Handling rapid imaging filter parameter changes using Rx

This article explains how to handle rapid changes in filter parameters and provide the latest values to the `EditingSession` to create a smooth user experience.

Introduction

In the Filter Effects sample, there is a `FilterPropertiesControl` class which is used to display the custom controls for manipulating the Filter properties. Since Filter is immutable, i.e. once created you cannot modify its properties. So once the filter is applied to an `EditingSession`, and the user changes the filter properties, a new Filter instance be created, the previous filter be removed from the `EditingSession` (by calling the **Undo** method) and then applying the new filter to the `EditingSession` . Now this update process takes some time and if the user changes the properties rapidly, the update process might not be able to keep up with the changes, resulting in the user noticing a lag or flickering of the image.

Yan, in his article Optimizing Imaging SDK use for rapidly changing filter parameters, suggested a state based solution for handling rapidly changing filter parameters. Using Rx, I have created a similar solution based on one of my earlier WPF controls, FluidStatusBar where I used a queue to keep track of the animation requests.

The Solution

In this solution too, I am using a queue to keep track of the filter parameters updated by the user. However, I am keeping only the latest filter property in the queue. Thus, each time the user changes the properties of the filter, the new properties replace the existing properties in the queue. Therefore the queue will always have Zero or One set of filter properties in it. As each update of the filter in the `EditingSession` takes a little processing time, the queue will keep track of the latest request made by the user. After the update of the filter in the `EditingSession` has completed, the queue will be queried to check if any more requests are made by the user. If yes, then the filter will be updated again. This workflow repeats until there are no more user requests.

```
public partial class MainPage : PhoneApplicationPage
{
    struct ColorVal
    {
        public double r;
        public double g;
        public double b;

        public ColorVal(double red, double green, double blue)
        {
            r = red;
            g = green;
            b = blue;
        }
    }

    EditingSession _session = null;
    bool _isBusy = false;

    Queue<ColorVal> _valueQueue = new Queue<ColorVal>();
    const int interval = 0;

    WriteableBitmap tempBmp = null;

    #region Red

    /// <summary>
    /// Red Dependency Property
    /// </summary>
    public static readonly DependencyProperty RedProperty =
        DependencyProperty.Register("Red", typeof(double), typeof(MainPage),
            new PropertyMetadata(0.0,
                new PropertyChangedCallback(OnRedChanged)));

    /// <summary>
    /// Gets or sets the Red property. This dependency property
    /// indicates the red value.
    /// </summary>
    public double Red
    {
        get { return (double)GetValue(RedProperty); }
        set { SetValue(RedProperty, value); }
    }

    /// <summary>
    /// Handles changes to the Red property.
    /// </summary>
    /// <param name="d">MainPage</param>
    /// <param name="e">DependencyProperty changed event arguments</param>
    private static void OnRedChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        MainPage page = (MainPage)d;
        double oldRed = (double)e.OldValue;
        double newRed = page.Red;
        page.OnRedChanged(oldRed, newRed);
    }

    /// <summary>
    /// Provides derived classes an opportunity to handle changes to the Red property.
    /// </summary>
    /// <param name="oldRed">Old Value</param>
    /// <param name="newRed">New Value</param>
    void OnRedChanged(double oldRed, double newRed)
    {
        UpdateFilter(Red, Green, Blue);
```


(C) Copyright Nokia 2013. All rights reserved.

```
    }

    #endregion

    ...

    #region ChosenPhoto

    /// <summary>
    /// ChosenPhoto Dependency Property
    /// </summary>
    public static readonly DependencyProperty ChosenPhotoProperty =
        DependencyProperty.Register("ChosenPhoto", typeof(BitmapSource), typeof(MainPage),
            new PropertyMetadata(null,
                new PropertyChangedCallback(OnChosenPhotoChanged)));

    /// <summary>
    /// Gets or sets the ChosenPhoto property. This dependency property
    /// indicates the photo initially chosen.
    /// </summary>
    public BitmapSource ChosenPhoto
    {
        get { return (BitmapSource)GetValue(ChosenPhotoProperty); }
        set { SetValue(ChosenPhotoProperty, value); }
    }

    /// <summary>
    /// Handles changes to the ChosenPhoto property.
    /// </summary>
    /// <param name="d">MainPage</param>
    /// <param name="e">DependencyProperty changed event arguments</param>
    private static void OnChosenPhotoChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
    {
        MainPage page = (MainPage)d;
        BitmapSource oldChosenPhoto = (BitmapSource)e.OldValue;
        BitmapSource newChosenPhoto = page.ChosenPhoto;
        page.OnChosenPhotoChanged(oldChosenPhoto, newChosenPhoto);
    }

    /// <summary>
    /// Provides derived classes an opportunity to handle changes to the ChosenPhoto property.
    /// </summary>
    /// <param name="oldChosenPhoto">Old Value</param>
    /// <param name="newChosenPhoto">New Value</param>
    void OnChosenPhotoChanged(BitmapSource oldChosenPhoto, BitmapSource newChosenPhoto)
    {
        if (_session != null)
            _session.Dispose();

        _session = new EditingSession(new WriteableBitmap(newChosenPhoto).AsBitmap());
        tempBmp = new WriteableBitmap((int)_session.Dimensions.Width, (int)_session.Dimensions.Height);
        DisplayPhoto = new WriteableBitmap(newChosenPhoto);

        Red = Green = Blue = 0.0;
    }

    #endregion

    ...

    // Constructor
    public MainPage()
    {
        InitializeComponent();

        this.DataContext = this;

        this.Loaded += MainPage_Loaded;
    }

    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        PhotoChooserTask task = new PhotoChooserTask();
        EventHandler<PhotoResult> handler = null;
        handler = (s, ea) =>
        {
            task.Completed -= handler;
            if (ea.TaskResult == TaskResult.OK)
            {
                BitmapImage imgBmp = new BitmapImage();
                imgBmp.SetSource(ea.ChosenPhoto);

                ChosenPhoto = imgBmp;

                redSlider.ValueChanged += (rs, re) =>
                {
                    Red = re.NewValue;
                };

                greenSlider.ValueChanged += (gs, ge) =>
                {
                    Green = ge.NewValue;
                };

                blueSlider.ValueChanged += (bs, be) =>
                {
                    Blue = be.NewValue;
                };
            }
        };

        task.Completed += handler;
        task.ShowCamera = false;
        task.Show();
    }

    private void OnFilterPropertyChanged(double p1, double p2, double p3)
    {
        _valueQueue.Clear();
        _valueQueue.Enqueue(new ColorVal(p1, p2, p3));

        ProcessNextQueueItem();
    }

    void ProcessNextQueueItem()
    {
        if (_isBusy)
            return;

        ColorVal val;
```


(C) Copyright Nokia 2013. All rights reserved.

```
        if (_valueQueue.Count == 0)
            return;

        val = _valueQueue.Dequeue();

        UpdateFilter(val.r, val.g, val.b);
    }

    async private void UpdateFilter(double red, double green, double blue)
    {
        if (_isBusy)
            return;

        _isBusy = true;

        if (_session != null)
        {
            {
                if (_session.CanUndo())
                {
                    // remove the previously applied value
                    _session.Undo();
                }
                _session.AddFilter(FilterFactory.CreateColorAdjustFilter(red, green, blue));
            }

            // Render to temporary bitmap to avoid concurrent access between GUI and SDK
            await _session.RenderToWriteableBitmapAsync(tempBmp);
            // Copy the pixel data to DisplayPhoto
            tempBmp.Pixels.CopyTo(DisplayPhoto.Pixels, 0);
            // request redraw
            DisplayPhoto.Invalidate();
        }

        _isBusy = false;

        ProcessNextQueueItem();
    }

    private void OnShowSettings(object sender, RoutedEventArgs e)
    {
        SettingsGrid.Visibility = Visibility.Visible;
    }

    private void OnHideSettings(object sender, RoutedEventArgs e)
    {
        SettingsGrid.Visibility = Visibility.Collapsed;
    }
}
```

In order to prevent concurrent access between Nokia Imaging SDK and GUI, in the `UpdateFilter` method, after the filter is applied, the result is rendered onto a temporary `WriteableBitmap`. After that the temporary `WriteableBitmap` is copied onto `DisplayPhoto` by using the `CopyTo` method.

This solution will keep the memory consumption low as the filter is not updated in the `EditingSession` for each property change made by the user. Instead it will be updated for the latest property change with respect to the completion of the filter update.

Source Code

Download |Source Code

Summary

Here I have provided another way of capturing the latest changes made to filter parameters and applying them to the image. However a more effective solution would be if `EditingSession` allowed changing of the applied filter parameters directly without calling Undo and reapplying the filter with new

(C) Copyright Nokia 2013. All rights reserved.

parameters.

References

- WriteableBitmapEx Library
- FluidStatusBar
- Filter Effects Sample
- Optimizing Imaging SDK use for rapidly changing filter parameters

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.