Optimizing Imaging SDK use for rapidly changing filter parameters

Note: This is an entry in the Nokia Imaging and Big UI Wiki Competition 2013Q4.

This article explains how to use the Nokia Imaging SDK efficiently during rapid user interaction.

29 Sep 2013

Introduction

Applying image effects with the Nokia Imaging SDK is both easy and efficient. However there are use-cases where a naive application of the filters can result in unnecessary memory use and processing, which can in turn result in less-than-smooth UI behavior. One such example is when a user connects image processing code directly to the value of a slider, and generates a new version of the image on every slider event.

This article provides an an overview of filter use and "limitations", an explanation of how they can be misused (taking the slider case as an example) and provides a state machine that can be used to handle the user interaction more effectively.

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:

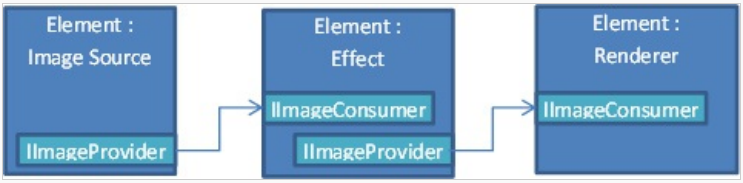
- Core concepts (Nokia Imaging SDK documentation)
- Download and add the libraries to the project (Nokia Imaging SDK documentation)
- Quick Start (Nokia Imaging SDK documentation)

Imaging SDK

To manipulate a picture, the Imaging SDK defines two interfaces :

- `IImageProvider`: Input class which provides pixels.
- `IImageComsumer`: Output class which consumes pixels provided by a `IImageProvider`.

These interfaces are used to make a pipeline between an Image source and a renderer:



Typical pipeline (Equivalent to `EditingSession` from the beta version of the Imaging SDK)

Image sources implement `IImageProvider`:

- `BitmapImageSource`: Picture is decoded in a `Bitmap` instance.
- `BufferImageSource`: Picture is an encoded file loaded in a IBuffer.
- `RandomAccessStreamImageSource`: access the encoded Picture file with a `IRandomAccessStream`.
- `StorageFileImageSource`: access the encoded Picture file with a `IStorageFile`.
- `StreamImageSource`: access the encoded Picture file with a `System.IO.Stream`.
- etc...

Note: The SDK can read Jpeg, Png, Gif , Bmp, Wbmp and Tiff files

To work with high resolution pictures, you must use a `IImageProvider` which uses the image file (the SDK will decode only the pixels it needs).

Renderers implement `IImageComsumer`:

(C) Copyright Nokia 2014. All rights reserved.

- `BitmapRenderer`: render to a `Nokia.Graphics.Imaging.Bitmap`
- `WriteableBitmapRenderer`: render to a `WriteableBitmapRenderer`
- `JpegRenderer`: encode the rendering to a JPG and return the file in an `IBuffer`

The SDK provides classes which implement the `IImageComsumer` and `IImageProvider` interface to modify pixels between an `ImageSource` and a Renderer :

- `FilterEffect`: applies a collection of `IFilter`. This collection is accessible through the `Filter` property and filters will be applied in order.
- `CustomEffectBase`: used to implement a custom filter.

Note: when you use a `FilterEffect`, the `IFilter` collection is accessible with the `Filter` property. You can easily add, remove and move filters. These filters are applied in order.

`IImageProvider` can be shared between pipelines. For example, to create a different pipeline which uses the same Image source.

Warning: The rendering process is asynchronous, and you can't reuse an `IImageProvider` until processing completes.

Unmanaged resources

The Nokia Imaging SDK is a WinPRT component coded in C++ and allocating unmanaged resources. Since C# uses a Garbage Collector, you don't know when its classes, with their unmanaged resources, will be deleted. Application memory can grow quickly if developers do not take specific action to release them promptly.

For this reason, Image sources, `FilterEffect`s and renderers implement the `IDisposable` interface. This interface indicates that you can de-allocate unmanaged resources by calling the `Dispose()` function.

To simplify `IDisposable` class use, C# provides the `using` keyword. This keyword is equivalent to `try`/`finally` where `Dispose()` is called in the `finally` section.

With using keyword

```
//define the pipeline :  source => effect => renderer
using (var source = new StreamImageSource(input))
using (var effect = new FilterEffect(source))
using (var renderer = new WriteableBitmapRenderer(effect, outputBitmap))
{
    effect.Filters = new IFilter[]{ new CreateAntiqueFilter()};
    await renderer.RenderAsync();
}//Dispose is called on source, effect and renderer.
```

Without using keyword

```
StreamImageSource source = null;
FilterEffect effect = null;
WriteableBitmapRenderer renderer = null;
try
{
    //instantiate the pipeline
    source = new StreamImageSource(input))
    effect = new FilterEffect(source))
    renderer = new WriteableBitmapRenderer(effect, outputBitmap))

    effect.Filters = new IFilter[]{ new CreateAntiqueFilter()};
    await renderer.RenderAsync();
}
```


(C) Copyright Nokia 2014. All rights reserved.

```
finally
{
    if(source != null)    source.Dispose(); //call Dispose
    source = null;

    if(effect != null)    source.Dispose(); //call Dispose
    effect = null;

    if(renderer != null)    source.Dispose(); //call Dispose
    renderer = null;
}
```

Please call `Dipose()` manually or use `using` keyword - depending on your application context. The `using` keyword is preferred if you want to apply an effect on your picture and then won't need to further re-use the Image source.

Render to an Image Control

When you want display the rendering result, it's better to use a `WriteableBitmap` which will not be scaled by the Image Control. .NET control dimensions are in logical pixels. To avoid scaling you must convert the Image control size in physical pixels and use it for your `WriteableBitmap`. To convert logical pixel dimensions to real pixel dimensions you need to get the factor between logical and physical pixels dimension. You could use the factor given by `System.Windows.Application.Current.Host.Content.ScaleFactor`. Unfortunately, Microsoft have decided that this value will be the same for **720P** and **1080p** screen. GDR3 gets the physical screen resolution with `DeviceExtendedProperties.GetValue("PhysicalScreenResolution")`). As a Windows Phone device always has the same width in logical dimension (480), it is simple to compute the correct factor and use a `WriteableBitmap` with an optimized size:

```
private double _ScreenToPixelFactor = 0;
private double ScreenToPixelFactor
{
    get
    {
        if (_ScreenToPixelFactor == 0)
        {
            try
            {
                _ScreenToPixelFactor =
((System.Windows.Size)DeviceExtendedProperties.GetValue("PhysicalScreenResolution")).Width
/ 480;
            }
            catch (Exception)
            {
                _ScreenToPixelFactor =
System.Windows.Application.Current.Host.Content.ScaleFactor / 100.0;
            }
        }
        return _ScreenToPixelFactor;
    }
}

...
var displayedBbitmap = new WriteableBitmap((int)(imageControl.ActualWidth *
ScreenToPixelFactor), (int)(imageControl.ActualHeight * ScreenToPixelFactor));
imageControl.Source = displayedBbitmap ;
```


(C) Copyright Nokia 2014. All rights reserved.

User interaction

It's possible to share an `IImageProvider` between several pipelines - note however that once rendering has started the `IImageProvider` cannot be reused until it completes. Applying image effects with the Nokia Imaging SDK is easy and efficient, so usually these limitations have little effect on app design.

There are however user interactions where a naive design can have an impact - for example if parameters can change rapidly a poor implementation will result in the creation of unnecessary pipelines for each new value.

The following sections explain this problem, and the solution, in the context of a slider controlling a filter parameter on an image. To simplify code, we use stream file picture as `StreamImageSource` input and a `WriteableBitmapRender` output. The `WriteableBitmap` size is optimized as explained in the previous section and displayed by an `Image` control.

Naive method

The `Slider` raises `ValueChanged` events when the user changes the slider position. A naive implementation would generate new images (and hence new filters) when the event is raised.

```
private void filterparam_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
    //reset stream position
    input.Seek(0, SeekOrigin.Begin);

    using (var source = new StreamImageSource(input))
    using (var effect = new FilterEffect(source))
    using (var renderer = new WriteableBitmapRenderer(effect, outputBitmap))
    {
        effect.Filters = new IFilter[]{
                                new SolarizeFilter(filterValue),
                                new TemperatureAndTintFilter(1 -  filterValue, 0.0)};

        await renderer.RenderAsync();
        outputBitmap.Invalidate();
    }
}
```

Warning: The pipeline can't be reused while renderer is processing. Therefore in this case a new pipeline is created for each new event.

Note: To manipulate a picture, we use the file stream. So we need to reset its position before each pipeline creation.

When a user moves the slider, many events are raised. With the above implementation each event results in recreation of the session and filters and a new asynchronous (parallel!) process to apply the effect to the image. Not only is generation of the parallel images not necessary, but this approach has the following implications:

- The app will use 100% of CPU and allocate a lot of unmanaged resources.
- The application GUI can be affected, resulting in "jerky" rendering of the image.
- Processing time is not constant and you don't know which are the real parameters to apply on the final displayed image.

Naive implementation code example

Sample code implementing this "naive" solution is in the **NokiaImagingFilter.cs** file.

This class is composed by three properties and one function:

- `Input`: file `stream`
- `Output`: target `Image` controls.
- `filterValue`: filter parameter. Range [0.0, 1.0]

(C) Copyright Nokia 2014. All rights reserved.

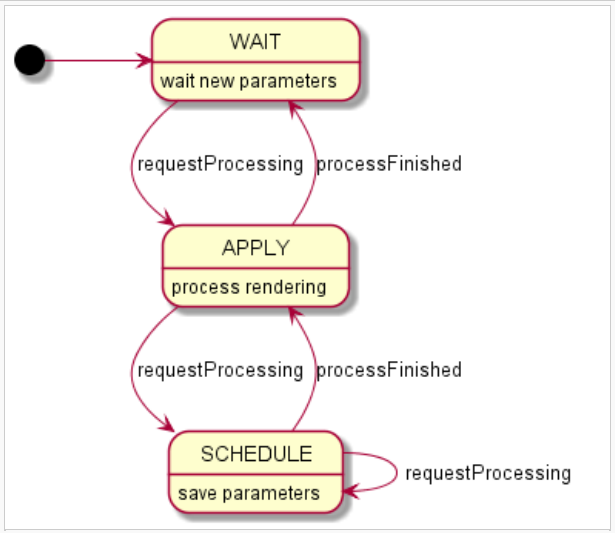
- `processRendering()`: process the rendering to a `WriteableBitmap`.

When a property is modified, `processRendering()` function is called.

Interactive State Machine

The naive method should be corrected to remove unnecessary processing, render the image display smoothly, and ensure the final image uses the correct parameters.

The approach we use to improve naive method is to create a very simple state machine.



Interactive State Machine Diagram

Note: Interactive State Machine is really simple to implement and you can easily adapt it for other context.

This State machine has three *States*:

- WAIT: wait new parameters to process a new rendering.
- APPLY: process the rendering.
- SCHEDULE: save new parameters for the new rendering.

And a *Transition* can be caused by two events :

- `requestProcessing`: parameters are updated. Request a new rendering processed.
- `processFinished`: rendering process is finished.

When the user begins to move the slider:

1. the parameter is updated => APPLY State become active => process the image using current parameter
2. if the parameter is updated and a rendering process has not finished => SCHEDULE State become active and the parameter is saved
3. when the rendering process is finished => APPLY State become active => process a new rendering

Note: If the user continues to move the slider, the state machine will loop between APPLY and SCHEDULE to avoid unnecessary processing.

Once the user finishes moving the slider :

- rendering process is finished and SCHEDULE State is active => APPLY State become active => process the image using saved parameter.
- rendering process is finished and APPLY State is active => WAIT State become active => no more process is necessary.

Note: As the SCHEDULE state saves the last parameters every time it is updated, rendering is always done using the correct/most recent parameters.

State machine sample code

Sample code implement this solution in **InteractiveNokiaImagingFilter.cs** file.

(C) Copyright Nokia 2014. All rights reserved.

The class is composed of nine properties

- `Input`: file `stream`
- `Output`: `Image` controls.
- `outputBitmap` : writeableBitmap displayed by the `Output`.
- `filterValue`: filter parameter. Range [0.0, 1.0]
- `source`: pipeline `Image`source.
- `effects`: pipeline `FilterEffect`.
- `renderer`: pipeline Renderer. Use `outputBitmap` as target.
- `solarizeFilter`: first filter applied by `effects`.
- `temperatureAndTintFilter`: second filter applied by `effects`.

And three functions:

- `requestProcessing()`: A parameter is updated.
- `processRendering()`: **process asynchronously** the rendering.
- `processFinished()`: Process rendering is finished. Called at the end of `processRendering()` function.

Pipeline is create once. `solarizeFilter` and `temperatureAndTintFilter` parameters are updated before each rendering. The `processRendering()` function is similar to the one used in the naive method described above, but with `processFinished()` called at the end and the pipeline reused.

```
async void processRendering()
{
    try
    {
        if (output != null && source != null)
        {
            //update filters parameters
            solarizeFilter.Threshold = filterValue;
            temperatureAndTintFilter.Temperature = 1.0 - 2.0 * filterValue;

            //start rendering
            await renderer.RenderAsync();
            outputBitmap.Invalidate();
        }
    }
    catch (Exception)
    {
    }
    finally
    {
        processFinished();
    }
}
```

To implement the *Interactive State Machine*, we represent :

- states with the enum `STATE`.
- the active state with the `currentState` member.

```
enum STATE
{
    WAIT,
     APPLY,
    SCHEDULE
```

 (C) Copyright Nokia 2014. All rights reserved.

```
};
 //Current State
 STATE currentState = STATE.WAIT;
```

Transitions are managed by `requestProcessing()` and `processFinished()` functions. These functions update the active `State` and call `processRendering` when active State is APPLY.

```
void requestProcessing()
{
 switch (currentState)
 {
  //State machine transition : WAIT -> APPLY
  case STATE.WAIT:
   currentState = STATE.APPLY;
   //enter in APPLY STATE => apply the filter
   processRendering();
   break;

  //State machine transition : APPLY -> SCHEDULE
  case STATE.APPLY:
   currentState = STATE.SCHEDULE;
   break;

  //State machine transition : SCHEDULE -> SCHEDULE
  case STATE.SCHEDULE:
   currentState = STATE.SCHEDULE;
   break;
 }
}
void processFinished()
{
 switch (currentState)
 {
  //State machine transition : APPLY -> WAIT.
  case STATE.APPLY:
   currentState = STATE.WAIT;
   break;
  //State machine transition : SCHEDULE -> APPLY.
  case STATE.SCHEDULE:
   currentState = STATE.APPLY;
   //enter in APPLY STATE => apply the filter
   processRendering();
   break;
 }
}
```

Note: When SCHEDULE state is active, we need to save parameters. In this implementation we use the last updated parameters. Since parameters are the class properties, we don't need to save it again.

Warning: This implementation is not thread safe as sliders event are called in UI thread. If you have a parameter which can be updated by another thread, you can use `Deployment.Current.Dispatcher.BeginInvoke()` to move parameter update to the UI Thread.

Optimizing rendering duration (optional)

(C) Copyright Nokia 2014. All rights reserved.

This section explains an optimization, where the perceived performance is improved by trading off the output resolution for reduced calculation time.

Warning: This approach may or may not be useful in your particular application or use-case.

The Imaging SDK is very optimized, but unfortunately the SDK doesn't explain its optimizations. As a result, for some use-cases output resolution is an important factor, in others it may have no effect at all.

Imaging SDK rendering duration depends on a number of different factors, including:

- Input type - encoded file or decoded buffer,
- Input resolution,
- Output resolution,
- Filter(s) used,
- Selected picture area,
- SDK cache,
- Hardware, memory, system
- etc

This section explores the effects of output resolution on the rendering time for a number of different picture input resolutions. The example uses the context of the user moving the slider - rendering an intermediate "low resolution" picture much faster. To do it, we use two `Writeablebitmap` as `Image` source:

- `bitmapHR`: full resolution bitmap
- `bitmapLR`: low resolution bitmap

Full resolution is the optimized dimension. Low resolution is simply the full resolution divided by a factor.

```
    bitmapLR = new WriteableBitmap(
        bitmapHR.PixelWidth / 2,
        bitmapHR.PixelHeight / 2
    );
```

Note: Unfortunately, when a bitmap is set as `Image` control source, a concurrent access between SDK and Image control appeared (see #Another correction)

To finish, we must know when the application must use low resolution or high resolution. If you use a slider, you can process low resolution rendering between its `ManipulationStarted` and `ManipulationCompleted` events.

```
 private void slider_ManipulationStarted(object sender,
System.Windows.Input.ManipulationStartedEventArgs e)
{
     interactiveFilter2.HRrendering = false;
}

private void  slider_ManipulationCompleted(object sender,
System.Windows.Input.ManipulationCompletedEventArgs e)
{
    interactiveFilter2.HRrendering = true;
}
```

Note: when rendering resolution change, requestProcessing is called.

As a `IImageProvider` can be shared, we create a `WriteableBitmapRendeder` for each resolution and use the same first pipeline step **Image source => FilterEffect**. When `processRendering()` is called, we can now choose between low resolution and high resolution renderer.

(C) Copyright Nokia 2014. All rights reserved.

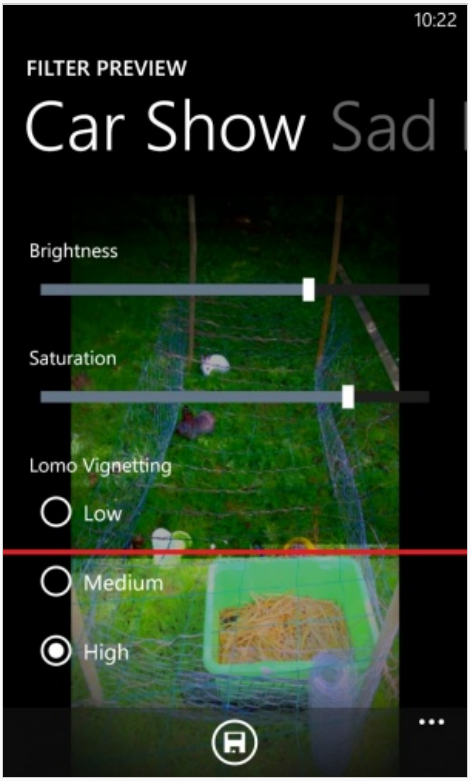
Sample code implement this solution in **InteractiveNokiaImagingFilter2.cs** file which updates **InteractiveNokiaImagingFilter.cs** with rendering time optimization.

As stated in the beginning of this section, the benefit of this optimization will depend on a number of factors:

- With an 8Mp picture rendering time decreases from 120ms to 90ms
- With a 41Mp picture, rendering time is constant 90ms.
- With another example using gestures for picture navigation with a 41Mp picture and half resolution, rendering time decreased from 500-160 ms to 180-60 ms.

Another correction

When you use only one `WriteableBitmap` as `Image` source, there is concurrent access between the Imaging SDK thread and the UI, so when the user moves the slider it is possible for the image to partially display the result from a number of slider positions. The snapshot below shows this phenomena - the top and bottom part of the displayed image (separated by a red line) are the result of different parameters.



Concurrent access phenomena when user move a slider

Using the state machine removes parallel rendering of images, but the concurrent access between SDK and Image control can still result in the Image showing the result of a number of renderings. To fix this you can use a temporary `WriteableBitmap` as the output target. After rendering, I copy this to the Image bitmap that is displayed.

```
    // process rendering to the temporary WriteableBitmap
    await renderer.RenderAsync();
    //Copy pixels
    _tmpBitmap.Pixels.CopyTo(_previewBitmap.Pixels,0);
     _previewBitmap.Invalidate(); // Force a redraw
```

Sample code

(C) Copyright Nokia 2014. All rights reserved.

Naive method

Interactive State Machine

The example app allows you to interactively view the difference between the two methods in terms of memory use and with optimized rendering duration :

1. Run application in release.
2. Click image icon bar and select a picture.
3. Select a pivot page :
   - **Naive** page use the naive method,
   - **Interactive** page use Interactive State Machine,
   - **Interactive2** page use Interactive State Machine with optimized rendering duration.
4. Move the slider.

The *Interactive State Machine* provides better interaction with the user.

Note: To increase the difference, you can test a 41 MP picture taken with Nokia Lumia 1020 or Nokia Pureview 808.

Filter Effects SDK sample

The Nokia Imaging SDK example Filter Effects can be installed from Windows Phone Store here. Since Nokia Imaging SDK 1.0, this sample has integrated the Interactive State Machine. The documentation for this example provides a basic explanation of how to handle modifying filter properties on the fly.

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:

- Reframing with gestures.
- Applying effects with user control.
- *Monsterification* is done with DirectX.

This application provide an unlimited trial version (You don't need to pay) and can be found on Windows Phone Store here

Reference

(C) Copyright Nokia 2014. All rights reserved.

- 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 Use the Nokia Imaging SDK to create a "Photo Cookbook" for Windows Phone 8 Devices
- Filter Testing app using the Imaging SDK
- 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.