Microsoft® Silverlight™

Hands-On Lab

Silverlight 4 – Webcam in Silverlight

Contents

Overview 3
Starter Solution 4
Add a default image display 5
Exercise 1 – Pixel Shaders and Implicit Styles 7
Task 1 - Implicit Styles 7
Task 2 - Pixel Shader Effects 7
Task 3 - InvertColorEffect 11
Task 4 - PixelateEffect 11
Task 5 - SwirlEffect 12
Task 6 - RippleEffect 12
Task 7 - Stop Animations 13
Task 8 - Projection 14
Exercise 2 – Hardware Interaction 17
Task 1 - Activate / Deactivate WebCam capture 17
Task 2 – Capture an Image Snapshot 21
Task 3 – Printing 22
Conclusion 24

Overview

Several new features were introduced in Silverlight 4 that can be used to create rich media applications. In addition to media enhancements, Silverlight 4 now supports implicit styles. This means that controls will implicitly pick up the style via the **TargetType** without having to explicitly specify the Style key resource or rely on the implicit style manager. Silverlight 4 provides the highly anticipated functionality of being able to securely interact with client machine hardware devices via the browser plug-in. This includes capturing information such as audio / video streams from webcams and microphones and spooling output into a printer device.

This lab demonstrates how to use rich media features of Silverlight 4 to effectively use implicit styles, create visual effects and use a webcam. The solution consists of 2 projects, a Silverlight 4 project and an ASP.NET server host application. During this lab we will be modifying the Silverlight project only.

In exercise 1 we will apply implicit styles, a new feature in Silverlight 4 which allows a style to be applied to all elements of matching type. Following this, we will apply various pixel shader effects to an image, animating various properties of these effects. Finally we will apply a perspective transform to the canvas and animate it to yield a 3D planar effect.

During exercise 2, we will add webcam capture functionality. This involves retrieving the default visual input devices of the host environment, checking if the devices can be initialized correctly, instantiating the **CaptureSource**, hook up to devices, starting and stopping capture, and setting a **VideoBrush Source** to the **CaptureSource**. Next, we will asynchronously capture an image from the video stream. Finally, we will open a **ChildWindow** dialog and display a still of the animation and the pixel shader effects. We will implement Print functionality to print this captured still to a print device via the Silverlight plugin.

Estimated completion time for this lab is 60 minutes.

Starter Solution

Open the MagicMirror solution for this lab in Visual Studio 2010. The solution file is located at "MagicMirror\Source\StarterSolution\begin".

Set the MagicMirror.Web project as the startup project, by right clicking the project in the Solution Explorer and selecting “Set as Startup Project”.

The solution consists of 2 projects, a Silverlight 4 project and an ASP.NET server host application. During this lab we will be modifying the Silverlight project only.

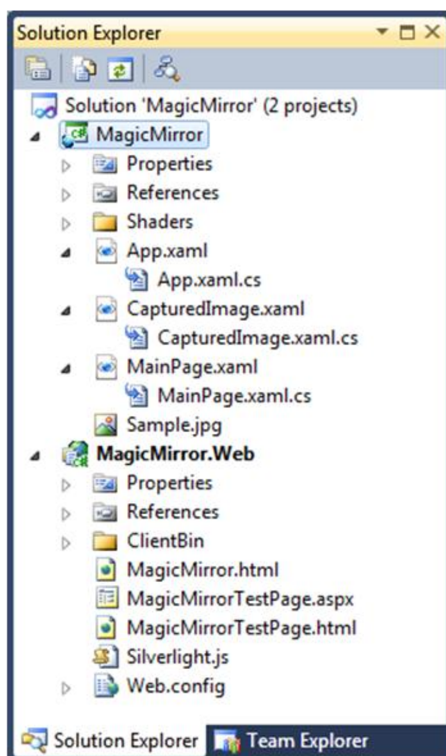

Figure 1
Project Structure

The Silverlight project contains the following items:

An App.xaml file and its associated code-behind App.xaml.cs file. The **Application Resources** dictionary declaratively defines a **Style** resource which we will use to implicitly style the Buttons within the application.

A MainPage.xaml file and its associated code-behind MainPage.xaml.cs file. This defines the MainPage **UserControl**, which is the **Application RootVisual**. In the declarative template, note the following elements:

- A **StackPanel** of **Button** controls – their event handlers are wired up, and during this lab, we will implement appropriate business logic here for controlling the application.
- A **Canvas** of border effects containing multiple **Path** elements – these paths define the background vector graphic of the application.

The code-behind contains the following stubs, which will be modified during this lab:

- Button_Click – several Buttons hook up to this event handler – it contains a switch case statement to detect the sender name and handle the appropriate action. We will use this event handler for applying Shader effects and starting animations.
- StopAnimations – this method will be called to stop any running **Storyboard** resources.
- btnRotate_Click – this event handler will toggle the running state of the **Perspective** rotation **Storyboard.**
- btnStart_Click - this event handler will enable / disable the webcam video **CaptureSource.**
- btnCapture_Click - this event handler will capture the current video frame as a bitmap for printing via a **ChildWindow** dialog.

A CaptureImage.xaml file and its associated code-behind CaptureImage.xaml.cs file. This defines the CaptureImage **ChildWindow**, which will be used to display and print a video frame captured as a bitmap. Its buttons are labeled "Close" and "Print". During Exercise 2 Task 3, we will implement logic in the print button event handler to print a page from Silverlight.

Sample.png image file – the build action of this included file is set to resource, so that it is compiled into the project .xap file.

A Shaders solution folder – this contains 4 pre-constructed pixel shader effects: InvertColorEffect, PixelateEffect, RippleEffect, and SwirlEffect. Pixel shader effects will be discussed in Exercise 1 Task 2.

Add a default image display

1. Open MainPage.xaml file
2. Add an **ImageBrush** UserControl Resource – pointing to a .png file resource.

   XAML

   ```
   <ImageBrush x:Name="imageBrush" ImageSource="Sample.png" Stretch="Fill" />
   ```

3. Locate the final **Path** element (the one at the bottom of the XAML file) with an x:Name of "TO_FILL". Set its **Fill** attribute to bind to the ImageBrush resource. We will dynamically change the Fill to display a **VisualBrush** wrapping the Webcam capture during Exercise 2 Task 1.

   XAML

```
Fill="{Binding Source={StaticResource imageBrush}}"
```

4. Run the application. You should see the background vector graphics, a horizontal StackPanel of Buttons and an image which is filling the topmost path.

Exercise 1 – Pixel Shaders and Implicit Styles

During this exercise, we will apply implicit styles, a new feature in Silverlight 4 which allows a style to be applied to all elements of matching type. Following this, we will apply pixel shader effects to an image and animate various properties of these effects. These custom effects include Ripple, Invert, Swirl and Pixelate. Finally we will apply a perspective transform to the canvas and animate it to yield a 3D planar effect.

Task 1 - Implicit Styles

In Silverlight 4 you can create Implicit Styles in addition to Explicit Styles. This new feature allows programmers to override default styles for common and third party controls with their own style, without the need to use explicit keys. In this exercise we will use supplied style and change the default look and feel of the application.

1. In App.xaml, in the Application resources section, locate the Style element. Remove the **x:Key** attribute and run the application.

   XAML

   ```
   <Style TargetType="Button">
       …
   ```

   > Notice how the Button controls implicitly pick up the style via its **TargetType**, without having to explicitly specify the Style key resource, or rely on the implicit style manager. The style contains an associated **ControlTemplate**, which threads the Button contents through a **ContentPresenter**.

2. Open MainPage.xaml, select the “Swirl” Button and explicitly set its Style to null, to opt out of the explicit style.

   XAML

   ```
   <Button Content="Swirl" Click="Button_Click" Style="{x:Null}"/>
   ```

3. Run the application again – notice that the particular button is no longer picking up the style.

Task 2 - Pixel Shader Effects

Silverlight 3 introduced Pixel Shader Effects, including the ability to add custom effects. Silverlight 3 supports 2 native effects – the blur effect and the drop shadow effect. Every **UIElement** has an **Effect** property. An effect outputs a modification of the pixel that is at the same location in the input texture as

its destination in the output texture, manipulate color but not position. The Effect class contains an **EffectMapping** method, which must be overridden by derived methods to return a **GeneralTransform** that takes a position and returns the post-effect corresponding position.

In order to create a custom effect, use the effects compiler (fxc.exe) from the DirectX SDK to compile a high level shading language (HLSL) **.fx** file into a **.ps** file, which is included in the Silverlight project, with its Build Action set to Resource. A .NET class must be created to represent the custom effect, which derives from **ShaderEffect** and provides a default constructor to initialize the effect.

The WPF Pixel Shader Effects library is available from http://wpffx.codeplex.com/ and can be adapted to work with Silverlight. Pixel Shader Effects are exposed via the **System.Windows.Media.Effects** namespace.

Open the Shaders solution folder. Note that there are 8 items in the folder: 4 *.ps files and 4 matching class files. These comprise 4 pixel shader effects: InvertColorEffect, RippleEffect, InvertEffect and SwirlEffect.

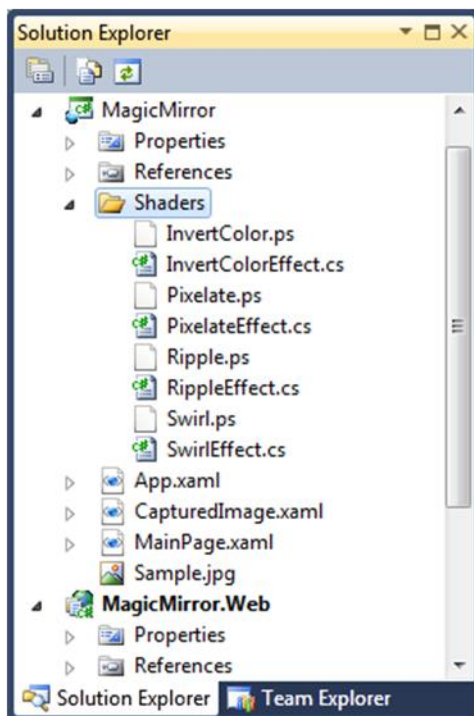

Figure 2

Pixel Shares in Project Structure

A **ShaderEffect** class provides a custom bitmap effect using a **PixelShader**. Each custom Pixel Shader Effect inherits from **ShaderEffect** and loads a shader model 2 pixel shader. Dependency properties declared in this class are mapped to registers defined in an associated *.ps file. The constructor points to the *.ps file as a relative Uri and creates an instance of the PixelShader. The **UpdateShaderValue**

method notifies the effect that the shader constant or sampler corresponding to the specified dependency property should be updated.

1. Open the InvertColorEffect and examine the class contents.

C#

```
namespace MagicMirror.ShaderEffectLibrary
{
    using System;
    using System.Windows;
    using System.Windows.Media;
    using System.Windows.Media.Effects;
    using System.Diagnostics;


    public class InvertColorEffect : ShaderEffect
    {
        #region Dependency Properties

        public static readonly DependencyProperty InputProperty =

             ShaderEffect.RegisterPixelShaderSamplerProperty("Input",
             typeof(InvertColorEffect), 0);

        #endregion

        #region Member Data

        private static PixelShader pixelShader;

        #endregion

        #region Constructors

        static InvertColorEffect()
        {
            pixelShader = new PixelShader();
            pixelShader.UriSource = new
                          Uri("/MagicMirror;component/Shaders/InvertColor.ps",
                   UriKind.Relative);
        }

        public InvertColorEffect()
        {
            this.PixelShader = pixelShader;

            UpdateShaderValue(InputProperty);
```

```
        }

        #endregion

        public Brush Input
        {
            get { return (Brush)GetValue(InputProperty); }
            set { SetValue(InputProperty, value); }
        }
    }
}
```

2. Open MainPage.xaml.cs and locate the Button_Click event handler. It contains a **switch** statement for determining the selected button. We are going to fill in the body of each case, in order to set the pixel shader effect applied to the **Path Fill** image, and to start a storyboard which will animate tweening of appropriate effect properties.

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  switch ((sender as Button).Content.ToString())
  {
    case "Pixelate":
      //TODO: start storyboard and set Path's effect
      break;
    case "Swirl":
      //TODO: start storyboard and set Path's effect
      break;
    case "Invert":
      //TODO: start storyboard and set Path's effect
      break;
    case "Ripple":
      //TODO: start storyboard and set Path's effect
      break;
    default:
      TO_FILL.Effect = null;
      break;
  }
}
```

3. In MainPage.xaml, add a XAML namespace mapping for the .NET namespace containing the custom shader effects:

XAML

```
xmlns:shaders="clr-namespace:MagicMirror.ShaderEffectLibrary"
```

4. We are now ready to utilize the custom pixel shader effects within the application.

Task 3 - InvertColorEffect

1. In MainPage.xaml, in "UserControl.Resources" section, declaratively add the InvertColorEffect, giving it a name. Save the file.

XAML

```
<shaders:InvertColorEffect x:Name="effectInvert" />
```

2. Open MainPage.xaml.cs and locate the Button_Click event handler. In the Invert case, set the Path's Effect to the InvertColorEffect.

C#

```
case "Invert":
      TO_FILL.Effect = effectInvert;
      break;
```

3. Run the application and click the Invert button. Notice that the image colors are inverted.

Task 4 - PixelateEffect

1. In MainPage.xaml, in "UserControl.Resources" section, declaratively add the PixelateEffect, giving it a name, and values for its **HorizontalPixelCount** and **VerticalPixelCount** properties. Save the file.

XAML

```
<shaders:PixelateEffect x:Name="effectPixelate" HorizontalPixelCounts="48"
      VerticalPixelCounts="48"/>
```

2. Create a Storyboard with 2 animations for animating these properties.

XAML

```
<Storyboard x:Name="sbPixelate" Storyboard.TargetName="effectPixelate"
             AutoReverse="True" RepeatBehavior="Forever">
    <DoubleAnimation Duration="00:00:04"
             Storyboard.TargetProperty="HorizontalPixelCounts"
             From="48" To="256" />
    <DoubleAnimation Duration="00:00:04"
      Storyboard.TargetProperty="VerticalPixelCounts"
      From="48" To="256" />
</Storyboard>
```

3. Open MainPage.xaml.cs and locate the Button_Click event handler. In the Pixelate case, set the Path's Effect to the PixelateEffect, and start the Storyboard animation.

C#

```
case "Pixelate":
      sbPixelate.Begin();
```

```
        TO_FILL.Effect = effectPixelate;
        break;
```

4. Run the application and click the Pixelate button. Notice that the image appears to be pixilated in an animated fashion.

Task 5 - SwirlEffect

1. In MainPage.xaml, in "UserControl.Resources" section, declaratively add the SwirlEffect, giving it a name. Save the file.

C#

```
<shaders:SwirlEffect x:Name="effectSwirl" />
```

2. Create a Storyboard with an animation for animating the Swirl.Factor property.

C#

```
<Storyboard x:Name="sbSwirl" Storyboard.TargetName="effectSwirl"
                   RepeatBehavior="Forever" AutoReverse="True">
    <DoubleAnimation Duration="00:00:02" Storyboard.TargetProperty="Factor"
                   From="-2" To="2" />
</Storyboard>
```

3. Open MainPage.xaml.cs and locate the Button_Click event handler. In the Swirl case, set the Path's Effect to the SwirlEffect, and start the Storyboard animation.

C#

```
case "Swirl":
        sbSwirl.Begin();
        TO_FILL.Effect = effectSwirl;
        break;
```

Task 6 - RippleEffect

1. In MainPage.xaml, in "UserControl.Resources" section, declaratively add the RippleEffect, giving it a name. Save the file.

XAML

```
<shaders:RippleEffect x:Name="effectRipple" />
```

2. Create a Storyboard with an animation for animating the RippleEffect.Phase property.

XAML

```
<Storyboard x:Name="sbRipple" Storyboard.TargetName="effectRipple"
                   Storyboard.TargetProperty="Phase">
    <DoubleAnimation From="30.0" To="0" Duration="00:00:10"
             RepeatBehavior="Forever" />
</Storyboard>
```

3. Add a **MouseMove** event handler to the canvas named “borderEffects” and create the corresponding event handler for the event:

XAML

```
MouseMove="borderEffects_MouseMove"
```

4. On the corresponding code behind page, set the RippleEffect.Center property according to MouseMoveEventArgs.GetPosition, which returns the (x,y) coordinate position (as a Point struct) evaluated against the supplied Canvas UIElement.

C#

```
private void borderEffects_MouseMove(object sender, MouseEventArgs e)
{
  Point mousePt = e.GetPosition(this);

  effectRipple.Center = new Point(mousePt.X / this.ActualWidth,
                                  mousePt.Y / this.ActualHeight);
}
```

5. In **MainPage.xaml.cs** locate the **Button_Click** event handler. In the Ripple case, set the Path’s Effect to the RippleEffect, and start the Storyboard animation.

C#

```
case "Ripple":
      sbRipple.Begin();
      TO_FILL.Effect = effectRipple;
      break;
```

6. Run the application and click the Ripple button**.**

7. Move the mouse over the image and observe the ripple effect (repeated by the storyboard) and that its center is set according to the mouse position.

Task 7 - Stop Animations

1. Open MainPage.xaml.cs.

2. Add the following code into the stopAnimations() method:

C#

```
private void stopAnimations()
{
  sbPixelate.Stop();
  sbRipple.Stop();
  sbSwirl.Stop();
}
```

3. At the end of this task, the complete Button_Click method should look like this:

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  stopAnimations();
  switch ((sender as Button).Content.ToString())
  {
    case "Pixelate":
      sbPixelate.Begin();
      TO_FILL.Effect = effectPixelate;
      break;
    case "Swirl":
      sbSwirl.Begin();
      TO_FILL.Effect = effectSwirl;
      break;
    case "Invert":
      TO_FILL.Effect = effectInvert;
      break;
    case "Ripple":
      sbRipple.Begin();
      TO_FILL.Effect = effectRipple;
      break;
             default:
             TO_FILL.Effect = null;
      break;
  }
}
```

4. At the end of this exercise we have pixel shader effects applied to the picture in the frame. We just learned how to use the provided pixel shader effects in order to provide the end user with live picture processing functionality and enrich the application's UI.

Task 8 - Projection

Silverlight 3 introduced a **Projection** property on **UIElement** which can be set to a **PlaneProjection**, an implementation of a 3D **PerspectiveTransform**. The UIElement that is projected into this 3D scene is interactive even if projected. Both front and back are interactive. The transforms that **PlaneProjection** applies to its **UIElement** are:

- A TranslateTransform exposed via LocalOffsetX, LocalOffsetY, LocalOffsetZ in PlaneProjection.
- A set of RotateTransform represented by the CenterOfRotationX, CenterOfRotationY and CenterOfRotationZ and the RotationX, RotationY and RotationZ properties.
- A TranslateTransform(3D), exposed via GlobalOffsetX, GlobalOffsetY, GlobalOffsetZ.

In this exercise we will apply plane projection for our application.

1. In MainPage.xaml , Add a **Canvas.Projection** then create a **PlaneProjection** and give it a name as shown next:

XAML

```
<Canvas x:Name="borderEffects" MouseMove="borderEffect_MouseMove">
    <Canvas.Projection>
        <PlaneProjection x:Name="borderProjection"/>
    </Canvas.Projection>
    …
```

2. Create a Storyboard that targets the **PlaneProjection**. Use this to animate the following 3 properties: **RotationX, RotationY,** and **RotationZ.**

XAML

```
<Storyboard x:Name="sbPerspective" Storyboard.TargetName="borderProjection">
    <DoubleAnimation AutoReverse="False" RepeatBehavior="Forever"
                     From="0" To="360" Duration="00:00:05"
                            Storyboard.TargetProperty="RotationX"/>
    <DoubleAnimation AutoReverse="False" RepeatBehavior="Forever"
                     From="0" To="360" Duration="00:00:05"
                            Storyboard.TargetProperty="RotationY"/>
    <DoubleAnimation AutoReverse="False" RepeatBehavior="Forever"
                     From="0" To="360" Duration="00:00:05"
                            Storyboard.TargetProperty="RotationZ"/>
</Storyboard>
```

3. In MainPage.xaml.cs , locate the btnRotate event handler and implement logic to Start / stop the storyboard.

C#

```
private void btnRotate_Click(object sender, RoutedEventArgs e)
{
    if (btnRotate.Content.ToString() == "Rotate")
    {
        btnRotate.Content = "Stop";

        sbPerspective.Begin();
    }
    else
    {
        btnRotate.Content = "Rotate";

        sbPerspective.Stop();
    }
}
```

4. Run the application. Select a pixel shader effect and click the Rotate button. The perspective rotation starts swirling around.

5. Set the Canvas.CacheMode to BitmapCache.

XAML

```
<Canvas x:Name="borderEffects" MouseMove="borderEffects_MouseMove"
  CacheMode="BitmapCache">
```

6. Run the application again. Select a pixel shader effect and click the Rotate button. The perspective rotation starts swirling around.

Exercise 2 – Hardware Interaction

During exercise 2, we will activate webcam capture. This will involve retrieving the default visual input devices of the host environment, checking if the devices can be initialized correctly, instantiating the **CaptureSource**, hook up to devices, starting and stopping capture, and setting a **VideoBrush Source** to the **CaptureSource**. Next, we will asynchronously capture an image from the video stream. Finally, we will open a **ChildWindow** dialog and display a still of the animation and the pixel shader effects. We will implement Print functionality to print this captured still to a print device via the Silverlight plugin.

Task 1 - Activate / Deactivate WebCam capture

Silverlight 4 has the ability to capture hardware device sources, including video sources such as webcams and audio sources such as microphones. In this task we will activate webcam capture – this will involve retrieving the default devices, checking if the devices can be initialized correctly, instantiating the **CaptureSource**, hook up to devices, starting and stopping capture and setting a **VideoBrush Source** to the **CaptureSource**.

The **System.Windows.Media** namespace exposes several classes for hardware capture. The core class is **CaptureSource**, which points to specific devices and starts and stops capture. The **CaptureDeviceConfiguration** class provides the **CaptureSource** with the configured **VideoCaptureDevice** and **AudioCaptureDevice.**

A **VideoBrush** paints an area with video content – set its source property to point to the **CaptureSource**.

1. Run the application and right click on the Plugin surface and click the Silverlight context menu to open the configuration dialog and see the Capture Device Configuration settings. This allows you to choose the default audio and video data source devices.

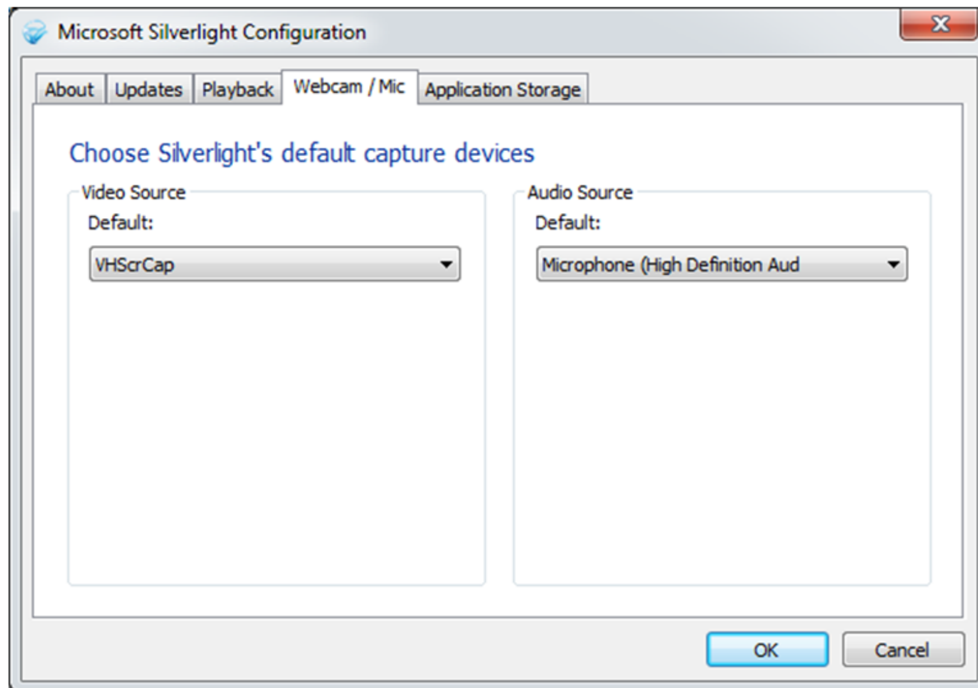

Figure 3

Webcam/Mic Tab in Silverlight Configuration Screen

2. In MainPage.xaml.cs, Declare a **CaptureSource** object instance – this is the core object that we will use to interact with device capture.

C#

```
CaptureSource cs = null;
```

3. Add code to the **btnStart_Click** event handle to toggle the button text according to state, and set the "Webcam off" state **Fill** of the **Path** to an **ImageBrush** retrieved from the UserControl's **Resources** dictionary.

C#

```
if (btnStart.Content.ToString() == "Webcam On")
{
       btnStart.Content = "Webcam Off";
       // TODO: Check devices using CaptureDeviceConfiguration object

}
else
{
       btnStart.Content = "Webcam On";

       TO_FILL.Fill = this.Resources["imageBrush"] as ImageBrush;
}
```

4. Next, add the following code into btnStart_Click to check if the devices can be accessed and ask the user if it is OK to use the devices using the **CaptureDeviceConfiguration** object. This lab only uses the webcam, however audio devices can also be used. Add the code within the CaptureDeviceConfiguration "if" statement.

C#

```
if (CaptureDeviceConfiguration.AllowedDeviceAccess ||
       CaptureDeviceConfiguration.RequestDeviceAccess())
{
      VideoCaptureDevice vcd =
        CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
      if (null != vcd) //Only check for webcam for this lab
      {

      }
      else
        MessageBox.Show("Error initializing Webcam or Mic.\nPlease install
                           device drivers and try again");
}
```

5. Instantiate the **CaptureSource**, set the video device retrieved from **CaptureDeviceConfiguration** and **Start** the capture process. Add the following code into the **if (null != vcd)** block within btnStart_Click:

C#

```
cs = new CaptureSource();
cs.VideoCaptureDevice = vcd;
cs.Start();
```

6. Create a **VideoBrush** and set its source to the **CaptureSource**. Set the **Path** in the UI to **Fill** with the VideoBrush when the webcam is toggled on.

C#

```
VideoBrush videoBrush = new VideoBrush();
videoBrush.Stretch = Stretch.Uniform;
videoBrush.SetSource(cs);
TO_FILL.Fill = videoBrush;
```

7. If the button is toggled off and the capture source was previously initialized, stop the capture by adding the following code into the last **else** block within btnStart_Click:

C#

```
if (null != cs)
       cs.Stop();
```

8. The complete event handler should look as follows:

C#

```
private void btnStart_Click(object sender, RoutedEventArgs e)
{
  if (btnStart.Content.ToString() == "Webcam On")
  {
    btnStart.Content = "Webcam Off";

    if (CaptureDeviceConfiguration.AllowedDeviceAccess ||
              CaptureDeviceConfiguration.RequestDeviceAccess())
    {
      VideoCaptureDevice vcd =
        CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
      if (null != vcd)
      {
        cs = new CaptureSource();
        cs.VideoCaptureDevice = vcd;
        cs.Start();

        VideoBrush videoBrush = new VideoBrush();
        videoBrush.Stretch = Stretch.Uniform;
        videoBrush.SetSource(cs);
        TO_FILL.Fill = videoBrush;
      }
      else
        MessageBox.Show("Error initializing Webcam or Mic.\nPlease install
                         device drivers and try again");
    }
  }
  else
  {
    btnStart.Content = "Webcam On";

    if (null != cs)
      cs.Stop();

    TO_FILL.Fill = this.Resources["imageBrush"] as ImageBrush;
  }
}
```

9. Run the application. Click the Button labeled “Webcam On”. A permission elevation request dialog is displayed. Click yes to proceed.

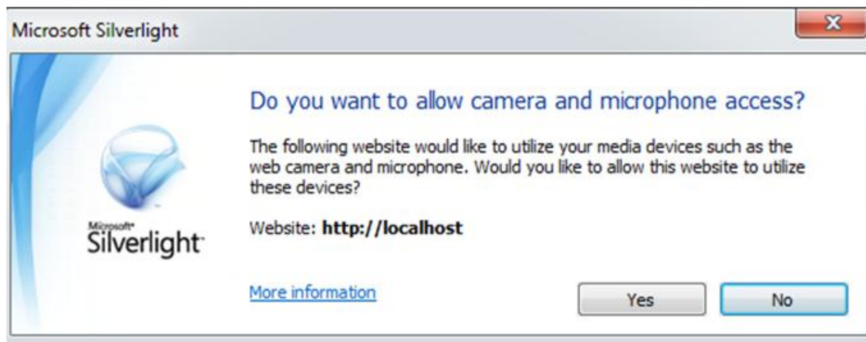

Figure 4

Webcam/Mic Usage Approval Dialog

10. Notice that the Path is filled with real-time video captured from the Webcam or emulator!

Task 2 – Capture an Image Snapshot

Now that we can enable webcam capture, we can capture an image snapshot from the video.

1. Add an image element to CapturedImage.xaml:

XAML

```
<Image x:Name="img" Stretch="Uniform"/>
```

2. In MainPage.xaml.cs, check with the **CaptureSource State** that the Webcam is actually running and capturing a video stream – check this against the **CaptureState** enum – **Failed, Started, Stopped.** Asynchronously capture an image from the video stream via **CaptureSource's CaptureImageAsync()** method.

C#

```
private void btnCapture_Click(object sender, RoutedEventArgs e)
{
    if (null != cs)
    {
        if (cs.State == CaptureState.Started)
        {
            cs.CaptureImageCompleted +=
              new EventHandler<CaptureImageCompletedEventArgs>(
                  cs_CaptureImageCompleted);
            cs.CaptureImageAsync();
        }
        else
        {
            MessageBox.Show("Start capture from Webcam and try again!");
        }
    }
    else
```

```
        MessageBox.Show("Start capture from Webcam and try again!");
}
```

3. In the cs_CaptureImageCompleted callback, set the source for the image in the **ChildWindow**. Indicate that the child window doesn't have a Close button. Open the Child window in a modeless manner to display the image.

C#

```
void cs_CaptureImageCompleted(object sender,
  CaptureImageCompletedEventArgs e)
{
  // initialize the child window
  CapturedImage ci = new CapturedImage();
  ci.HasCloseButton = false;
  ci.img.Source = e.Result;

  ci.Show();
  cs.CaptureImageCompleted -= new
    EventHandler<CaptureImageCompletedEventArgs>(cs_CaptureImageCompleted);

}
```

4. A **WritableBitmap** provides a BitmapSource that can be written to and updated. Run the application. Start animating the perspective transform and set a pixel shader. Notice how the capture Button opens a **ChildWindow** dialog and displays a still of the animation and the pixel shader effects. Click Close to close the **ChildWindow**. You can see that this is very visually impressive functionality.

Task 3 – Printing

To complete hardware interaction picture, Silverlight 4 now comes with printer output support. This is provided via the **System.Windows.Printing** namespace, using the core class **PrintDocument**, which provides a print dialog and pages for printing from a Silverlight application.

In the CapturedImage.xaml.cs **ChildWindow** code behind, find the PrintButton_Click event handler. Instantiate a **PrintDocument** , attach an event handler for its PrintPage event, and set the PageVisual to print. Finally, call the Print method.

The **PrintDocument.PrintPage** event – occurs when the page is printing. It has a **PrintPageEventArgs** with the following properties: **PageVisual** – set the **UIElement** to print (in this case the Image element) , **HasMorePages** – a Boolean value – use to get or set whether there are more pages to print.

PrintDocument.Print() – starts the printing process by opening the print dialog.

1. Add the following to the Print button event handler:

C#

```
PrintDocument pd = new PrintDocument();
pd.PrintPage += (s, args) =>
{
    args.PageVisual = img;
    args.HasMorePages = false;
};

pd.Print("SilverlightPrintJob");
```

2. Run the application. Start animating the perspective transform and set a pixel shader. Click the capture Button to open a **ChildWindow** dialog and display a still of the animation and the pixel shader effects. Click Print to print this captured still – The Silverlight plugin present you with a standard windows print dialog for selecting the output device, satisfying a core business case scenario.

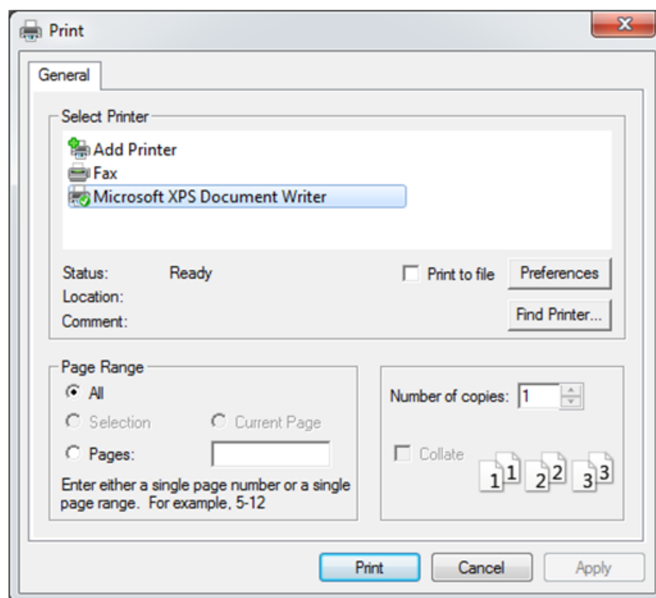

Figure 5
Print Dialog

Conclusion

During exercise 1, we applied implicit styles, a new feature in Silverlight 4 which allows a style to be applied to all elements of matching type. Following this, we applied various pixel shader effects to an image, animating various properties of these effects. Finally we applied a perspective transform to the canvas and animated it to yield a 3D planar effect.

During exercise 2, we activated webcam capture. This involved retrieving the default visual input devices of the host environment, checking if the devices can be initialized correctly, instantiating the **CaptureSource**, hook up to devices, starting and stopping capture, and setting a **VideoBrush Source** to the **CaptureSource**. Next, we asynchronously captured an image from the video stream. Finally, we opened a **ChildWindow** dialog and displayed a still of the animation and the pixel shader effects. We implemented Print functionality to print this captured still to a print device via the Silverlight plugin.