Real-time camera effects on Windows Phone 7 and 8

This article explains how to efficiently read a live stream from the camera on Windows Phone 7 and 8 devices and how to apply an effect filter on this stream.

25 Aug 2013

Winner: This article was a winner in the Nokia Imaging Wiki Competition 2013Q3.

Introduction

Most of the photo apps on the market are pretty simple. They allow users to take a photo, then apply some interesting vintage filter and save the photo. But only a few applications have a true live viewfinder with a real-time applied filter. Would it be nice to see how the photo will look like, even before you press the shutter button? What about some other live effects like a pixelated 8-bit style screen?

We will show you in this article how to make an app (in C# language) that can run on both Windows Phone 7 and Windows Phone 8 devices. Our goal will be to design a sufficiently fast implementation that can run flawlessly even on the oldest and slowest devices. We will go through several ideas and implementations and we will compare them.

The main ideas

- We will use a **VideoBrush** component at first. We will demonstrate its limitations.
- We will read ARGB values directly from the camera, and see how this values can be efficiently packed and unpacked.
- We will discover a faster solution (reading values from the YCbCr buffer).
- We will discuss how to make this processing even better (to work with a smaller resolution).
- We will see an idea how to make an interesting 8-bit style photo app based on these methods.

We will be demonstrating these methods on the Windows Phone 7 (WP7) project. In the last chapter we will cover the differences between WP7 and Windows Phone 8 (WP8):

- We will see how to use this approach on WP8 (thanks to MonoGame Framework)
- We will compare all methods and measure the rendering time

Nokia Imaging SDK comparison

There is also an alternative approach for how to read a real-time stream from the camera. The Nokia Imaging SDK provides a lot of useful methods for manipulating images and applying effects and filters. It is an efficient, easy to use library. You can use a couple of prepared filters to apply them on the live stream, as described in Real-time Filter Demo. You can also combine these filters together to create some new interesting effects: Combining imaging filters to create new real-time camera effects

Creating a completely new filter is a harder operation. If you want to manipulate directly the RGB data of every pixel; if you want to work with smaller resolution, or if you want to compose the final effect from smaller images (8-bit style effect), this article can help you a lot.

Nokia Imaging SDK is also available only for WP8, this solution runs smoothly on both WP7 and WP8 devices.

Demonstration of the effects

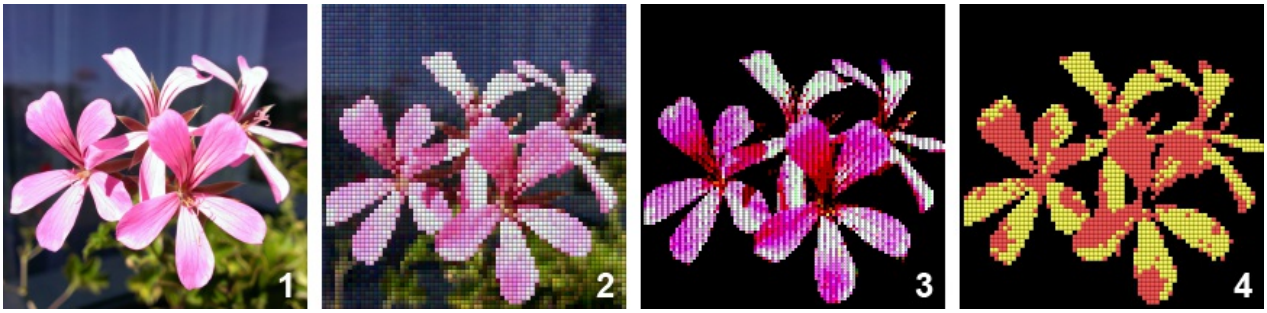


Image 1: Live stream from the camera. **Image 2**: 8-bit effect (a picture composed of smaller images). **Image 3 and 4:** 8-bit effect with a limited color palette, color threshold and some dithering algorithm applied (not more discussed in this article).

VideoBrush component

The simplest solution for showing a direct stream from the camera is the `VideoBrush` API. The code is the same on WP7 and WP8. You can add this code

into your XAML page:

```
<Rectangle Width="320" Height="320">
    <Rectangle.Fill>
        <VideoBrush x:Name="viewfinderBrush" />
    </Rectangle.Fill>
</Rectangle>
```

In the code-behind you can initialize the camera object, using PhotoCamera class and set the stream to the VideoBrush:

```
PhotoCamera camera;

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    // Initialize a camera object, set stream to the VideoBrush
    camera = new PhotoCamera();
    viewfinderBrush.SetSource(camera);
}

protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
    camera.Dispose();
}
```

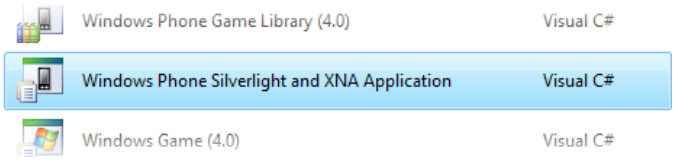
Do not forget to add an **ID_CAP_ISV_CAMERA** capability. This is a very simple solution. The stream from the camera is quite distorted and wrongly rotated, but you can fix this by setting a RelativeTransform on the Rectangle. But what is a real pity, you can't modify this stream in any way. You can't apply any filter or effect on the VideoBrush object, you don't have an access to the raw data of the pixels.

Reading from the buffer

We will use a different approach. As you can discover, there are some GetPreviewBuffer methods on the PhotoCamera object. This is the way how we can get the raw data from the camera. But we would like also to display this data somewhere, in some fast and efficient way. WritableBitmap component from the Silverlight is very slow, writing data pixel by pixel using this component can take a hundreds of milliseconds. We will need to find another solution.

We will create a new combined Silverlight/XNA project for Windows Phone 7 (we will talk about WP8 later). We would like to combine the ease of XAML and Silverlight with the power of the XNA in one solution. We will use a very similar approach to one described in the article Use Camera with XNA.

You need to have installed a Windows Phone SDK 7.1. Open your Visual Studio 2010 (Express or the Professional/Ultimate) and create a new Windows Phone Silverlight and XNA Application project.



To be able to read from the camera we need to place a fake Canvas to our XAML code. Add to **GamePage.xaml** page this line of code:

```
<Canvas x:Name="cameraDisplayFake" Width="0" Height="0"></Canvas>
```

We will create a helper object now. Let's create a new class **CamReader.cs** and place into it these declarations:

```
PhotoCamera camera;
int[] previewBuffer, outputBuffer;
Point previewSize, outputSize;
```

As you can see, there is some preview buffer and the output buffer. The preview buffer will contain the raw values from the camera and will be usually 640x480 pixels big. Applying effects on this large resolution would be very resource intensive and slow. We do not need such a big resolution for the live preview. Our output buffer will be smaller, for example only 240x240 px. All computations and filter calculations will be executed on this smaller resolution. We can scale up this image later, it will be a little blurred, but that does not matter much for us.

This is how the CamReader.StartCamera method can look like:

```
public void StartCamera(Dispatcher dispatcher, Canvas fakeCamCanvas, Point outputSize)
{
    this.outputSize = outputSize;

    if (camera != null)
        StopCamera(fakeCamCanvas);

    // Create a PhotoCamera instance
    if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
        camera = new PhotoCamera(CameraType.Primary);
    else if (PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing))
        camera = new PhotoCamera(CameraType.FrontFacing);
    else
    {
        System.Windows.MessageBox.Show("Cannot find a camera on this device");
        return;
    }

    // Wait camera initialization before create the buffer image
    camera.Initialized += (a, b) =>
    {
        // Move to UI thread
        dispatcher.BeginInvoke(() =>
        {
            outputBuffer = new int[outputSize.X * outputSize.Y];
            previewSize = new Point((int)camera.PreviewResolution.Width, (int)camera.PreviewResolution.Height);
```

```
            previewBuffer = new int[previewSize.X * previewSize.Y];
        }
        );
    };

    // Initialize camera
    // - we need to initialize VideoBrush to given Canvas, otherwise the stream wouldn't be loaded
    var brush = new VideoBrush();
    brush.SetSource(camera);
    fakeCamCanvas.Background = brush;
}
```

Reading ARGB data

Now we can design a method for reading ARGB data from the camera:

```
public int[] GetBufferFromARGB()
{
    if (camera != null && previewBuffer != null)
    {
        // Get preview Image buffer (640x480)
        try { camera.GetPreviewBufferArgb32(previewBuffer); }
        catch { return null; }

        // Select outputBuffer pixels (outputSize = smaller than 640x480 preview pixels)
        CopyValuesToOutputBuffer();

        // Swap Red and Blue channel (Silverlight -> XNA's texture)
        SwapRedBlueChannel(outputBuffer);
        return outputBuffer;
    }
    return null;
}
```

This method retrieves the data from the camera to preview buffer (640x480). Then the selected values are copied to the smaller output buffer (see the next paragraph for the method implementation). Output colors are swapped to correct XNA's `Texture2D` format and finally the output buffer is returned.

The output buffer is a linear array of integer values. Every integer has the compressed info about the whole RGB value of the pixel and its alpha (transparency). In the field `outputSize` we have the X and Y size of the output image.

Helper methods

The first method is used to copy values from the preview to output buffer. Here you can find quite a complex variant. This implementation automatically crops the image by the correct output ratio. The image is also auto-rotated and scaled.

```
private void CopyValuesToOutputBuffer()
{
    // Copies data from preview buffer (640x480) to smaller output buffer in correct ratio
    Point start = Point.Zero, incr = Point.Zero;
    GetOutputParams(ref start, ref incr);

    // Inserts values to the output buffer
    for (int y = 0; y < outputSize.Y; y++)
        for (int x = 0; x < outputSize.X; x++)
        {
            // Auto flip / rotate the image to output
            int sourceX = Math.Min(start.X + y * incr.X, previewSize.X - 1);
            int sourceY = Math.Min(previewSize.Y - (start.Y + x * incr.Y) - 1, previewSize.Y - 1);
            int i = sourceX + sourceY * previewSize.X;
            outputBuffer[x + y * outputSize.X] = previewBuffer[i];
        }
}
```

```
private void GetOutputParams(ref Point start, ref Point incr)
{
    // Returns correct params for the output buffer
    // = start index & increment, how should we read from the preview buffer
    start = Point.Zero;
    incr = new Point(previewSize.X / outputSize.Y, previewSize.Y / outputSize.X);

    if (previewSize.X / (float)previewSize.Y > outputSize.Y / (float)outputSize.X)
    {
        // Preview is wider, output buffer will be cropped at left/right side
        start.X = (int)((previewSize.X - outputSize.Y * previewSize.Y / (float)outputSize.X) / 2);
        incr.X = (previewSize.X - 2 * start.X) / outputSize.Y;
    }
    else
    {
        // Output buffer is wider (preview is taller, crop at top/bottom)
        start.Y = (int)((previewSize.Y - outputSize.X * previewSize.X / (float)outputSize.Y) / 2);
        incr.Y = (previewSize.Y - 2 * start.Y) / outputSize.X;
    }
}
```

Note: We are using the integer division for calculating the increment (for performance reasons). This will work fine if the output buffer is sufficiently smaller than the camera preview. If the output and preview sizes are almost similar, this method may give you poor results.

Note: You can find a new object for reading from the camera in the WP8 API: `PhotoCaptureDevice`. It is possible to set there a preview size of the buffer directly (so you do not need to call this method anymore). Unfortunately this API is not available for older WP7 devices.

The next method converts the image buffer into XNA format:

```
private void SwapRedBlueChannel(int[] buffer)
{
    for (int i = 0; i < buffer.Length; ++i)
```

```
    {
        // Converts a packed RGB integer value from Silverlight's to XNA format
        // - we need to switch a red and blue channel
        buffer[i] = (int)((uint)buffer[i] & 0xFF00FF00)
                | (buffer[i] >> 16 & 0xFF)
                | (buffer[i] & 0xFF) << 16;
    }
}
```

And finally, we should include to our `CamReader` class the method for stopping the camera. It will be called when we leave the app, or when the app was switched to background:

```
public void StopCamera(Canvas fakeCamCanvas)
{
    if (camera != null)
        camera.Dispose();
    camera = null;
    previewBuffer = null;
    fakeCamCanvas.Background = null;
}
```

Rendering the live stream

Now we have prepared a helper class **CamReader.cs**. We would like to integrate this logic into our app.

Let's open a **GamePage.xaml.cs** file. This is the place where we can write the code for the XNA Framework.

Add these declarations to this file:

```
Point outputSize;
CamReader camReader;
Texture2D texture;
```

And add this initialization logic to method `OnNavigatedTo`:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    //...
    // Initialize the camera reader + XNA's texture (of the correct size)
    GraphicsDevice device = SharedGraphicsDeviceManager.Current.GraphicsDevice;
    outputSize = new Point(240, 240);
    texture = new Texture2D(device, outputSize.X, outputSize.Y);

    camReader = new CamReader();
    camReader.StartCamera(this.Dispatcher, cameraDisplayFake, outputSize);
    //...
}
```

Our `CamReader` object and the texture will be initialized. Note that we are passing a `cameraDisplayFake` object to our `StartCamera` method. This is the one component that we have already added into our XAML code.

We can insert a similar logic into `OnNavigatedFrom`:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    camReader.StopCamera(cameraDisplayFake);
    //...
}
```

Now we can finally render our photo stream on the display! Method `OnDraw` is called 30 times per second by default. We obtain a data from `CamReader` object every frame, then copy this data to the `texture` and draw this `texture`.

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    GraphicsDevice device = SharedGraphicsDeviceManager.Current.GraphicsDevice;
    device.Clear(Color.CornflowerBlue);

    // Get camera values, draw image directly on screen by SpriteBatch (by converting into Texture2D)
    int[] buffer = camReader.GetBufferFromARGB();
    if (buffer != null)
    {
        //ApplyEffect(buffer, outputSize, effect);   // Applies special filter on image
        device.Textures[0] = null;
        texture.SetData<int>(buffer);

        spriteBatch.Begin();
        spriteBatch.Draw(texture, new Vector2(0, 0), Color.White);
        spriteBatch.End();
    }
}
```

Note: We can change the interval of the drawing loop from standard 30 frames per second (FPS) to a different value. You only need to change the `timer.UpdateInterval` in the `GamePage` constructor into `TimeSpan.FromTicks(166666);` and the rendering will run at amazing 60 FPS!

Note: We can combine a XAML code (Silverlight) and the XNA content on one page together. XAML elements can be rendered into texture and displayed even over the content from the XNA. Similar situation is on the WP8, C#/XAML content can be also combined with MonoGame Framework at one page.

Working with image data

XNA is a very powerful framework for manipulating with images. In this sample we are drawing the texture on a point 0,0 (into the upper left corner), in 100% zoom. We can easily change the rendering code to draw this image two times bigger (= in 200% zoom):

```
spriteBatch.Draw(texture, Vector2.Zero, null, Color.White, 0f, Vector2.Zero, 2f, SpriteEffects.None, 0);
```

For this texture - we can also set the scale, the rotation, transparency. Before we convert our buffer into the texture, we can also apply any special filter on these data, for example: grayscale, sepia, lomo, vintage... Anything what we can imagine.

8-bit style renderer

So, we have a raw data from the camera. We do not need to convert them to the texture directly. We can work with this values and, for example, we can draw an extra image for every pixel!

We can make an interesting 8-bit looking app with a real live preview. The power of the XNA Framework will handle it (the slowest Windows Phones can draw around 120x120 non-transparent textures at once, in more than 20 FPS).

This 8-bit effect can look like this:

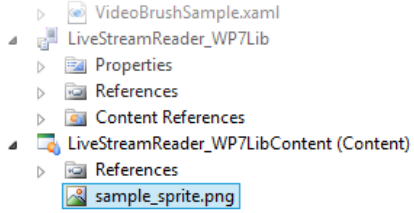
Note: These sample images have been generated with some advanced filters. Colors were restricted to a limited palette (for example only 16 specific colors) with a specified threshold. Variants of the Ordered Dithering and Floyd-Steinberg algorithm were applied.

Rendering pixels from images

We will show here the most simple version of this 8-bit renderer. Every pixel obtained from the camera will be drawn as a small image, colored with the RGB data. For example - our sample image will be 6x6 pixels big. If we set the output buffer of the camera to 80x80 pixels, the final image will be 480x480 px big.

You can download the sample image here:

And add this image to your Content project:



Now we can load this image into our app. Add to your **GamePage.xaml.cs** this declaration:

```
Texture2D sprite;
```

Modify the `OnNavigated` method like this:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    //...
    // Initialize the camera reader + small Sprite texture
    sprite = contentManager.Load<Texture2D>("sample_sprite");
```

```
    outputSize = new Point(80, 80);
    camReader = new CamReader();
    camReader.StartCamera(this.Dispatcher, cameraDisplayFake, outputSize);
    //...
}
```

The image will be loaded to the `sprite` variable.

Note: We are using a standard XNA Content pipeline, the image is converted to the uncompressed **.xnb** file, then loaded into the texture. We can also load this image directly from .png file.

Now we can draw the values. Modify the `OnDraw` method like this:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    //...
    // Get camera values, draw image
    int[] buffer = camReader.GetBufferFromARGB();
    if (buffer != null)
    {
        // Draw a pixelated 8-bit effect
        spriteBatch.Begin();
        Draw8BitStyle(spriteBatch, buffer);
        spriteBatch.End();
    }
}
```

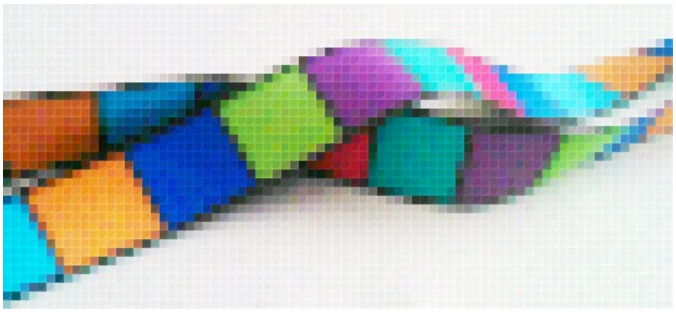
Our method `Draw8BitStyle` will simply read the values from the buffer, extract the color data and draw the sprite for every pixel:

```
private void Draw8BitStyle(SpriteBatch spriteBatch, int[] buffer)
{
    // Draw a pixelated 8-bit effect from the buffer
    for (int y = 0; y < outputSize.Y; y++)
        for (int x = 0; x < outputSize.X; x++)
        {
            int val = buffer[y * outputSize.X + x];

            // Convert value into color, draw the sprite
            int a = val >> 24;
            int b = (val & 0x00ff0000) >> 16;
            int g = (val & 0x0000ff00) >> 8;
            int r = (val & 0x000000ff);

            Vector2 position = new Vector2(x * sprite.Width, y * sprite.Height);
            spriteBatch.Draw(sprite, position, new Color(r, g, b));
        }
}
```

Our live camera stream will look like this now. This is awesome!



Tip: If you want to know how to convert RGB values back into one integer value, this method will help you:

```
private int PackRGBValue(byte r, byte g, byte b)
{
    // Pack individual RGB components into one single integer
    uint output = 0xff000000; // Alpha
    output |= (uint)b;
    output |= (uint)(g << 8);
    output |= (uint)(r << 16);
    return unchecked((int)output);
}
```

Note: You can find both variants of the algorithm in the attached sample project. The component for measuring frames per second (FPS) from the XNA is also attached.

Reading from the YCbCr buffer

Our photo stream from the camera looks great. But it's still not perfect. This implementation is horribly slow on the first generation of Windows Phone devices. For example on the Samsung Omnia 7 Device, the 120x120 pixels large scene composed of 4x4 px sprites runs only at **6 FPS**! Can we fix it?

Fortunately there is a second method on the `PhotoCamera` object for reading raw values from the camera: `GetPreviewBufferYCbCr`. It is little more complicated to use, but gives us much better results. The main difference between these methods is - when we read an ARGB data from the camera, the whole 640x480 buffer is internally converted into RGB values. It doesn't matter that we wanted only a 120x120 px small array. All values had to be converted.

Method `GetPreviewBufferYCbCr` returns a buffer from the camera in the "more raw" format. We will convert to RGB values only those pixels that we really

want. I can reveal you in advance that this implementation will run on the Samsung Omnia 7 Device faster than **20 FPS**!

Note: Do not worry about Nokia Lumia product line. Both implementations are sufficiently fast even on the Nokia Lumia 710 or 800 Device. The ARGB method runs at a stable **15 FPS**, the `YCbCr` method runs at **22 FPS**.

Note: Windows Phone 8 devices will be even faster, on the Windows Phone 8X Device by HTC the ARGB method runs at **27 FPS** and the faster YCbCr method is attacking **30 FPS** (we are talking about WP7 application in a compatible mode, the native WP8 solution will be discussed in the last chapter of this article).

Implementation

We will extend our `CamReader` class. Add here these declarations:

```
// Special buffer for the YCbCr reader
YCbCrPixelLayout bufferLayout;
byte[] yCbCrPreviewBuffer;
```

The `StartCamera` method will also be slightly modified. Add these initialization logic to the `dispatcher.BeginInvoke()` call:

```
// Buffer initialization for reading from the YCbCr buffer
bufferLayout = camera.YCbCrPixelLayout;
yCbCrPreviewBuffer = new byte[bufferLayout.RequiredBufferSize];
```

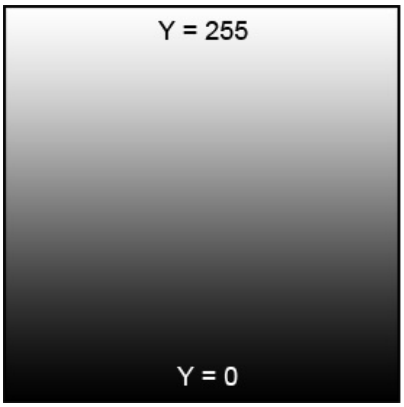
And finally we can add here a new method - `GetBufferFromYCbCr`. It is the alternative to the `GetBufferFromARGB`, it returns data in the same format (but it is faster).

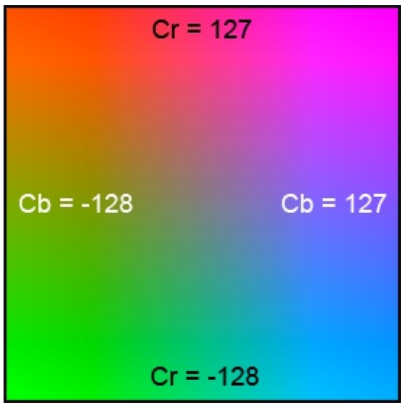
```
public int[] GetBufferFromYCbCr()
{
    if (camera != null && yCbCrPreviewBuffer != null)
    {
        // Get preview Image buffer
        try { camera.GetPreviewBufferYCbCr(yCbCrPreviewBuffer); }
        catch { return null; }

        // Select outputBuffer pixels (outputSize = smaller than 640x480 preview pixels)
        CopyYCbCrValuesToOutputBuffer();
        SwapRedBlueChannel(outputBuffer);
        return outputBuffer;
    }
    return null;
}
```

YCbCr to RGB conversion

YCbCr is a pretty interesting color model. Y stands for the Luminance (brightness), Cr and Cb are Chrominance values (they represents the color difference). If we load to our app only a Luminance (Y) value, we will obtain a grayscale image.





Now we need to implement the last few helper methods. We want to select the specific pixels from the YCbCr buffer, extract the correct Y, Cr and Cb values, convert them back to the ARGB and finally to save the value of each pixel to the output buffer.

```
private void CopyYCbCrValuesToOutputBuffer()
{
    // Copies data from YCbCr buffer to smaller output buffer (in correct ratio)
    Point start = Point.Zero, incr = Point.Zero;
    GetOutputParams(ref start, ref incr);

    for (int y = 0; y < outputSize.Y; y++)
        for (int x = 0; x < outputSize.X; x++)
        {
            // Auto flip / rotate the image to output
            int sourceX = Math.Min(start.X + y * incr.X, previewSize.X - 1);
            int sourceY = Math.Min(previewSize.Y - (start.Y + x * incr.Y) - 1, previewSize.Y - 1);

            // Read Y, Cb and Cr values of the pixel
            byte yValue;
            int cr, cb;
            GetYCbCrFromPixel(bufferLayout, yCbCrPreviewBuffer, sourceX, sourceY, out yValue, out cr, out cb);

            // Convert to RGB and write to output buffer
            int rgbPackedValue = YCbCrToArgb(yValue, cb, cr);
            outputBuffer[x + y * outputSize.X] = rgbPackedValue;
        }
```

```
}
```

```
private void GetYCbCrFromPixel(YCbCrPixelLayout layout, byte[] currentPreviewBuffer, int xFramePos, int yFramePos, out byte y, out i
{
    // Parse YCbCr value from given pixel
    // - finds the bytes corresponding to the pixel location in the frame.
    int yBufferIndex = layout.YOffset + yFramePos * layout.YPitch + xFramePos * layout.YXPitch;
    int crBufferIndex = layout.CrOffset + (yFramePos / 2) * layout.CrPitch + (xFramePos / 2) * layout.CrXPitch;
    int cbBufferIndex = layout.CbOffset + (yFramePos / 2) * layout.CbPitch + (xFramePos / 2) * layout.CbXPitch;

    y = currentPreviewBuffer[yBufferIndex];
    cr = currentPreviewBuffer[crBufferIndex];
    cb = currentPreviewBuffer[cbBufferIndex];

    // Cr and Cb will be from -128 to 127, Y (luminance) from 0 to 255
    cr -= 128;
    cb -= 128;
}
```

Note: These methods for unpacking YCbCr values and converting them into RGB were inspired by the former article on the MSDN (Windows Phone 7 Camera Color Conversion). The content is no longer available there.

Especially the next method for converting YCbCr value to the RGB looks totally devilish. But don't worry, it's a pretty standard algorithm. You can find lots of implementations of it on the internet. This conversion uses the integer-only divisions again (for the performance reasons).

```
private int YCbCrToArgb(byte y, int cb, int cr)
{
    // Converts YCbCr to packed RGB
    int r, g, b;
    uint argbPixel;

    // Integer-only division.
    r = y + cr + (cr >> 2) + (cr >> 3) + (cr >> 5);
    g = y - ((cb >> 2) + (cb >> 4) + (cb >> 5)) - ((cr >> 1) + (cr >> 3) + (cr >> 4) + (cr >> 5));
    b = y + cb + (cb >> 1) + (cb >> 2) + (cb >> 6);

    // Clamp values to 8-bit RGB range between 0 and 255.
    r = r <= 255 ? r : 255;
    r = r >= 0 ? r : 0;
    g = g <= 255 ? g : 255;
    g = g >= 0 ? g : 0;
    b = b <= 255 ? b : 255;
    b = b >= 0 ? b : 0;

    // Pack individual components into a single pixel.
    argbPixel = 0xff000000; // Alpha
    argbPixel |= (uint)b;
    argbPixel |= (uint)(g << 8);
    argbPixel |= (uint)(r << 16);

    // Return the ARGB pixel.
    return unchecked((int)argbPixel);
}
```

Now if you switch your rendering logic in the **GamePage.xaml.cs** file to this line, you should see an acceleration in your frame rate speed:

```
int[] buffer = camReader.GetBufferFromYCbCr();
```

Windows Phone 8

We were talking about Windows Phone 7 devices so far. But what about newer WP8 phones? Every WP8 device can run older WP7 applications in the compatibility mode. If you run this sample on your WP8 device, it will work correctly.

But there are some limitations. Such an obsolete app runs only in fixed 800x480 resolution, we don't have an access to bigger resolutions of the HD screens. On some devices with a different screen ratio (HTC 8X), the black border at the top is displayed. We can access some newer APIs from the old app via reflection, but this is also limited. If we want to take full advantage of the new WP8 API (while still supporting the older WP7 devices), we need to make two apps. The Windows Phone Store is prepared for this scenario, we can upload two **.xap** files into one app submission, the correct version will be automatically downloaded for user.

Fortunately the WP8 API is very similar to the older WP7 API. Maybe they don't call it Silverlight anymore, but there is still the same C# and XAML. Except the one missing component... The XNA Framework. A minute of silence, please... If we select a new WP8 project, we have no chance to access this very powerful graphics engine. The whole idea about the efficient drawing images is now gone. So, how to solve this?

MonoGame Framework

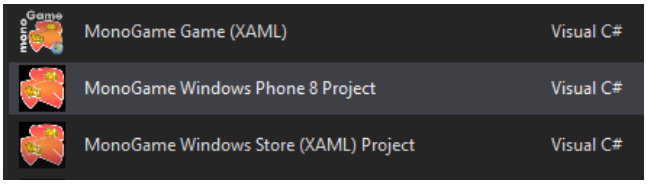
There is a solution! MonoGame framework is an open source alternative to the XNA Framework. It is a light wrapper around the native DirectX calls and provides us the almost identical API to the XNA. It is a project with very active community and a very good support from the developers. This framework allows you to port your apps and games to WP8, Windows 8 or even on the Android and iOS.

I can recommend this framework. At this moment the Windows Phone 8 version is almost without any issues. The fast app switching works. There is already a working WP8 template in the installer. Maybe, I only did not manage how to change a resolution of the graphics buffer in the latest version. But I hope this will be fixed soon.

Note: You can also compile this framework by yourself from the source code. There is already one article on this wiki how to set up the MonoGame: XNA Games On Windows Phone 8 with Monogame, but it is a little outdated now. Most of the problems are fixed now. You can just install the installer, set up a new template project (or link a MonoGame **.dll**) and everything will work fine, like if you are using a real XNA.

Windows Phone 8 port

We can switch this app to the Windows Phone 8 API. You will need to install a MonoGame Framework and create a new Windows Phone Game project in the Visual Studio 2012.



Our helper **CamReader.cs** object will stay the same. We can move this file into another project, or into some Portable Library, if we want to share it between the projects.

XAML code from the original **GamePage.xaml** will also stay the same. The code-behind in the **GamePage.xaml.cs** will be slightly different. In the MonoGame WP8 template, this logic was split into two files. We will only keep the main C#/XAML logic in this file, the XNA logic will be moved into **Game1.cs** file. Take a look into the attached source code: Media:LiveStreamReader.zip

In this package there is a working WP8 project, with the actual compiled MonoGame binaries. There are also both **.xap** installation files for WP7 and WP8 versions, you can try them on your phone.

Note: This approach (with a split logic into two files) is more similar to the standard XNA template. If you ever used the basic XNA WP7 template, there was also the **Game1.cs** class there. Only the combined XNA/Silverlight WP7 project has a different name convention.

Note: Do not forget that MonoGame has still one limitation. The Content Pipeline is not working yet in this framework (converting **.png** images into the uncompressed **.xnb** files). You need to copy your .xnb files directly to the project (into the Content folder), or to load these images directly from the stream without the Content Pipeline (as was mentioned before).

The results

We have made two applications, one specific for WP7 and the second for WP8. Both apps share the same core logic, the first app is rendered with the XNA Framework, the WP8 version uses MonoGame. If we start both apps on the same WP8 device, we will see that their performance is almost similar. For example on the Lumia 720 Device both implementations run at **20 FPS**.

WP8 version can take advantage of the all new APIs, in runs also in a native resolution of the display. This is the reason why this implementation can be a little slower on devices with HD displays. You need to render more than 2.5x more pixels (than in the 800x480 px WP7 version).

Summary

In this tutorial we have learned how to efficiently read a live stream from the camera and how to display it in the app. Our implementation runs smoothly on both Windows Phone 7 and 8 devices. We have discussed a couple of methods how to make this solution better and faster.

The interesting use of this algorithm was presented, we have learned how to make an original 8-bit looking photo app. We have figured out how we can integrate other live filters and effects to this app.

We have managed to build this app without any line written in C++, all presented code was written in C#. We have learned how to combine a Silverlight and XNA Framework together into one WP7 solution and how to achieve a similar goal with the MonoGame Framework on Windows Phone 8.