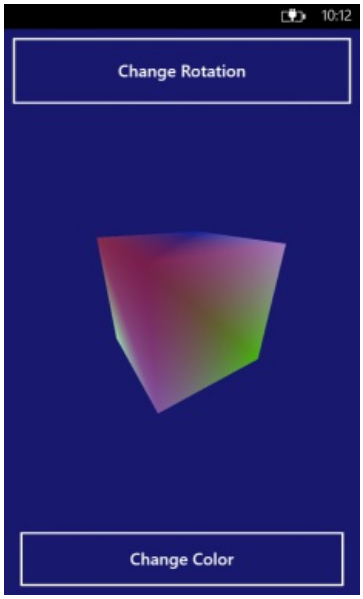
Windows Phone Direct3D XAML Application Introduction

This article introduces developers to the Direct3D XAML application template in Windows Phone 8.

Introduction

Windows Phone 8 offers the developer the option to create a Direct3D XAML interop application where the developer can combine Direct3D graphics with XAML interface. We are going to take a quick look at the Visual Studio 2012 Direct3D XAML Application template and make a small example on how to make the XAML/C# side of the application interact with the C++/Direct3D side. Possible uses for a Direct3D XAML application could for example be games where the developer wants to create the UI and game controls with XAML, or a smart app where some more high quality graphics are required.

At the end of this article we will have an example built on the template in which XAML UI buttons control the color and rotation direction of a Direct3D rendered cube (which is already implemented in the template).



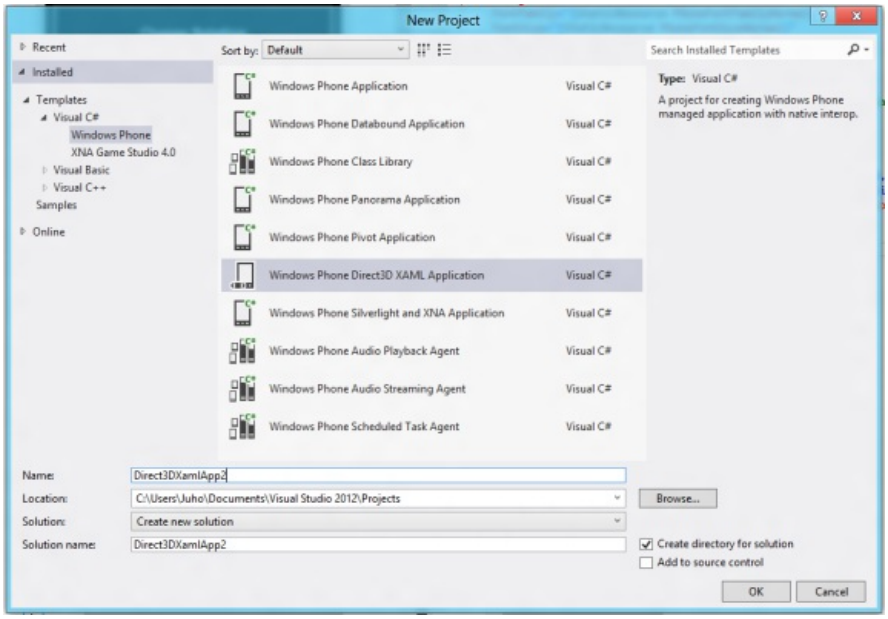
Prerequisites

It's recommended that before you read this article you have a basic understanding of Windows Phone Application development. Basic understanding of Direct3D is also recommended.

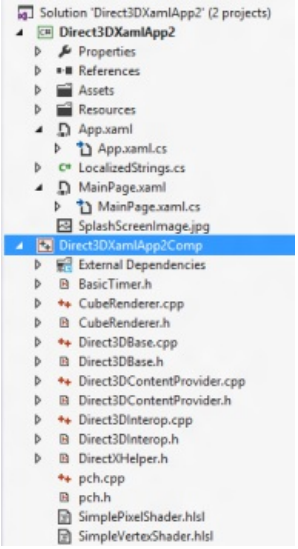
Visual Studio Direct3D XAML Application Template

The D3D XAML app template in Visual Studio can be created from **File || New Project... || Templates || Visual C# || Windows Phone || Windows Phone Direct3D XAML Application**. In this example the name Direct3DXamlApp2 is used for the solution.

(C) Copyright Nokia 2014. All rights reserved.



In it's default state the solution has two projects:



- **Direct3DXamlApp2:** A XAML/C# project similar to a normal Windows Phone Application project. It has all the .xaml ui design files and the .cs files corresponding to those.
- **Direct3DXamlApp2Comp:** Contains the Direct3D C++ files needed to set up and render Direct3D graphics. The contents are similar to the Native C++ Direct3D template contents.

The basic functionality between these two projects is that the -Comp project code creates a DrawingSurface object which can then be used as an XAML element in the XAML/C# project and can be modified the same way (for example resized and applied to a grid) as a normal XAML UI element. Other XAML elements, such as buttons and texts can be placed on top of the DrawingSurface element.

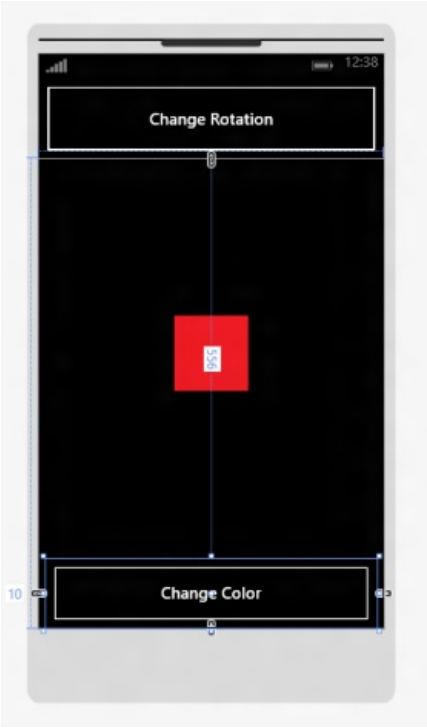
Example: Adding XAML Buttons to Control the Direct3D Rendered Cube

Next we are going to add some modifications to the template to showcase the interaction between XAML/C# and C++.

Creating the XML UI

First off, open the MainApp.xaml-file and add two buttons to the screen, one on top of the screen and the other in the bottom. Set the top button text as "Change Rotation" and the bottom button text as "Change Color". Also change the buttons' names to rotationButton and colorButton. Finally add a Click-property to both of the buttons with rotationButton's Click-property directing to "rotationButton_Clicked" event handler and colorButton's directing to "colorButton_Clicked". We will be adding these event handlers to the C#-code later on. Here's how the UI looks in the designer window:

(C) Copyright Nokia 2014. All rights reserved.



Alternatively you can just copy-paste this xml-code on top of the MainApp.xaml source:

```
<phone:PhoneApplicationPage
    x:Class="Direct3DXamlApp2.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <!--LayoutRoot is the root grid where all page content is placed-->
    <Grid x:Name="LayoutRoot" Background="Transparent"
          MouseLeftButtonDown="LayoutRoot_MouseLeftButtonDown"
          MouseLeftButtonUp="LayoutRoot_MouseLeftButtonUp"
          MouseMove="LayoutRoot_MouseMove" Margin="0,115,0,0">
        <DrawingSurface x:Name="DrawingSurface" Margin="0,-113,0,0" />
        <Button x:Name="rotationButton" Click="rotationButton_Click" Content="Change
Rotation" HorizontalAlignment="Left" Margin="0,-113,0,0" VerticalAlignment="Top"
Height="113" Width="480"/>
        <Button x:Name="colorButton" Click="colorButton_Click" Content="Change Color"
HorizontalAlignment="Left" Height="97" Margin="10,556,0,0" VerticalAlignment="Top"
Width="460"/>
    </Grid>

</phone:PhoneApplicationPage>
```

Adding C++ Methods

Next we will add methods to the C++ side of the application. We are going to be needing methods for changing the cube's rotation

(C) Copyright Nokia 2014. All rights reserved.

direction and for the color changing.

The C# code has a Direct3DInterop member created from the C++ class with the same name. The Direct3DInterop class then has a CubeRendeder object as a private member. To pass a command from the C# code, we need to add methods to both the Direct3DInterop class and the CubeRenderer class. In the C# code we then call the Direct3DInterop member's method which will then call the CubeRenderer member's method.

First, let's add the needed methods to CubeRenderer. We are going to call the methods ChangeRotation() and ChangeColor(). Their declarations in CubeRenderer.h look like this:

```
public:
    ...
    void changeRotation();
    void changeColor();
```

We also need to add three private members: a float for the cube's rotation degrees, a boolean member for rotation direction switching and an XMFLOAT3 pointer to which we store an array of XMFLOAT3's as the cube's vertex colors.

```
private:
    ...
    float rotation;
    boolean rotationNegative;
    XMFLOAT3* cubeColors;
```

Now we add the methods' definitions to CubeRenderer.cpp. In the changeRotation() method we simply switch the rotation boolean. In the changeColor() method we first randomize different RGB color values and then store them into the cubeColors XMFLOAT3 array and finally we create a new D3D vertex buffer and pass the new color values to it.

```
void CubeRenderer::changeRotation()
{
 if(rotationNegative)
  rotationNegative = false;
 else
  rotationNegative = true;
}

void CubeRenderer::changeColor()
{
 float r, g, b;

 for(int i = 0; i < 8; i++)
 {
  r = (float)rand() / ((float)RAND_MAX / 255.0f);
  g = (float)rand() / ((float)RAND_MAX / 255.0f);
  b = (float)rand() / ((float)RAND_MAX / 255.0f);
  cubeColors[i] = XMFLOAT3(r/255.0f, g/255.0f, b/255.0f);
 }


 VertexPositionColor cubeVertices[] =
        {
     {XMFLOAT3(-0.5f, -0.5f, -0.5f), cubeColors[0]},
            {XMFLOAT3(-0.5f, -0.5f,  0.5f), cubeColors[1]},
            {XMFLOAT3(-0.5f,  0.5f, -0.5f), cubeColors[2]},
            {XMFLOAT3(-0.5f,  0.5f,  0.5f), cubeColors[3]},
            {XMFLOAT3( 0.5f, -0.5f, -0.5f), cubeColors[4]},
```


(C) Copyright Nokia 2014. All rights reserved.

```
            {XMFLOAT3( 0.5f, -0.5f,  0.5f), cubeColors[5]},
            {XMFLOAT3( 0.5f,  0.5f, -0.5f), cubeColors[6]},
            {XMFLOAT3( 0.5f,  0.5f,  0.5f), cubeColors[7]},
        };

        D3D11_SUBRESOURCE_DATA vertexBufferData = {0};
        vertexBufferData.pSysMem = cubeVertices;
        vertexBufferData.SysMemPitch = 0;
        vertexBufferData.SysMemSlicePitch = 0;
        DX::ThrowIfFailed(
            m_d3dDevice->CreateBuffer(
                &CD3D11_BUFFER_DESC(sizeof(cubeVertices), D3D11_BIND_VERTEX_BUFFER),
                &vertexBufferData,
                &m_vertexBuffer
                )
            );
}
```

We also modify the Update() method to take account the rotation float when doing the model matrix transformation and to increase or decrease the rotation value depending on the rotationNegative boolean:

```
void CubeRenderer::Update(float timeTotal, float timeDelta)
{
    XMVECTOR eye = XMVectorSet(0.0f, 0.7f, 1.5f, 0.0f);
    XMVECTOR at = XMVectorSet(0.0f, -0.1f, 0.0f, 0.0f);
    XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

    m_constantBufferData.view = XMMatrixTranspose(XMMatrixLookAtRH(eye, at, up));
    m_constantBufferData.model =
XMMatrixTranspose(XMMatrixRotationY(XMConvertToRadians(rotation)));

    if(rotationNegative)
        rotation -= 1.0f;
    else
        rotation += 1.0f;
}
```

Finally, we need to make some additions to the Direct3DInterop's constructor. Here, format the three new private members and also start up the randomizing service.

```
CubeRenderer::CubeRenderer() :
    m_loadingComplete(false),
    m_indexCount(0)
{
 rotation = 0.0f;
 rotationNegative = false;

 srand((unsigned)time(0));

 cubeColors = new XMFLOAT3[8];
}
```

Also, remember to add an include to <time.h> to make the rand() method work.

In the Direct3DInterop class, we are going to add two methods: ChangeCubeRotation() and ChangeCubeColor(). Go ahead and

(C) Copyright Nokia 2014. All rights reserved.

add their declarations to Direct3DInterop.h:

```
public:
    ...
    void ChangeCubeRotation();
    void ChangeCubeColor();
```

In Direct3DInterop.cpp, we have the definitions for the methods. In the definitions, we only call the m_renderer member's changeRotation() and changeColor() methods which we created earlier:

```
void Direct3DInterop::ChangeCubeRotation()
{
 m_renderer->changeRotation();
}

void Direct3DInterop::ChangeCubeColor()
{
 m_renderer->changeColor();
}
```

Those are all the additions we need to do to the C++ project. **Important note:** for the newly created C++ methods to become available in the C# code (like in VS auto fill), we first need to build the Direct3DXamlApp2Comp C++- project. Do this before moving to the next step.

Adding C# Code to Call C++ Methods

Last thing we need to do is to call the Direct3DInterop class methods we created earlier in the C# code. We will implement event handlers for the two XAML buttons we created and call the methods in the handlers.

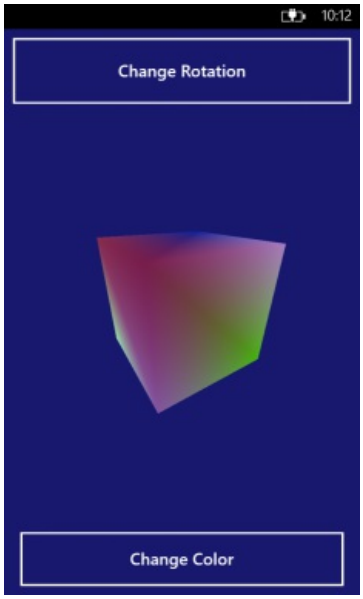
Open up the MainPage.xaml.cs and add the following event handling methods for the buttons. Inside the methods we call the corresponding Direct3DInterop methods.

```
 private void rotationButton_Click(object sender, RoutedEventArgs e)
{
     m_d3dInterop.ChangeCubeRotation();
}

private void colorButton_Click(object sender, RoutedEventArgs e)
{
    m_d3dInterop.ChangeCubeColor();
}
```

And that's all the C# code we need to add. The application is now ready to run. When the "Change Rotation" button is clicked, the cube should change its rotation direction and when the "Change Color" button is clicked, the cube color should change into a random one.

(C) Copyright Nokia 2014. All rights reserved.



(C) Copyright Nokia 2014. All rights reserved.