Real-time camera viewfinder filters in Native code

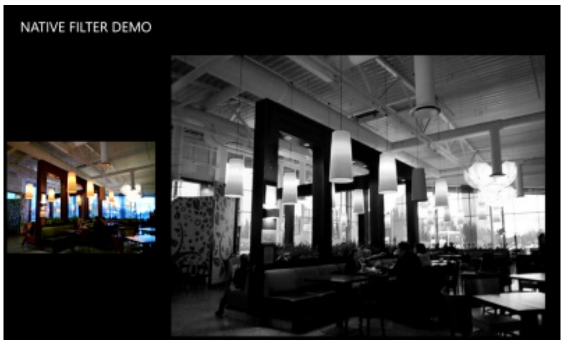
This article explains how to create real-time camera filters for Windows Phone 8, using native code (C++).

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

Introduction

One of the big new features of Windows Phone 8 SDK is the support for C and C++, also known as native code support. In this article, we will have a brief look at how one can exploit that support to create real-time filters for the camera. For simplicity's sake, the example will implement a simple gray filter, that will convert camera input on the fly and show the result onscreen.

The demo application will look like this:



.

Why native filters?

Microsoft has published a very similar example, where they do live conversion of the camera viewfinder images to grayscale. The example was written for Windows Phone 7 (you can download it here), but it also works well in WP8. This works well, but the gray filter is quite simple; more complicated filters will require more computation, and the CPU is quickly maxed out when we try to process the camera input at several frames per second. The speed gain by going closer to the metal may be needed for more complex algorithms. Also, you might already have your own image filters written in C and/or C++ for other platforms, that you can reuse without converting them to C#. Finally, as we will see in other wiki entries, the native side opens further optimization possibilities, like using DirectX or the ARM Neon instruction set.

Setting up the viewfinder

Our UI will be XAML based. The UI will control everything, while the C++ side is rather dumb, simply executing the filtering when asked to. Let's first create the projects for both the XAML and the C++ components:

- Start by creating a new project, of type ***Windows Phone App***. You will find the template under the **Visual C#/Windows Phone** category. That's where our UI will be coded.
- Add a new project to your solution, of type ***Windows Phone Runtime Component***. That template is under the **Visual C++/Windows Phone** category. That will become our image filter.

We then add a live camera stream to our UI. That is easily done by:

- In your XAML, define a rectangle that will be painted using a `VideoBrush`:

```
<Grid x:Name="LayoutRoot" Background="Transparent">
        <Rectangle Width="640" Height="480" Canvas.ZIndex="1">
            <Rectangle.Fill>
                <VideoBrush x:Name="viewfinderBrush" />
            </Rectangle.Fill>
        </Rectangle>
  </Grid>
```

- In the page loaded event, create a PhotoCaptureDevice, and set it as the source of the `VideoBrush`:

```
Windows.Foundation.Size resolution = new Windows.Foundation.Size(640, 480);
m_camera = await PhotoCaptureDevice.OpenAsync(CameraSensorLocation.Back, resolution);
ViewfinderBrush.SetSource(m_camera);
```

By now, with these 10 lines of codes, you should have an application with a functional camera! Note that in the last step, we use the Windows PRT class **Windows.Phone.Media.Capture.PhotoCaptureDevice** which is new to Windows Phone 8. In WP7, one would have to use the Silverlight/.NET class **Microsoft.Devices.PhotoCamera**. Because it's a Win PRT class, it can be accessed by both managed and native code; we will soon take advantage of that possibility.

Displaying the filtered frames

The `VideoBrush` is easy to use but it doesn't offer us a way to get in-between the camera and the brush to apply our filter. We need to find another way to display the modified frames coming from the camera. Several strategies are possible, but the strategy chosen must have a reasonable camera lag (the time between the photons enters the camera until the scene is displayed on the screen) as well as a decent frame rate. Let's have a quick look at 3 possible strategies:

1. Using an `Image` control, with `WriteableBitmap` as a source.
2. Using a `MediaElement` control, with a custom `MediaStreamSource` as a source.
3. Using a DirectX texture.

Using an Image control

This is the strategy used in the How to: Work with Grayscale in a Camera Application for Windows Phone MSDN sample. The UI control defined in XAML is:

```
<Image x:Name="MyCameraViewfiner"  Width="640" Height="480">
```

The source of the `Image` is set to a `WriteableBitmap`. When the frames from the camera have been filtered, the resulting array of pixels is copied into to that `WriteableBitmap`.

```
Deployment.Current.Dispatcher.BeginInvoke(delegate()  // Switch to UI thread
{
    // Copy to WriteableBitmap.
    ARGBPx.CopyTo(wb.Pixels, 0);
     wb.Invalidate();
});
```

The problem with this is the switch to UI thread (`Deployment.Current.Dispatcher.BeginInvoke`). The switch is required so that the application does not update the bitmap while the UI thread is doing something with it. Switching between threads is quite slow, and the lag associated with that solution was too high for the application.

Using a MediaElement

The MediaElement is 'the' UI control that is used for media playback, videos or audio, streamed media or from a local file. It takes care of all the buffering, audio sync and displaying logic and it's highly optimized. By defining a custom `MediaStreamSource`, we can feed the filtered camera frames to the `MediaElement` that will take care of displaying them properly.

```
<MediaElement x:Name="MyCameraMediaElement"
                             IsHitTestVisible="False"
                             Margin="4" Width="640" Height="480" />
```

Defining your own `MediaStreamSource` requires a little bit of work, but it's all worth it. The lag with this solution is small, and the frame rate we can achieve is pretty good. This is the solution we will take into use.

Using a DirectX texture

When it comes to camera lag and performance, this is probably the best solution. However DirectX has a fairly steep learning curve, and to keep things simple we will not use it. Look over this article if you're interested in using DirectX.

Our own MediaStreamSource

To feed video frames to the `MediaElement` we need to define our custom `MediaStreamSource`. Pete Brown walks us through how this should be done in his blog. Check it out for more details, here we will only go through the big lines.

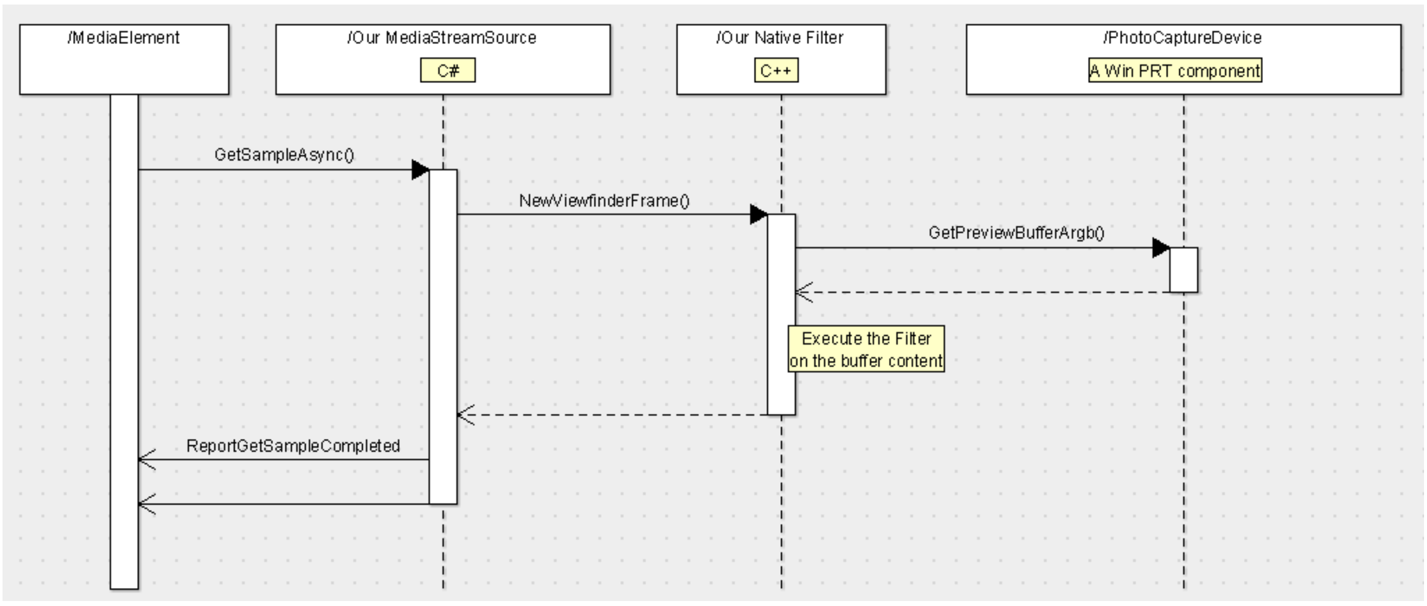
Our `MediaStreamSource` will define a single stream of type video (RGBA). Since our real source for video data is the camera, we have a source of infinite length and that can't be seeked. Our initialization will look like this:

```
mediaStreamAttributes[MediaStreamAttributeKeys.VideoFourCC] = "RGBA";
mediaStreamAttributes[MediaStreamAttributeKeys.Width] = _frameWidth.ToString();
mediaStreamAttributes[MediaStreamAttributeKeys.Height] = _frameHeight.ToString();
mediaStreamDescriptions.Add(_videoStreamDescription);

 // a zero timespan is an infinite video
 mediaSourceAttributes[MediaSourceAttributesKeys.Duration] =
TimeSpan.FromSeconds(0).Ticks.ToString(CultureInfo.InvariantCulture);
// Can't seek.
mediaSourceAttributes[MediaSourceAttributesKeys.CanSeek] = false.ToString();
```

Whenever the `MediaElement` decides it needs a new video frame, it will call the `GetSampleAsync()` method from our `MediaStreamSource`.

We will implement the following sequence diagram :



To get the data of the frames coming from the camera, we will have to call to the platform functionality **Windows::Phone::Media::Capture::ICameraCaptureDevice::GetPreviewBufferArgb**. Let's look at its definition:

```
void GetPreviewBufferArgb(Platform::WriteOnlyArray<int, 1U>^ pixels)
```

That method fills an array of our choice with the camera data. That's a copy operation we can't avoid. The buffer is of type `Platform::WriteOnlyArray`, which according to the MSDN documentation, is to be used when the caller passes an array for the method to fill. We will also use that buffer type to communicate between our managed component and our native component.

The public interface of our native component will be:

```
    public ref class WindowsPhoneRuntimeComponent sealed
    {
    public:
        WindowsPhoneRuntimeComponent();
         void Initialize(Windows::Phone::Media::Capture::PhotoCaptureDevice^
captureDevice);
         void NewViewfinderFrame( Platform::WriteOnlyArray<int,1U>^ inputBuffer,
             Platform::WriteOnlyArray<uint8,1U>^ outputBuffer);
    ...
     };
```

On the managed side, we allocate the `inputBuffer` and the `outputBuffer` when the application is initialized, and reuse those buffers for each frame:

```
_cameraData = new int[_frameWidth * _frameHeight];
_frameBufferSize = _frameWidth * _frameHeight * _framePixelSize;
_cameraFilteredData = new byte[_frameBufferSize]
```

Note that for this example, my data buffer is in RGBA format. It's a format easy to handle, and the most familiar for most of us. However, it is not very efficient in terms of size and image manipulation performance. Using YUV format/color space would make more sense.
That covers the big lines of the data handling of the application. I skipped the not-so-interesting code, get the source code of the full project from the link in top right corner of this page. The last thing to do is the filtering itself.

The filtering in C++

For the filtering, I took the code from Nils Pipenbrinck's excellent blog entry on Neon optimization.

```
void WindowsPhoneRuntimeComponent::ConvertToGrayOriginal(
Platform::WriteOnlyArray<int,1U>^ frameData)
{
 uint8 * src = (uint8 *) frameData->Data;
 uint8 * dest = (uint8 *) frameData->Data;
 int n = frameData->Length;
 int i;
 for (i=0; i<n; i++)
 {
  int r = *src++; // load red
  int g = *src++; // load green
  int b = *src++; // load blue
  src++; //Alpha

  // build weighted average:
  int y = (r*77)+(g*151)+(b*28);

  // undo the scale by 256 and write to memory:

  *dest++ = (y>>8);
  *dest++ = (y>>8);
  *dest++ = (y>>8);
  dest++;

 }
}
```

If you look in the project code, you will find the same filter, but optimized with the ARM Neon instruction set. I'll let you pick the one you prefer.

Wrapping it up

Hopefully this short article will help you getting started writing native filters with Windows Phone 8. Remember to always keep the performance in mind, these types of applications are very CPU intensive. Using C++ for your filters is one way to improve the performance. If the performance gain is not enough for your use cases, you might have a look at hooking your camera directly into DirectX, or optimizing your filter with Neon instructions.

Source code

The source code is maintained in Nokia Project. Look for the zip file at the bottom of the page.