone Native C++ and DirectX - First Direct3D App, setting up Touch and Sensors

11 Nov 2012

Introduction

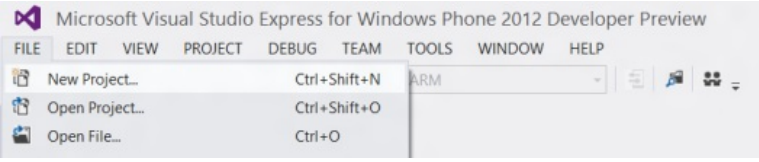
In this article we go through the basic steps of creating a first native Direct3D app. We take a look at the files created in the project template and also show you how to utilize touch input and Windows Phone sensors i.e. accelerometer and gyroscope.

Prerequisites

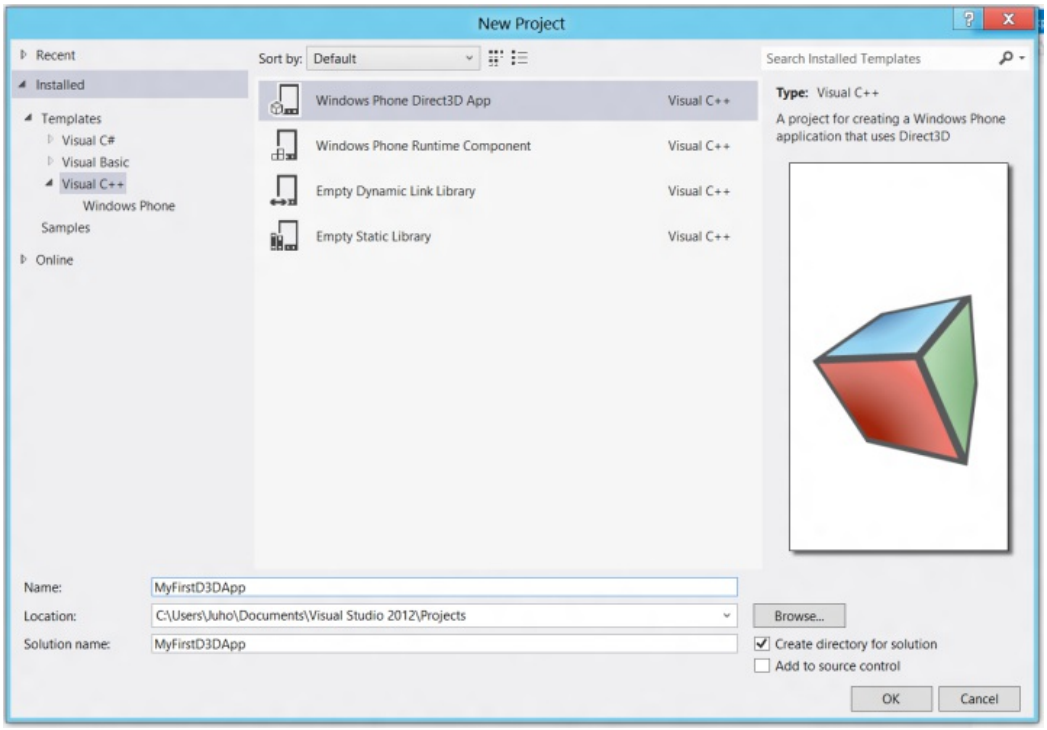
Before going further, make sure you have the Windows Phone 8 SDK with at least Visual Studio 2012 Express installed.

Creating a new Direct3D solution in Visual Studio

To begin creating a new native Direct3D application, start up your Visual Studio 2012 and select **FILE| New Project...** .



From the New Project –window, from the list on the left side select **Templates** and then the **Visual C++** -sub-item. Now select **Windows Phone Direct3D App** from the center list.

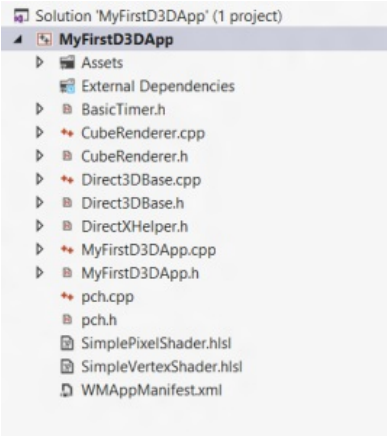


On the lower side of the window, you can set the name of your application. Let's name this app **MyFirstD3DApp**. Now just click **OK** and Visual Studio will create a new solution.

Template solution contents

Now we can take a look at the solution structure and the files the template has created.

(c) Copyright Nokia 2013. All rights reserved.



First under the solution is the main project for our application, named with the name we provided in the creation. Under this project we have all the code files and other assets used in the project. Here's a small description for each of these.

Assets-folder: Includes all the current and future non-code assets of your application, such as graphics textures, sound files and videos. In the beginning this folder contains images that are used to display the application icon on different menus of the Windows Phone OS, such as the main tile screen and the application list menu.

External Dependencies –folder: This folder is automatically generated and updated to include all the header files for the libraries used in your application. For example, if you were to include the standard <string.h> in one of your code files, it would appear here.

BasicTimer.h: Like the name suggests, this is a basic timer class. Timers are needed to support many functionalities of a game application. It includes functionality for resetting and updating time and also getting a total time and a time delta.

CubeRenderer.cpp/CubeRenderer.h: Define the CubeRenderer class which inherits the Direct3DBase (explained below) class. Like the class name suggests, the rendering (drawing to the screen) of the 3D cube is done in this class. Various Direct3D functions are set up here, such as shaders, buffers and the dimensions of the cube model.

Direct3DBase.cpp/Direct3DBase.h: Definition for the abstract Direct3D base class which renderer classes (such as the CubeRenderer) inherit. It includes all the Direct3D objects that the child classes share, such as D3DDevice, D3DContext and SwapChain.

DirectXHelper.h: This header has two helper functions for DirectX. ThrowIfFailed()-function takes a DirectX function that returns a HRESULT as a parameter. If the HRESULT is a failure, the function automatically throws a corresponding exception. The other function is ReadDataAsync() which reads data from a binary file asynchronously. It is used in the CubeRenderer-class.

MyFirstD3DApp.cpp/MyFirstD3DApp.cpp: The main application code files (includes the main-function). The application is initialized and its application window set up here. These files also include the event handlers for events like application suspending and resuming and touch events. The renderer class (in this case CubeRenderer) is also created here.

Pch.cpp/pch.h: Precompiled headers are included here. These files aren't necessarily needed if the libraries included in them are included elsewhere in the project.

SimplePixelShader.hlsl and SimpleVertexShader.hlsl are both used in Direct3D object rendering. Both of the shaders are compiled from these files and they are loaded in the CubeRenderer.

WMAppManifest.xml: This file contains details about the application. From it the developer can for example set a custom application icon, change the display name of the app, set the application capabilities and minimum hardware requirements and also change the version number of the app for future updates.

How to use touch input

Getting touch input in a native C++ application is quite straightforward when using the project template, because the event handlers for touch events have already been introduced in it. This leaves only the task of deciding how to use the touch event data to the developer.

Let's go to **MyFirstD3DApp.h** and look at the class functions. Inside protected functions you will find three function declarations related to touch events: OnPointerPressed(), OnPointerReleased() and OnPointerMoved. These three event handlers are called depending on what type of touch event has occurred: OnPointerPressed when the device user first lands a finger or maybe a stylus on the touch surface. If the user starts moving the finger or stylus after it's been pressed to the device, the OnPointerMoved is called. Finally, when the user lifts the finger from the touch surface, OnPointerReleased is called. Generally all touches will have at least a pressed and released event.

All three event handling functions take two parameters. The first one is the sender of the touch event, in this case the CoreWindow-class on which the application is run on and which has the touch listener. The second parameter is the touch event arguments given as a PointerEventArgs class which contains the data for the event; in this case the screen coordinates of the touch.

When we switch to **MyFirstD3DApp.cpp** we find the definitions of the three above functions. The definitions are empty by default and it is meant for the developer to add his own custom event handling inside them.

For this example, we are going to simply print out the touch coordinates on the debug output window. To print strings to the output window in debug mode in WP8, we will use the function OutputDebugStringA. It takes a LPCSTR (a constant char) as a parameter. First, to convert the touch coordinates to a string we are going to read them into a ostringstream, then convert it to a string, and finally pass it converted to const char*with c_str() to the OutputDebugStringA.

(C) Copyright Nokia 2013. All rights reserved.

To get the touch coordinates from the PointerEventArgs class the event handler received as a parameter, we use the class's CurrentPoint member call to get a PointerPoint class out of it. After this we call the PointerPoint's Position member which then has the X and Y positions of the touch event as floats. We read these two positions into the ostringstream and print them in the output window like explained above.

First, add these two includes to the beginning of the file to be able to use the ostringstream and string functions, and also add the std namespace:

```
#include <string.h>
#include <sstream>

using namespace std;
```

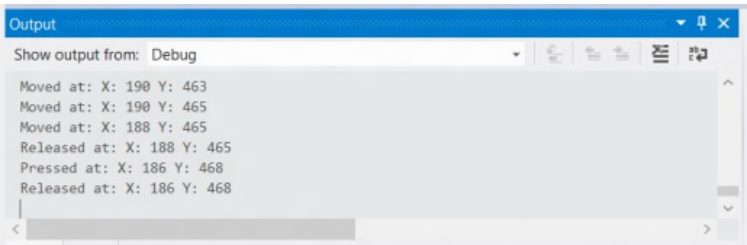
And now add the code inside the touch event handler functions:

```
void PhoneDirect3DApp1::OnPointerPressed(CoreWindow^ sender, PointerEventArgs^ args)
{
 ostringstream sstream;
 sstream << "Pressed at: " << "X: " << args->CurrentPoint->Position.X << " Y: " << args->CurrentPoint->Position.Y << "\n";
 string s = sstream.str();
 OutputDebugStringA(s.c_str());
}

void PhoneDirect3DApp1::OnPointerReleased(CoreWindow^ sender, PointerEventArgs^ args)
{
     ostringstream sstream;
 sstream << "Released at: " << "X: " << args->CurrentPoint->Position.X << " Y: " << args->CurrentPoint->Position.Y << "\n";
 string s = sstream.str();
 OutputDebugStringA(s.c_str());
}

void PhoneDirect3DApp1::OnPointerMoved(CoreWindow^ sender, PointerEventArgs^ args)
{
 ostringstream sstream;
 sstream << "Moved at: " << "X: " << args->CurrentPoint->Position.X << " Y: " << args->CurrentPoint->Position.Y << "\n";
 string s = sstream.str();
 OutputDebugStringA(s.c_str());
}
```

Now you can build and run the solution in debug mode. When the application starts on the emulator (or on the device if you are using one), try clicking on the screen. The output window should now show the coordinates of the touch. If the output window isn't visible, go to the **VIEW** menu and click on **Output** and it should appear on the lower side of the screen.



How to set up accelerometer/gyrometer

Next we are going to see how to get data out of the device's sensors in native C++. The method with both accelerometer and gyrometer is very similar and will only take a few lines of code.

Before we can use the sensors however, we have to grant the application permissions to access them on the device. We do this by going to the **WMAppManifest.xml**-file, opening the **Capabilities** tab and checking **ID_CAP_SENSORS**. With this the application now has permission to use the device's sensors it is deployed on. Without it, we would receive an access permission exception and the application would crash.

Accelerometer

The steps for utilizing the accelerometer are fairly simple. First we just introduce an object from the Accelerometer class which can be found under the Windows::Devices::Sensors namespace. To get the accelerometer data properly, it is recommended to introduce the object somewhere where it can be used in a function that is executed at regular intervals. In this example, we are going to use the accelerometer in the CubeRenderer class because we can use its Update-function.

First let's add a declaration for the Accelerometer object to our **CubeRenderer.h** file. We are also going to need an AcceleroMeterReading object to store the reading to.

```
private:
    …
    Windows::Devices::Sensors::Accelerometer^ m_accelerometer;
    Windows::Devices::Sensors::AccelerometerReading^ m_accReading;
```

Now that we've declared the class members, we can go to CubeRenderer.cpp. **In the constructor**, we are going to format the declared Accelerometer with the class's static function GetDefault() which returns a pointer of the device's accelerometer.

```
m_accelerometer = Windows::Devices::Sensors::Accelerometer::GetDefault();
```

After this we can move to CubeRenderer's Update-function. Here we will get the current reading of the accelerometer and store it in the m_accReading pointer using the Accelerometer's GetCurrentReading() method. To see the reading data, we will print it on the output window the same way we did in the touch input example.

(C) Copyright Nokia 2013. All rights reserved.

You can get the X, Y and Z axis acceleration data from the reading as doubles using the `AccelerationX`, `AccelerationY` and `AccelerationZ` properties.

Now it's just a matter of reading those doubles into an ostringstream and printing that stream with the OutputDebugStringA-method.

```
void CubeRenderer::Update(float timeTotal, float timeDelta)
{
    …
    if (m_accelerometer != nullptr)
    {
 try
        {
            m_accReading = m_accelerometer->GetCurrentReading();
     ostringstream sstream;
            sstream << "Acceleration: " << "X: " << m_accReading->AccelerationX << " Y: " << m_accReading->AccelerationY << " Z: " <
     string s = sstream.str();
     OutputDebugStringA(s.c_str());
        }
 catch(Platform::Exception^ e)
 {
     // there is a bug tracking this issue already
     // we need to remove this try\catch once the bug # 158858 hits our branch
     // For now, to make this App work, catching the exception
     // The reverting is tracked by WP8 # 159660
 }
    }
}
```

To see how to test the accelerometer on the emulator, see the "Testing the sensors on the emulator"-paragraph at the end of this article.

Gyrometer

The steps to add gyrometer functionality are exactly the same as with the accelerometer. First we introduce an object from the Gyrometer class which can be found under the same namespace as the accelerometer (Windows::Devices::Sensors). After that we store the readings to a GyrometerReading object and use those readings later in whatever way we require.

First set up the Gyrometer and GyrometerReading objects in Cuberenderer.h:

```
private:
    …
    Windows::Devices::Sensors::Gyrometer^ m_gyrometer;
    Windows::Devices::Sensors::GyrometerReading^ m_gyroReading;
```

Then in the **constructor in Cuberenderer.cpp**, we format the declared Gyrometer with the GetDefault() static function:

```
m_gyrometer = Windows::Devices::Sensors::Gyrometer::GetDefault();
```

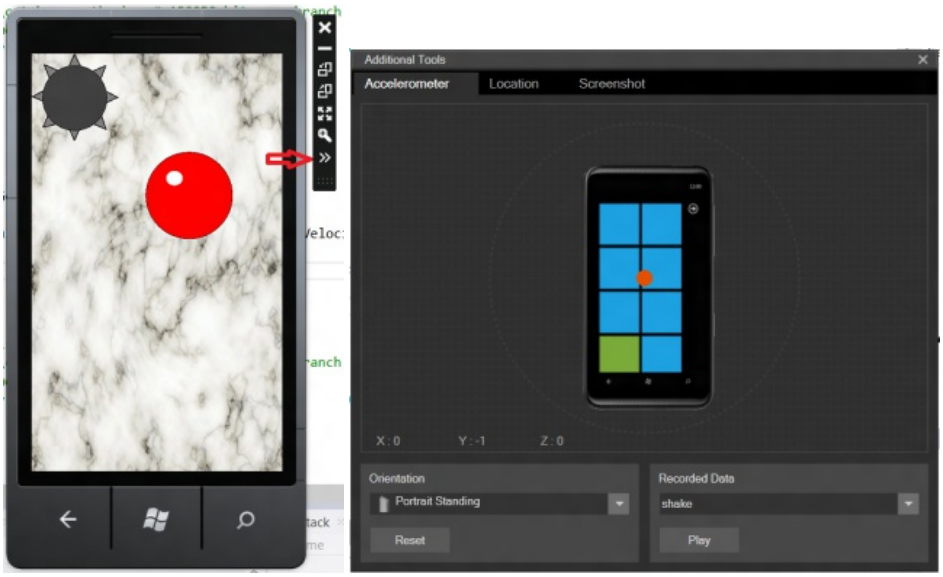
Now in the **Update-function**, we store the gyrometer reading to the m_gyroReading pointer with the gyrometer's GetCurrentReading() method and print it out to the output window as before.

```
void CubeRenderer::Update(float timeTotal, float timeDelta)
{
    …
    if (m_gyrometer != nullptr)
    {
        try
        {
            m_gyroReading = m_gyrometer->GetCurrentReading();
            ostringstream sstream;
            sstream << "Gyro angular velocity X: " << m_gyroReading->AngularVelocityX << " Y: " << m_gyroReading->AngularVelocityY <
            m_gyroReading->AngularVelocityZ << "\n";
            string s = sstream.str();
            OutputDebugStringA(s.c_str());
        }
 catch(Platform::Exception^ e)
 {
     // there is a bug tracking this issue already
     // we need to remove this try\catch once the bug # 158858 hits our branch
     // For now, to make this App work, catching the exception
            // The reverting is tracked by WP8 # 159660
        }
    }
}
```

Testing the Sensors on the Emulator

To simulate sensor activity on the emulator, you can simply open the **Additional Tools** window and go to the **Accelerometer-tab**. From here you can move the virtual phone by dragging the red dot. You can also change the orientation and simulate a phone shake from the drop-down menus and buttons in the bottom of the window.

(C) Copyright Nokia 2013. All rights reserved.



(c) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.