Creating a Lens application that uses HLSL effects for filters

This article covers how to create a Lens application that applies different filters to the photos. These filters are programmed in High Level Shading Language (HLSL) and are executed on the GPU to take advantage of the new DirectX functionality introduced in Windows Phone 8.

13 Jan 2013

Note: This article was a winner in the Windows Phone 8 Wiki Competition 2012Q4.

Introduction

Instagram is one of the most popular photo applications for iPhone (and for Android). It started the tradition of applying post-processing filters to pictures an artistic twist by simulating the effect of snapping the photos with an old Polaroid or Lomographic camera. These effects, although very computationally expensive, can be achieved with relative ease. Thanks to the new DirectX APIs available in Windows Phone 8, which allow us to execute the image processing in the GPU. In this tutorial, we will create an application that allows us to preview the camera input, snap a photo and apply an HLSL post-processing effect to it.

Creating the base project

Open a new instance of Visual Studio and create a project based on the **Windows Phone App** template. We will be using a standard C#/XAML application and modify it further to accustom our requirements.

Modifying the MainPage

Start by opening the **MainPage.xaml** file and deleting the Grid control named `LayoutRoot` and all its children elements. Now create a control of type `DrawingSurfaceBackgroundGrid` as the page's root and give it a name (we will be calling it `DrawingSurface`). This is required when your application is going to use full screen mode for advanced graphics rendering through DirectX, so you don't get any performance penalties derived from the XAML composition system. You can find more info about this control in the MSDN article Direct3D with XAML apps for Windows Phone 8.

Adding SharpDX references

Instead of using the default DirectX interoperatiblity though a separate C++ DLL, we are going to use the SharpDX library to make drawing calls from C# code. This library is a wrapper of the underlying DirectX functions that allow them to be used in any .NET language and currently supports Windows Desktop, Windows Metro and Windows Phone 8. Start by heading to the downloads section and get the package of your choice: "binary only" includes the libraries and "full package" has some sample code on how to perform common tasks with SharpDX.

Create a new directory inside your project's folder called **Lib**. Open it and create two child folders called **x86** and **ARM**. Decompress the package you downloaded and from the **Bin** folder, copy the contents of **Standard-wp8-x86** to your **x86** folder and **Standard-wp8-ARM** to **ARM**. In your project, right click on **References** and select **Add References...**, and in the new window click **Browse** and navigate to the **Lib\x86** folder to select the following assemblies:

- SharpDX.dll
- SharpDX.DXGI.dll
- SharpDX.Direct3D11.dll
- SharpDX.Toolkit.dll
- SharpDX.Toolkit.Game.dll
- ShaprDX.Toolkit.Graphics.dll

Since the assemblies aren't AnyCPU, and currently we only have the x86 ones referenced, we must edit our project manually so the compiler references the correct ones. Close Visual Studio and open your CSPROJ file in your favourite text editor and look for the items named `Reference`, like this one:

```
<Reference Include="SharpDX">
  <HintPath>Lib\x86\SharpDX.dll</HintPath>
</Reference>
```

You need to change the **x86** part of the path to **$(Platform)**, so it ends like this:

```
<Reference Include="SharpDX">
  <HintPath>Lib\$(Platform)\SharpDX.dll</HintPath>
</Reference>
```

What we have written is a MSBuild property that gets replaced with the current platform name when building the project, so the correct version of the assemblies is used. Repeat this step for all existing references to SharpDX. When finished, save the changes and reopen the solution in Visual Studio.

Note: If Visual Studio can't find the SharpDX references, go to the **Build > Configuration Manager...** menu and change **Active solution platform** to **x86**. Remember to change it again to **ARM** when deploying to a Windows Phone 8 device.

Application loop and basic drawing

Although we are creating a standard XAML navigation-based application, we are going to leverage some of the DirectX functionality to the framework provided by **SharpDX.Toolkit**. This is a collection of classes and utilities that mimics a subset of the XNA framework, and is provided as an extension to the core SharpDX libraries. If you have previous experience with XNA, you will find some of the code we are going to write very familiar.

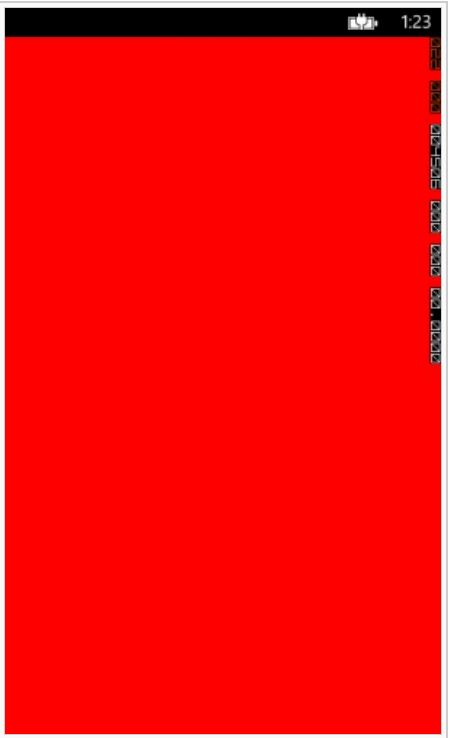
Start by creating a new class and naming it **MainLoop**. Make it inherit from **SharpDX.Toolkit.Game** and add a private field of type **SharpDX.Toolkit.GraphicsDeviceManager**. Now go to the constructor and initialize this field, passing `this` as the only parameter. At last, override the virtual function `Draw()` and add the line `GraphicsDevice.Clear(GraphicsDevice.BackBuffer, Color.Red);` to its body. This should be the result:

```
public class MainLoop : Game
{
    GraphicsDeviceManager deviceManager;

    public MainLoop()
    {
        this.IsFixedTimeStep = false;
        deviceManager = new GraphicsDeviceManager(this);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(GraphicsDevice.BackBuffer, Color.Red);

        base.Draw(gameTime);
    }
}
```

When created, the `GraphicsDeviceManager` will fetch the appropriate graphics adapter and initialize a valid graphics device that will allow the application to issue draw calls to the screen through the DirectX runtime. This, in turn, will allow us to tell the `GraphicsDevice` in the `Draw()` function to draw whatever we want; for now, we will clear the entire screen to red. Also, we set the `IsFixedTimeStep` property to `false`, which will tells the game engine to suppress frame dropping if the game is running too low. Doing so will prevent a bug in SharpDX for Windows Phone 8 from occouring, where old back buffers are drawn and flickering occours. Finally, to make this loop run independently while the application is open, instantiate it in the `MainPage` and call its `Run()` method with the `DrawingSurfaceBackgroundGrid` you created as its parameter:

```
public partial class MainPage : PhoneApplicationPage
{
    MainLoop loop;

    public MainPage()
    {
        InitializeComponent();

        loop = new MainLoop();
        loop.Run(this.DrawingSurface);
    }
}
```

Run your application in the emulator or a device and it should display as follows:



Initial version of the application using DirectX drawing

Displaying the camera feed onscreen

Now that we have a working DirectX context, we are going to access the camera API to obtain its preview image and drawing it onscreen.

Creating and drawing a DirectX texture

Go to **MainLoop.cs** and add a public member variable of type **SharpDX.Toolkit.Graphics.Texture2D** (be careful not to use the one in the **SharpDX.Direct3D11** namespace!), and a private one of type **SharpDX.Toolkit.Graphics.SpriteBatch**. The texture will be drawn onscreen every frame via the `SpriteBatch` and will hold the camera preview data in the future. Now, create an override for the function `Initialize` of `MainLoop`; this function gets called when the DirectX device and adapter have been successfully created, and will initialize a blank version of our texture and the much needed `SpriteBatch`.

```
protected override void Initialize()
{
    CreateTexture(640, 480);
    spriteBatch = new SpriteBatch(GraphicsDevice);

    base.Initialize();
}
```

The function `CreateTexture` is just a shortcut for the creation and initialization of the texture, to make the code cleaner. Here is the code:

```
private void CreateTexture(int textureWidth, int textureHeight)
{
    previewTexture = Texture2D.New(GraphicsDevice, textureWidth, textureHeight, PixelFormat.B8G8R8A8.UNorm);

    Color[] data = new Color[textureWidth * textureHeight];
    for (int i = 0; i < textureWidth * textureHeight; i++)
    {
        data[i] = Color.White;
    }

    previewTexture.SetData<Color>(data);
}
```

We just create it by calling `Texture2D.New` and passing the appropriate arguments. Be careful that the `PixelFormat` must be `B8G8R8A8.UNorm` since that's the order the camera will return the colour bytes in, and we will be saving an extra swizzling by declaring it this way. Lastly, the function will initialize an array of `Color` objects to `White` and feed it as the initial data to the texture.

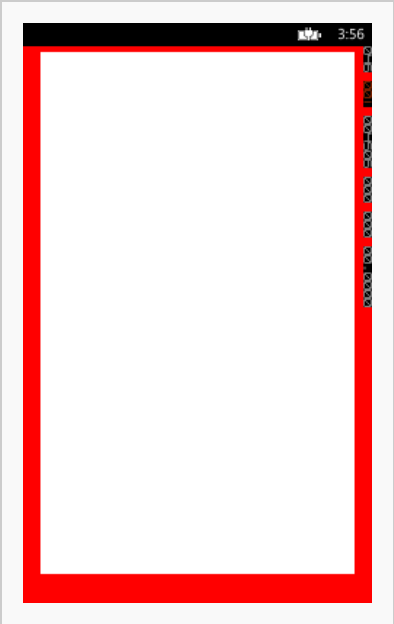
Note: As of version 2.4.1, SharpDX doesn't support backbuffer orientations other than portrait. We are going to manually rotate and scale the texture when drawing it so it appears in landscape mode.

Now we have to modify the `Draw` function so the `SpriteBatch` previously created draws the texture in fullscreen mode:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(GraphicsDevice.BackBuffer, Color.Red);

    float backBufferXCenter = GraphicsDevice.BackBuffer.Width / 2;
    float backBufferYCenter = GraphicsDevice.BackBuffer.Height / 2;

    float textureXCenter = previewTexture.Width / 2;
    float textureYCenter = previewTexture.Height / 2;

    float yScale = (float)GraphicsDevice.BackBuffer.Width / (float)previewTexture.Height;
    float xScale = (float)GraphicsDevice.BackBuffer.Height / (float)previewTexture.Width;

    spriteBatch.Begin();
    spriteBatch.Draw(previewTexture, new Vector2(backBufferXCenter, backBufferYCenter), null, Color.White, (float)Math.PI / 2.0f,
        new Vector2(textureXCenter, textureYCenter), new Vector2(xScale, yScale), SpriteEffects.None, 0.0f);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

We calculate the center of both the backbuffer and out texture, so we can properly align the drawing origin to the center of the screen. Next, we obtain the scale in both axis in which the texture must be multiplied so it fits in the entire screen without overflowing. And at last, we apply a rotation of Pi/2 radians (90 degrees) to give it the correct landscape orientation. When executed, the white texture should cover all the red background:

DirectX texture covering the entire screen.



Texture scaled to show how it covers the background.

Obtaining camera preview and updating the texture

First of all, go open your **WMAppManifest.xml** file and add the capability **ID_CAP_ISV_CAMERA** and the requirement **ID_REQ_REARCAMERA** using the visual editor included in VS2012. This allows you to get the privileges to access the camera hardware and restricts your app to devices that have at least a rear camera, respectively.

Instead of using the old **Microsoft.Devices.PhotoCamera** class, we are going to take advantage of the functionality added in **Windows.Phone.Media.Capture.PhotoCaptureDevice**. One of the more crippling limitations of the old `PhotoCamera` was that you had to launch a separate thread and call `GetPreviewBufferArgb32()` whenever you wanted to update your camera preview. With `PhotoCaptureDevice` we can subscribe to the `PreviewFrameAvailable` event and get notified automatically when such data is available.

Start by adding a new private member variable of type `PhotoCaptureDevice` to your `MainLoop` class. We are going to instantiate it in our `Initialize()` function, but we will need to obtain first the camera's supported preview resolution for creating our texture with the appropriate size:

```
protected override async void Initialize()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    Size previewSize = PhotoCaptureDevice.GetAvailablePreviewResolutions(CameraSensorLocation.Back)[0];
    Size captureSize = PhotoCaptureDevice.GetAvailableCaptureResolutions(CameraSensorLocation.Back)[0];

    CreateTexture((int)previewSize.Width, (int)previewSize.Height);
    photoDevice = await PhotoCaptureDevice.OpenAsync(CameraSensorLocation.Back, captureSize);
    photoDevice.PreviewFrameAvailable += photoDevice_PreviewFrameAvailable;

    base.Initialize();
}
```

The function has been marked as `async` so we can `await` the call that initializes the camera. We obtain the preview size by calling `PhotoCaptureDevice.GetAvailablePreviewResolutions` with `CameraSensorLocation.Back` as its parameter to query the hardware to return all supported preview resolutions by that camera. We index the first element of the array to get the smallest preview size. Note that the call to `CreateTexture` has been appropriately changed to use the new size instead of the old, hardcoded parameters. And at last, `PhotoCaptureDevice.OpenAsync` asynchronously gets the camera device with the properties we want (back facing and specified capture resolution). To make sure we get an updated buffer of the camera's viewpoint, we subscribe to the `PreviewFrameAvailable` event. Since the event is raised in a seperate thread, we shall not access the photo device directly to get the data. However, the thread which runs the game loop can be easily signalled to process our data:

```
void photoDevice_PreviewFrameAvailable(ICameraCaptureDevice sender, object args)
{
    newPreviewFrameAvailable = true;
}
```

We can now retreive the new frame in the Draw() method, which is called by the thread which runs our game loop. This operation is thread safe, since our PhotoCaptureDevice was created by this thread.

```
protected override void Draw(GameTime gameTime)
{
    if(newPreviewFrameAvailable)
    {
        int[] data = new int[previewTexture.Width * previewTexture.Height];
        sender.GetPreviewBufferArgb(data);
        previewTexture.SetData<int>(data);
        newPreviewFrameAvailable = false;
    }

    GraphicsDevice.Clear(GraphicsDevice.BackBuffer, Color.Red);
```

```
    // ... Snip ...
}
```

The code above obtains the ARGB data into an array of `int` values and feeds it to the texture. Since the endianness for DirectX is switched, we don't have to perform any byte swizzling to make the individual colour components fit the texture's format. Run the application and you should see the camera's input drawn on the screen:

Camera preview drawn as a DirectX texture.

Warning: Calling `GetPreviewBufferArgb` when running the application in the emulator returns a pure white screen instead of the debug screen with the moving box. The sample code contains functionality to load and display a predefined photo so you can test it without needing a device, although it isn't discussed in this article.

Applying a shader to the camera preview

Now that we are capturing the camera's current view and drawing it, we can proceed with the main purpose of this article: applying an HLSL effect to it.

Creating and compiling the shader

SharpDX.Toolkit fully supports FX effect files, although you can only compile them at runtime in Desktop platforms. To compile them to a binary format we are going to use the **tkfxc.exe** tool, which you can find inside of the **Bin\Win8Desktop-net40** folder of the SharpDX binaries you downloaded. Our application will then be able to load this binary blob and use it for drawing.

We are going to create a simple color inversion effect, so create a new file called **Inverted.fx** and write the following HLSL code:

Warning: It is **critical** that you keep the sampler and input variable's names the same as here, as well as the function's parameter order; the default `SpriteBatch` shader of SharpDX.Toolkit (based in XNA's one) has these values hard-coded, and altering any of these names can lead to the shader compiling correctly but failing silently when drawing. You can still change the name and body of the vertex and pixel shader functions, though.

```
Texture2D<float4> Texture : register(t0);
sampler TextureSampler : register(s0);

cbuffer ProjectionMatrix : register(b1)
{
    row_major float4x4 MatrixTransform : packoffset(c0);
};

void InvertedVS(inout float4 color    : COLOR0,
                      inout float2 texCoord : TEXCOORD0,
                      inout float4 position : SV_Position)
{
    position = mul(position, MatrixTransform);
}

float4 InvertedPS(float4 color : COLOR0,
                       float2 texCoord : TEXCOORD0) : SV_Target0
{
 float4 colorSampler = Texture.Sample(TextureSampler, texCoord);
 return float4(1.0 - colorSampler.x, 1.0 - colorSampler.y, 1.0 - colorSampler.z, 1) * color;
}

technique Inverted
{
    pass
    {
        EffectName = "InvertedEffect";

        VertexShader = compile vs_2_0 InvertedVS();
        PixelShader  = compile ps_2_0 InvertedPS();
    }
}
```

If you are versed in HLSL you will see that this shader is pretty straightforward; if not, we are multiplying the vertex position with the correct transformation matrix in the vertex shader and doing a texture fetch and returning the inverted colour value in the pixel shader. **tkfxc.exe** will automatically compile our shader model 2.0 effects to 4_0_level_9_1 compatibility mode. The maximum shader model value supported in Windows Phone 8 is 3.0, which gets translated to 4_0_level_9_3.

To compile it, make sure the file is in the same directory as your **tkfxc.exe** binary (or you have it correctly added to the PATH) and execute the following command in a Command Prompt window:

```
tkfxc.exe /FoInverted.tkfxo Inverted.fx
```

This will compile the shader in a binary form and output to the file **Inverted.tkfxo** (by default it's **output.tkfxo**). If something bad happened you will get a red text output, so check what went wrong and fix the shader until it compiles.

Tip: You can output the binary code to a C# source code file containing an array of `byte` objects. To do this, specify the parameter **/FcInverted.cs** instead of **/FoInverted.tkfxo**.

Loading and applying the shader

Go back to the **Solution Explorer** window of Visual Studio and create a new folder called **Content** inside your project. Copy and paste the .tkfxo file resulting from the previous step inside this folder, right click it, select **Properties** and change the value of **Build Action** to **Content**. This will add the binary shader to our application's package and we will be able to load it at runtime.

Add a new private member variable of type **SharpDX.Toolkit.Graphics.Effect** to your `MainLoop` class and add the following line to its `Initialize()` function:

```
Content.RootDirectory = "Content";
```

This is exactly the same as in XNA; we are telling the default **ContentManager** to use that directory as its root directory. Now, create a new override for `void LoadContent` and load the effect with the following code:

```
protected override void LoadContent()
{
    inversionEffect = Content.Load<Effect>("Inverted.tkfxo");

    base.LoadContent();
}
```

Make sure that the value you pass as a parameter is the same path, minus the Content folder, where you have your tkfxo file. At last, to apply the effect when drawing the texture, locate your `spriteBatch.Begin` call and change it with this:

```
spriteBatch.Begin(SpriteSortMode.Deferred, inversionEffect);
```

We are passing the default `SpriteSortMode`, but we are telling SharpDX to ignore the default shader and apply our custom one when drawing this batch. Now look at the result:

Camera preview with our colour inversion effect.

Taking a photo and processing it

We have built the infrastructure to preview our camera input; now we can proceed to capture a shot and save it to our camera roll.

Capturing the photo

Go back to your **MainPage.xaml** file (preferably in Blend) and add a function callback to the **Tap** event of the `DrawingSurfaceBackground`. Mark it as `async` and write a call to the yet unimplemented function `loop.CapturePhoto` inside:

```
private async void DrawingSurface_Tap(object sender, System.Windows.Input.GestureEventArgs e)
{
    await loop.CapturePhoto();
}
```

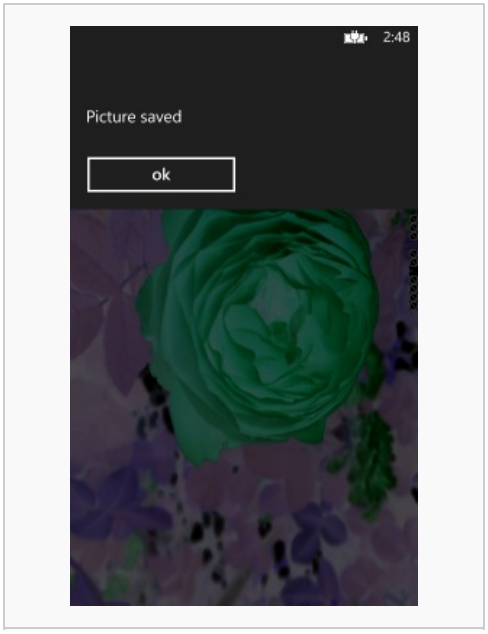
We are going to take the photo in a blocking call since we need to do some critical processing inside this function, and no other DirectX or UI process should interrupt it. Back to **MainLoop**, the function `CapturePhoto` must be declared as `async Task` so it can be awaited without having to return any data type. Inside it, we are going to proceed with the following steps:

- Create a `CameraCaptureSequence` object named `singlePhoto` by calling `photoDevice.CreateCaptureSequence` and specifying that we only want to capture one frame.
- Declare a temporal `MemoryStream` that will be assigned to the `CaptureStream` property of the first (and only) `singlePhoto.Frames`; this way, we get a buffer filled with our photo data, specified in format YCbCr.
- Call `photoDevice.PrepareCaptureSequenceAsync` to prepare the camera for taking a shot, and then `singlePhoto.StartCaptureAsync` to capture it; make sure to `await` both calls.
- Rewind the temporal `MemoryStream` buffer so we don't get any exceptions when saving it.
- Instantiate a `Microsoft.Xna.Framework.Media.MediaLibrary` and call its `SavePictureToCameraRoll` function to save our photo data to the Camera Roll album.

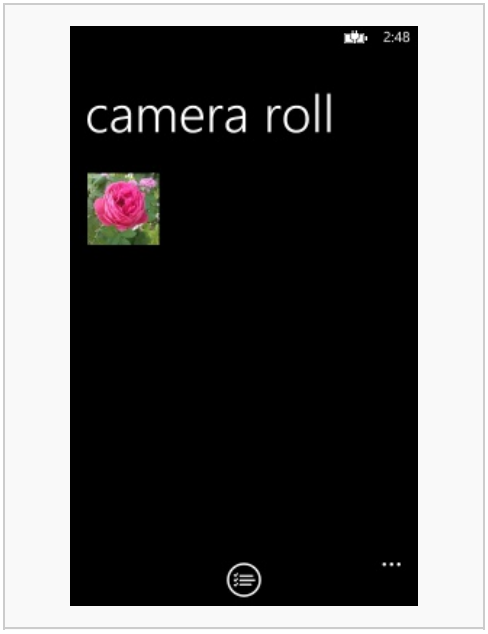
Note: Before trying to access `Microsoft.Xna.Framework.Media.MediaLibrary`, make sure you have added the `ID_CAP_MEDIALIB_PHOTO` capability to your **WMAppManifest.xml** file.

```
public async Task CapturePhoto()
{
    MemoryStream frameCaptureStream = new MemoryStream();
    CameraCaptureSequence singlePhoto = photoDevice.CreateCaptureSequence(1);

    singlePhoto.Frames[0].CaptureStream = frameCaptureStream.AsOutputStream();
    await photoDevice.PrepareCaptureSequenceAsync(singlePhoto);
    await singlePhoto.StartCaptureAsync();

    frameCaptureStream.Seek(0, SeekOrigin.Begin);

    Microsoft.Xna.Framework.Media.MediaLibrary library = new Microsoft.Xna.Framework.Media.MediaLibrary();
    Microsoft.Xna.Framework.Media.Picture picture = library.SavePictureToCameraRoll("HSLCamera" + DateTime.Now.Ticks + ".jpg", frame

    System.Windows.MessageBox.Show("Picture saved");
}
```

This should be the result:



Saving the picture to the photo library.



Camera roll with the picture saved.

Applying the shader to the final photo before saving

We have just captured our first photo, but it is doesn't have any kind of processing! We have to add some extra DirectX functionality to get the desired result:

- Create a new `Texture2D` member variable, with the same size as the camera's capture resolution and with the format `PixelFormat.R8G8B8A8.UNorm`.
- Create a new `RenderTarget2D` member variable, with the same size as the camera's capture resolution and with the format `PixelFormat.B8G8R8A8.UNorm`.
- Go back to the `CapturePhoto()` function and add the following improvements:
  - Create a `WriteableBitmap` of the same size as the camera's capture resolution; we will use it as an intermediate decoding buffer because the camera's data is supplied in YCbCr format and DirectX doesn't have any method to handle this pixel format.
  - Call `SetSource` in the `WriteableBitmap` passing the `MemoryStream` we created to handle the capture stream as its parameter.
  - Call the `SetData` function of the full sized texture passing `decodeBitmap.Pixels` as the parameter; this will feed raw RGBA data to our DirectX texture holding the high-resolution photo.
  - Set the active render target to the one we created.
  - Set the viewport to one matching the size of the render target.
  - Do the same `SpriteBatch` calls as when drawing the preview, except this time we draw the full sized texture.
  - Restore the back buffer as the active render target.
  - Call `GetData` in our render target passing as its parameter `decodeBitmap.Pixels`. This will feed back to the `WriteableBitmap` our processed pixel data.
  - Create a temporal `MemoryStream` and use it to hold the output of `decodeBitmap.SaveJpeg`. We **need** to manually encode our picture to JPEG before saving it because `SavePictureToCameraRoll` only accepts data in YCbCr or JPEG formats.
  - Save it as we did previously.

```
public async Task CapturePhoto()
{
    MemoryStream frameCaptureStream = new MemoryStream();
    WriteableBitmap decodeBitmap = new WriteableBitmap((int)photoDevice.CaptureResolution.Width, (int)photoDevice.CaptureResolution.

    CameraCaptureSequence singlePhoto = photoDevice.CreateCaptureSequence(1);
    singlePhoto.Frames[0].CaptureStream = frameCaptureStream.AsOutputStream();
    await photoDevice.PrepareCaptureSequenceAsync(singlePhoto);
    await singlePhoto.StartCaptureAsync();

    frameCaptureStream.Seek(0, SeekOrigin.Begin);
    decodeBitmap.SetSource(frameCaptureStream);
    captureTexture.SetData<int>(decodeBitmap.Pixels);

    GraphicsDevice.SetRenderTargets(renderTarget);
    GraphicsDevice.SetViewports(new SharpDX.Direct3D11.Viewport(0, 0, (int)photoDevice.CaptureResolution.Width, (int)photoDevice.Cap
    spriteBatch.Begin(SpriteSortMode.Deferred, inversionEffect);
```

```
    spriteBatch.Draw(captureTexture, Vector2.Zero, Color.White);
    spriteBatch.End();
    GraphicsDevice.SetRenderTargets(GraphicsDevice.BackBuffer);

    renderTarget.GetData<int>(decodeBitmap.Pixels);

    MemoryStream mem = new MemoryStream();
    decodeBitmap.SaveJpeg(mem, (int)photoDevice.CaptureResolution.Width, (int)photoDevice.CaptureResolution.Height, 0, 100);
    mem.Seek(0, SeekOrigin.Begin);

    Microsoft.Xna.Framework.Media.MediaLibrary library = new Microsoft.Xna.Framework.Media.MediaLibrary();
    Microsoft.Xna.Framework.Media.Picture picture = library.SavePictureToCameraRoll("HSLCamera" + DateTime.Now.Ticks + ".jpg", mem);

    System.Windows.MessageBox.Show("Picture saved");
}
```

If you did it correctly, your photos will be saved now with the correct shader effect applied:

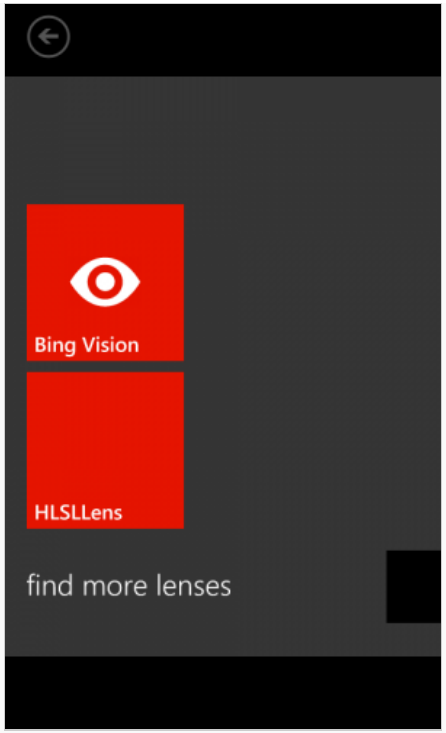
Correctly processed photo saved in the Camera Roll.

Adding Lens functionality

At last, to make sure you can select your app as a Lens when taking a photo with the default Camera application, manually edit the **WMAppManifest.xml** file and paste this code after the **Tokens** declaration:

```
<Extensions>
  <Extension ExtensionName="Camera_Capture_App" ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5631}" TaskID="_default" />
</Extensions>
```

Congratulations, now you have a fully functional lens application that can be extended in any way you want. You can download the complete source code here: Media:HLSLLens.zip.



Lens selection screen.

Remarks

This article shows just the bare minimum to make an application of this type; more flexibility and error handling should be implemented if you plan on extending it. Here is a list of features that would make it more robust:

- Check if the maximum camera resolution is bigger than the maximum DirectX texture size (4096x4096 pixels), and resize it accordingly before processing.
- Add more HLSL effects and provide a menu to select the active one.
- Use the app as both a Lens and a photo editor to apply effects to already taken pictures.
- Add more effects than plain colour processing: distortion, blending with a pattern...
- Add in-app image navigation.

Reference

- SharpDX documentation.
- Lens extesibility for Windows Phone.
- Advanced photo capture for Windows Phone 8.