Memory-efficient Navigation in Very High Resolution Images on Windows Phone

This article delivers a re-usable UI control which enables memory-efficient zoom, pan and rotate inside very high resolution (or gigapixel) images.

08 Sep 2013

Note: This article has been updated to Nokia Imaging SDK v1.0

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

Introduction

Very high resolution images, like those produced with the Lumia 1020 (38Megapixels) or even gigapixel images (e.g. Almeida Júnior - Saudade - 450 MegaPixels and Bernat Martorell - Altarpiece of Saint Vincent - 709 MegaPixels) can consume enormous amounts of memory. Without clever and efficient design, apps implementing common imaging operations quickly run into memory and performance limitations.

The Nokia Imaging SDK uses RAJPEG technology to load and decode only a part of a JPG picture, which is much more memory-efficient than working with the whole image. This article explains how to use the SDK to implement a UI control that will make it possible to zoom, pan and rotate inside a very high resolution image.

The sample code includes classes, under MIT licence, which can easily be re-used to provide these effects in your own applications.

Problem overview

To work with high resolution images they first need to be decoded. On Windows Phone you must manipulate 4-byte `ARGB` pixels, so a high resolution picture captured with the Lumia 1020 (dimension 7712 × 4352) requires a decoded pixel buffer size of 7712 × 4352 * 4 = **128 MB**. A Gigapixel image like Almeida Júnior - Saudade needs a decoded pixel buffer size of **1.68 GB** or Bernat Martorell - Altarpiece of Saint Vincent need **2.8 GB**.

Unfortunately, Windows Phone 8 limits application memory to 150 MB and 300MB for low and high memory devices (respectively). While it is possible to increase this limit to 180MB and 380MP - 570MB with the ID_FUNCCAP_EXTEND_MEM capability, decoding a high resolution picture will still use a significant amount of the allowed memory (particularly on low-memory phones). In addition, Windows Phone 8 limits the texture size to 4096x4096 and downscales high resolution images. If picture dimensions are too high, for example when working with a Gigapixel image, an `OutOfMemory` exception is raised internally and picture is not fully decoded.

The Nokia Imaging SDK uses RAJPEG technology to load and decode only a part of a JPG picture, which is much more memory-efficient than working with the whole image. It also provides the *Reframing Filter* for accurate oriented region of interest (ROI) extraction - which is an important for implementing many of the key UI operations used in computer imaging:

- Picture transformations (translate, scale, rotate} and equivalent for XAML `CompositeTransform`.
- Reframing
- Image alignment

This *ReframingFilter* is used to implement controls to zoom, pan and rotate inside a very high resolution image.

Pre-requisites

A basic understanding of how the Imaging SDK is used is recommended (but not essential). The links below provide a good starting point:

- Core concepts (Nokia Imaging SDK documentation)
- Download and add the libraries to the project (Nokia Imaging SDK documentation)
- Quick Start (Nokia Imaging SDK documentation)
- How to Use the Nokia Imaging SDK to create a "Photo Cookbook" for Windows Phone 8 Devices
- Filter Testing app using the Imaging SDK

This article uses the *Interactive State machine* explained in Optimizing Imaging SDK use for rapidly changing filter parameters to optimize navigation.

Reframing Filter

The key to navigating/manipulating a high resolution image is to be able to efficiently extract a specific area. The Imaging SDK

(C) Copyright Nokia 2014. All rights reserved.

1.0 provides the *Reframing Filter* which can extract an oriented area from an `ImageProvider`. It's parameters are :

- `ReframingArea`: Rectangle describing the position and size of the reframing area.
- `PivotPoint`: The point around which rotation is done. By default it's the `ReframingArea` center.
- `Angle`: Rotation of the reframing area clockwise around the `PivotPoint`.

Note: The Imaging SDK is very optimized and only final pixels will be decoded.

Warning: Imaging SDK 1.0 performance for re-framing is affected by the angle used, and can degrade by as much 500%. This problem should be corrected in a future version.

Picture Navigation

The method described for extracting an oriented ROI is very efficient, but it is still a computationally intensive and time consuming task. In order to remain responsive to user input while navigate within a very high resolution (or gigapixel) image, and consume the minimum memory, we need to ensure that it is called only when necessary.

The first section below explains how we process and use gestures to navigate within a UI control. The following two sections explain methods to navigate efficiently in high resolution and gigapixel images. Both methods use an Interactive State Machine to ensure that during a gesture only one image is processed at a time. Both methods also display a lower resolution image during the gesture to make rendering more responsive to user input, and only display the final higher resolution image when the gesture is complete.

The main difference between the methods is how they generate the low resolution image while the gesture is being performed. The first method simply extracts a lower resolution image, trading off resolution for improved performance. The second pre-extracts a low resolution image for the whole picture and displays this during navigation using XAML transforms.

The methods are benchmarked for maximum and minimum rendering time on images with different (very high!) resolutions, using a Lumia 1020 + GDR3.

Gesture

This section explains how we process and use gestures to navigate in the picture. We will use gesture to compute the following position data:

- `ReframingArea` center
- `Rotation angle`: in degrees. *FreeRotationFilter* and `CompositeTransform` use angle in degrees.
- `Scale`: factor between the `ReframingArea` and UI control size.

To control gestures, we need the UI control size. In code, this size is represented by `outputSize`.

To receive Gesture event, we will add a delegate on the UI control which displays the image :

- `ManipulationStarted`: user touches the screen. Initialize gesture control.
- `ManipulationDelta`: user moves fingers. Compute new position data.
- `ManipulationCompleted`: user finishes gesture. Save last position data.

This event manipulates two types of gestures:

- `Translation`: translate the picture.

(C) Copyright Nokia 2014. All rights reserved.

- `Pinch`: zoom and orientation of the picture.

When the user makes a gesture, this can add and remove fingers. We need to know when the gesture type changes. We simply create an enum and a member which saves the last gesture type.

```
enum GESTURE_TYPE
{
    NONE,
    TRANSLATION,
    PINCH
};
GESTURE_TYPE oldGestureType = GESTURE_TYPE.NONE;
```

To optimize navigation we will optimize the rendering during the user gesture, and we need an information to know if we need to process a low or high resolution rendering. We simply add an enum and a member

```
enum RESOLUTION
{
    LOW,
    HIGH
};

RESOLUTION outputResolution = RESOLUTION.LOW;
```

When `ManipulationStarted` is raised, we initialize gesture information and begin low resolution rendering

```
public virtual void ManipulationStarted(object sender, ManipulationStartedEventArgs arg)
{
    oldGestureType = GESTURE_TYPE.NONE;
    outputResolution = RESOLUTION.LOW;
    saveLastPositionData();
}
```

When `ManipulationCompleted` is raised, we request a high resolution rendering

```
public virtual void ManipulationCompleted(object sender, ManipulationCompletedEventArgs
arg)
{
     outputResolution = RESOLUTION.HIGH;
     requestProcessing();
}
```

The most important event is `ManipulationDelta`. When it is raised, we need to compute new position data in function of the gesture type and fingers position/move. To simplify this computation, we use `CompositeTransform` which is compatible with our position data.

The simplest gesture type is the translation. This information is accessible from `ManipulationDeltaEventArgs`. This type is used to compute the new ROI centre position. We must convert screen translation in Picture translation in function of scale and rotation.

```
oldGestureType = GESTURE_TYPE.TRANSLATION;

var translation = arg.CumulativeManipulation.Translation;

//create CompositeTransform with scale and rotation data.
CompositeTransform gestureTransform = new CompositeTransform();
gestureTransform.ScaleX = gestureTransform.ScaleY = scale;
gestureTransform.Rotation = rotation;
```

(C) Copyright Nokia 2014. All rights reserved.

```
//apply inverse transformation
translation = gestureTransform.Inverse.Transform(translation);

//compute new ROI center position
newPos.X = oldPos.X - translation.X;
newPos.Y = oldPos.Y - translation.Y;
```

A pinch gesture is use to control the zoom and the rotation. When the user touches with two fingers `ManipulationStartedEventArgs.PinchManipulation` is not null. To compute new position data we will compute:

- `deltaScale` : pinch scale
- `deltaRoration` : pinch rotation
- `translation` : picture navigation follows user fingers.

Pinch Manipulation gives current and first origin finger position. `deltaScale` is the ratio between the gesture origin fingers distance and current gesture fingers distance.

```
var p1 = arg.PinchManipulation.Original.PrimaryContact;
var p2 = arg.PinchManipulation.Original.SecondaryContact;
var p3 = arg.PinchManipulation.Current.PrimaryContact;
var p4 = arg.PinchManipulation.Current.SecondaryContact;


deltaScale = Math.Sqrt((p4.X - p3.X) * (p4.X - p3.X) + (p4.Y - p3.Y) * (p4.Y - p3.Y))
                    /
                    Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y - p2.Y) * (p1.Y -
p2.Y));
```

To compute `deltaRotation` we use a solution provided by Real-time rotation of the Windows Phone 8 Map Control

```
public static double angleBetween2Lines(PinchContactPoints line1, PinchContactPoints
line2)
{
    if (line1 != null && line2 != null)
    {

        double angle1 = Math.Atan2(line1.PrimaryContact.Y - line1.SecondaryContact.Y,
                                   line1.PrimaryContact.X - line1.SecondaryContact.X);
        double angle2 = Math.Atan2(line2.PrimaryContact.Y - line2.SecondaryContact.Y,
                                   line2.PrimaryContact.X - line2.SecondaryContact.X);
        double angle = (angle1 - angle2) * 180 / Math.PI;

        return angle;
    }
    else { return 0.0; }
}
```

Note: The Nokia Imaging SDK uses an angle between [0, 360] degrees. The sample code recomputes current angle in this interval.

When the user move their fingers, the image displayed under a finger should be always the same so we need to translate the ROI centre position. To compute it, we will compute origin relative finger position and current relative finger position. The translation is the difference between this two position. We will use the first contact information.

```
// Translate manipulation
var originalCenter = arg.PinchManipulation.Original.PrimaryContact;
{
   //UI control center match with ROI center
    originalCenter.X -= outputSize.Width / 2;
```


(C) Copyright Nokia 2014. All rights reserved.

```
    originalCenter.Y -= outputSize.Height / 2;

   //CompositeTransform with original scale and rotation
    CompositeTransform gestureTransform = new CompositeTransform();
    gestureTransform.Rotation = rotation;
    gestureTransform.ScaleX = gestureTransform.ScaleY = scale;
    originalCenter = gestureTransform.Inverse.Transform(originalCenter);
}

var currentCenter = arg.PinchManipulation.Current.PrimaryContact;
{
   //UI control center match with ROI center
    currentCenter .X -= outputSize.Width / 2;
    currentCenter .Y -= outputSize.Height / 2;

    //CompositeTransform with current scale and rotation.
    CompositeTransform gestureTransform = new CompositeTransform();
    gestureTransform.Rotation = rotation + deltaRoration;
    gestureTransform.ScaleX = gestureTransform.ScaleY = (scale * deltaScale);
    currentCenter = gestureTransform.Inverse.Transform(currentCenter );
}
//compute new ROI center position.
currentPos.X = originPos.X - (currentCenter .X - originalCenter.X);
currentPos.Y = originPos.Y - (currentCenter .Y - originalCenter.Y);
```

Method 1: Imaging SDK

In this method, we use the Nokia Imaging SDK to extract the ROI for each generated position. An Interactive State Machine ensures that while the user is supplying input the code only renders a new image (based on most current input) when the previous one has completed. We also optimize for rendering duration - trading off the output resolution for reduced calculation time while the user is supplying input. Both strategies are described in Optimizing Imaging SDK use for rapidly changing filter parameters.

The sample code which implements this solution is in **Method1/Method1Filter.cs**.

For this solution we only need to convert position data for the `ReframingFilter`:

- angle is simply the current angle * -1
- {{lcode|ReframingArea} is computed from `currentScale` and current centre.

```
var currentSize = new Size(
                   outputSize.Width / currentScale ,
                   outputSize.Height /currentScale );
var corner = new Point(currentPos.X - currentSize.Width / 2, currentPos.Y -
currentSize.Height / 2);
var rect = new Rect(corner, currentSize)

reframing.ReframingArea = new Rect(corner, currentSize);
reframing.Angle = -1 * currentAngle;

await renderer.RenderAsync();

outputBitmapTmp.Pixels.CopyTo(outputBitmap.Pixels, 0);
outputBitmap.Invalidate();
```

Benchmarking

The table below shows the memory use and rendering time achieved using this method for different sized images using "Method 1". Note that using this method the low resolution rendering is performed during user input, while the high resolution rendering is performed after user input stops.

Note: Re-framing performance is significantly affected by the angle used. For the purpose of benchmarking the calculations have been done with angle equal to zero. When the problem is resolved, a new benchmark will be done with non-zero angles.

(C) Copyright Nokia 2014. All rights reserved.

Note: Application start with 27 MB Memory use and JPG file must be copied in memory.

Image size	File size	Memory use	Low Resolution rendering	High Resolution rendering
2592 x 1936	2.25 MB	34 MB	33 ms - 274 ms	83 ms - 300 ms
7728 x 4354	9.85 MB	50 MB	31 ms - 664 ms	70 ms - 637 ms
15036 x 30000	131 MB	266 MB	38 ms - 446 ms	161 ms - 711 ms
24794 x 30000	253 MB	474 MB	49 ms - 440 ms	136 ms - 513ms

The measurements show that:

- Memory use is much lower than when loading the whole image into memory, and depends on image file size
- Rendering time depends on output resolution and scale. It increases with zoom out.
- Processing time is short enough that rendering tracks user interaction properly.
- High and low resolution rendering quality depend only on the factor between these two resolutions. Factor in the Sample is **2**.

Image size	Low Resolution rendering	High Resolution rendering
15036 x 30000		

Tip: The reason that rendering time increases with zoom out is due to the macroblock structure of the JPEG compression.

The JPEG compression works by processing 8x8 pixels macroblocks. Decoding a macroblock is a pretty expensive operation, as it involves a cosine transform and some other calculations. When zooming at the pixel level, decoding/processing one macroblock will provide 64 pixels of the final image (a macroblock is 8x8 pixels). As you zoom out, the pixels needed to create the final image are further and further apart : you need to decode more and more macroblocks to fetch the pixels that will be on the final image, and while each macroblock that must decoded is still a 8x8 macroblock, only a few of the pixels of the block will be used in the final image, others are discarded. The increase in the amount of macroblocks that must be decoded explains the increase in the rendering time. When zoom out far enough, you may end up in a situation where you only need one pixel per macroblock.

Through some special tricks, it's really fast to get that single pixel - rendering becomes very fast!

You will find here a nice overview of the JPEG compression process.

Warning: The Imaging SDK returns an `InvalidArgument` exception when we use `ReframingFilter` with very large rectangle.

This means that we can't process rendering if we want display a large part of a GigaPixel picture

Method 2: Imaging SDK and `ImageBrush`

With the first method we extract and render a low resolution image while the user makes a gesture and then extract the high resolution image when they stop. While the Imaging SDK is efficient, the extraction cost is significant.

The second approach is very similar, except that instead of extracting new low resolution images during the gesture, we simply navigate (pan, rotate, zoom) within a pre-created low resolution image using XAML transformation. When the user completes their gesture we use the Nokia Imaging SDK (via our `OrientedROIFilter`) to extract the high resolution ROI. An Interactive State Machine ensures that the "high resolution" oriented ROI is only extracted when needed.

This approach is much less computationally expensive and extremely responsive to user input, at the cost of more memory used for the low resolution navigated within gestures - and lower quality during gestures when deeply zoomed. Sample code implementing this solution in **Method2/Method2Filter.cs**.

The first step is to generate the low resolution picture that is displayed with XAML transform while the user is performing a gesture. This is created from the input picture - we simply compute a factor to decrease the higher dimension to 4096 and use Imaging SDK to generate it.

(C) Copyright Nokia 2014. All rights reserved.

Warning: The Nokia Imaging SDK returns an `InvalidArgument` exception when we use `CropFilter` with very large rectangle. The sample code contains a function which cuts picture in NxN sub pictures to decode it. This method can take few seconds with very large picture.

To display the low resolution picture we use `ImageBrush`. First, we replace the `Image` control with a `Canvas` and set its background with an `ImageBrush`

```
brush = new ImageBrush();
output.Background = brush;
//We need to control all the transformation
brush.Stretch = Stretch.None;
//Orin point must be the top left of the picture.
brush.AlignmentX = AlignmentX.Left;
brush.AlignmentY = AlignmentY.Top;
```

When a low resolution rendering is processed, we use the low resolution picture as `ImageBrush` source and apply a transformation. We need to convert position data to a XAML `CompositeTransform`:

```
 var transform = new TransformGroup();
//Use a scaleTransform to resize low resolution picture dimension
transform.Children.Add(new ScaleTransform() { ScaleX = inputSize.Width /
InputLR.PixelWidth, ScaleY = inputSize.Height / InputLR.PixelHeight });
{
    CompositeTransform gestureTransform = new CompositeTransform();

   //Transform center is the ROI center
    gestureTransform.CenterX = newPos.X ;
    gestureTransform.CenterY = newPos.Y ;

    //scale and rotation data
    gestureTransform.Rotation = currentRotation;
    gestureTransform.ScaleX = gestureTransform.ScaleY = currentscale;

    //translate to match ROI center with the canvas center.
    gestureTransform.TranslateX = - newPos.X +outputSize.Width / 2.0;
    gestureTransform.TranslateY =  - newPos.Y+outputSize.Height / 2.0;
    transform.Children.Add(gestureTransform);
}
//set brush image source and its transform
if (brush.ImageSource != InputLR)
     brush.ImageSource = InputLR;

//set the transformation
 brush.Transform = transform;
```

When the user completes the gesture we need to extract and display a higher resolution image. The high resolution rendering processes a picture in *physical* pixel dimensions. So we need to rescale to logical dimensions to display the high resolution ROI picture:

```
outputBitmapHR.Invalidate();
 if (brush.ImageSource != outputBitmapHR)
{
    brush.ImageSource = outputBitmapHR;
    brush.Transform =  new ScaleTransform() { ScaleX = 100.0 /
System.Windows.Application.Current.Host.Content.ScaleFactor, ScaleY = 100.0 /
System.Windows.Application.Current.Host.Content.ScaleFactor };
}
```

Since we use a low resolution picture, and ROI extraction time increases with zoom out, we can avoid high resolution rendering when `current scale < InputLR.PixelWidth / inputSize.Width`. In this case low and high resolution rendering give similar

(C) Copyright Nokia 2014. All rights reserved.

results (the difference is only the pixel interpolation between Imaging SDK and XAML).

Warning: The sample app will sometimes display "black preview" when the user makes a gesture while an image is deeply zoomed. This is caused by a bug with XAML - you can avoid it by limiting the current scale value.

Benchmarking

The table below shows the memory use and rendering time achieved using method 2 for different image sizes.

Note: Re-framing performance is significantly affected by the angle used. For the purpose of benchmarking the calculations have been done with angle equal to zero. When the problem is resolved, a new benchmark will be done with non-zero angles.

Note: that the application uses 27 MB on start, and the JPG file must be copied in memory.

Image size	File size	Memory use	Low Resolution rendering	High Resolution rendering
2592 x 1936	2.25 MB	54 MB	~0	51 ms - 134 ms
7728 x 4354	9.85 MB	83 MB	~0	37 ms - 209 ms
15036 x 30000	131 MB	298 MB	~0	47 ms - 432 ms
24794 x 30000	253 MB	529 MB	~0	55 ms - 474 ms

The measurements show that:

- This approach uses more memory than method 1, but is still low enough for working with a 38MP picture on a low memory device.
- High resolution rendering time is dependent on the scale, and increases with zoom out.
- Low resolution rendering is near-instantaneous.
- The display when users make gestures is very smooth
- High and low resolution rendering quality depend on the factor between the input picture and it's low resolution version, and the interpolation method between Imaging SDK and XAML.
  - For a 38MP picture this factor is only 7728/4096 = **1.88**.
  - But for Gigapixel its 30000/4096 = **7.32**.

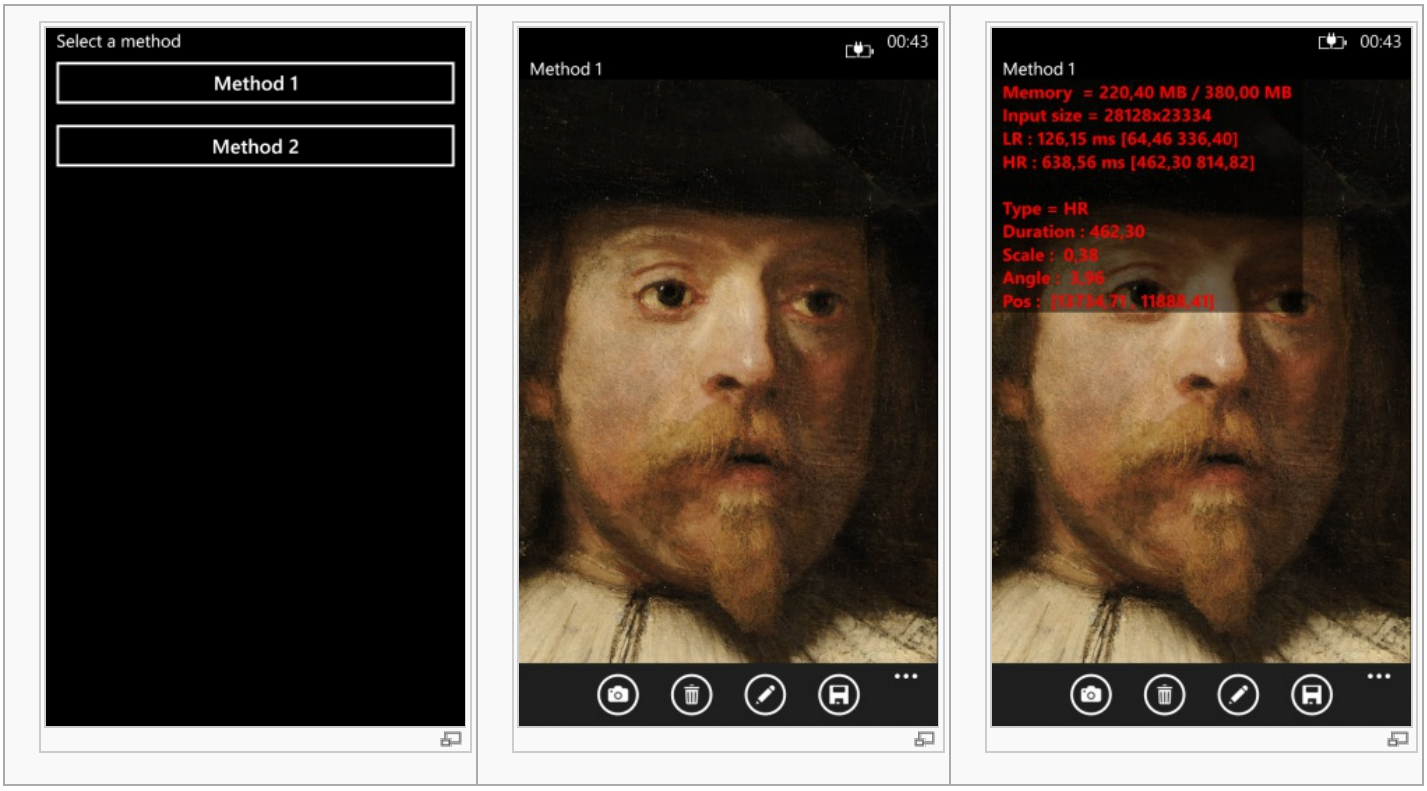
The images below show the difference in image quality during the gesture and on completion using method 2. As in the above point, the quality difference is greater for gigapixel images, but still acceptable for navigation purposes.

Image size	Low Resolution rendering	High Resolution rendering
7728 x 4354		
15036 x 30000		

(C) Copyright Nokia 2014. All rights reserved.

Sample code

The example code implements the two methods described above and displays information about their performance.



To test a method:

1. Run application in release.
2. Select a method.
3. Select a picture.
4. Navigate in picture.

The application provides the following options in the menu:

- `image`: select an image.
- `memory`: tell garbage collector to collect memory.
- `show/hide`: show and hide method information
- `ave roi`: extract the current oriented ROI and save in jpg format.

While you navigate in the picture, the test application can display the following information:

- Memory: current memory used in mega Byte and application memory limit,
- Input size: original Picture size,
- LR: Low Resolution rendering time=> mean - [min max] in millisecond,
- HR: High Resolution rendering time=> mean - [min max] in millisecond,
- Type: last rendering type,
  - LR: Low Resolution,
  - HR: High Resolution.
- Duration : last rendering time,
- Scale : current scale,
- Angle: current orientation,
- Pos : current ROI center.

High resolution pictures can be downloaded from here (these have been shown to work on both low and high memory devices):

- National Geographic : On assignment with the Nokia Lumia 1020
- Sample gallery: Nokia Lumia 1020.
- Nokia unveils 41-megapixel 808 PureView smartphone, threatens digital camera revolution
- A large collection of sample Nokia Lumia 1020 pictures from Flickr
- Nokia Lumia 1020 - © sheridan01

(C) Copyright Nokia 2014. All rights reserved.

GigaPixel images can be downloaded from the Google Art Project. These pictures work only on devices with more memory - and the jpg file size is limited to 180MB (To be used with Imaging SDK, it's copied in memory). Opening a Gigapixel picture will take a few seconds.

Note: The example uses `ID_FUNCCAP_EXTEND_MEM` capability to increase memory limit to 180MB, 380MP and 570MB

Warning : in the sample, the picture is accessed by a stream. So you should normally use `StreamImageSource`. Unfortunately, this class have a memory overhead.So, i copy the jpg file to a memory stream and use a `BufferImageSource`.

```
// var source = new StreamImageSource(s);
MemoryStream mstream = new MemoryStream();
s.CopyTo(mstream);
var source = new BufferImageSource(mstream.GetBuffer().AsBuffer());
```

</code>

How to add the control in your own application

The sample code provides a class for each method with the same interface:

- Method 1 is implemented in **Method1/Method1Filter.cs**
- Method 2 is implemented in **Method2/Method2Filter.cs**

Each class uses two public properties:

- `Input`: JPG stream of the high resolution picture
- `Output`: UI control will display the picture area.
  - Method 1 uses an Image control
  - Method 2 uses a Canvas control

and three public methods :

- `void Dispose()`: free unmanaged memory created by Imaging SDK.
- `IBuffer GenerateReframingPicture()`: return a buffer containing a JPG file of reframing area.
- `string Info()`: method information

To add one of these methods you have to instance the corresponding class and:

1. add an Image control for Method 1 or Canvas control for Method 2 in your page
2. set Output property with this control
3. set Input property with a picture jpg stream.
4. call `Dispose()` when you have finished.

You can find an integration example in **Method1/PageMethod1.xaml** and **Method2/PageMethod2.xaml** PhoneApplicationPage.

(C) Copyright Nokia 2014. All rights reserved.

Real example

The techniques described in this article have been used in my commercial "monsterification" (monster image editing) app: MonsterCam.

Monster Cam Tag

MonsterCam is one of first applications based on the Imaging SDK, and offers:

- Real time ROI extraction with gestures.
- Applying effects with user control.
- *Monsterification* is done with DirectX.

This application uses Method 1 to extract ROI and "monsterify" it with DirectX.

This application provides an unlimited free trial version and can be found on Windows Phone Store here

Summary

This article has delivered two re-usable UI controls which can be used to navigate within very high resolution (or gigapixel) images. Both methods can be used to process the sort of very high resolution images generated on the Lumia 1020 and are extremely responsive to user input - even on low memory devices.

The main differences are:

- memory use: the second method needs to create a low resolution picture.
- responsiveness when user performs a gesture: the first method uses `ReframingFilter` for each position while the second method displays a low resolution picture with a transformation. The second method provides a smoother interaction.

Unfortunately, due to their memory constraints and that file must be loaded in memory, low memory devices can display Gigapixel images with a lower resolution.

The methods provided have been used in the commercial app MonsterCam.

Reference

- Optimizing Imaging SDK use for rapidly changing filter parameters
- Real-time rotation of the Windows Phone 8 Map Control
- Image Recipes sample code (MSDN)
- Core concepts (Nokia Imaging SDK documentation)
- Download and add the libraries to the project (Nokia Imaging SDK documentation)
- Quick Start (Nokia Imaging SDK documentation)
- Nokia Imaging SDK API Reference (Nokia Imaging SDK documentation)
- How to (not) deadlock your UI using async-await
- Bringing async/await to the Contacts service
- Asynchronous Programming For Windows Phone 8
- IDisposable Interface (MSDN)

(C) Copyright Nokia 2014. All rights reserved.