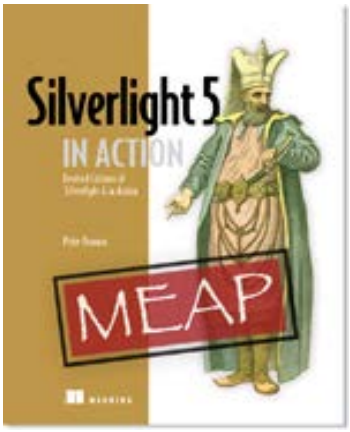
MANNING PUBLICATIONS



Silverlight 5 in Action
By Pete Brown

Effects in Silverlight come in two primary forms: built-in effects, implemented in the native Silverlight hardware-accelerated runtime code; and pixel shaders, implemented by folks like us using a combination of managed code and High Level Shader Language (HLSL) and run in software. In this article, based on chapter 23 of Silverlight 5 in Action, author Pete Brown discusses both types of effects.

Silverlight Effects

Much as is the case with animation, the subtle and appropriate use of effects can make the difference between a UI that just sits there and one that really pops, drawing your eye to information that's important to you.

Effects in Silverlight come in two primary forms: *built-in effects*, implemented in the native Silverlight hardware-accelerated runtime code; and *pixel shaders*, implemented by folks like us using a combination of managed code and High Level Shader Language (HLSL) and run in software. The former allows for maximum performance for common effects such as blur and shadows. The latter provides a lot of flexibility to allow us to provide our own effects, while not breaking out of the sandbox.

In this article, we'll cover both types of effects. We'll start with how to use the built-in effects and follow that up with a primer on creating your own pixel shader effects.

Using built-in effects

Silverlight has two built-in effects: blur and drop shadow. The effects may be used on any element or group of elements in the visual tree.

Elements that have effects applied remain as interactive as they did prior to the effect. Although it may be hard to read the text in a blurred-out `TextBox`, the `TextBox` is still fully functional.

Blur effect

The *blur effect* in Silverlight, implemented through the `BlurEffect` class, provides a way to shift an element or group of elements out of focus, as though you were looking at it through frosted glass or a bad lens.

Blur has only one property of interest: `Radius`. The `Radius` property controls how large an area is sampled when the blur is run: the larger the radius, the blurrier the result. Note that the larger the radius, the more computations required to achieve the blur—a potential performance consideration, especially if a large area or animation is involved.

Listing 1 shows how to use the `BlurEffect` on a group of UI elements in a `StackPanel`.

Listing 1 A blur with a 4-pixel radius

Result:

XAML:

```
<StackPanel x:Name="Elements" Margin="10">
  <TextBlock Text="Hello World" Margin="10" />
  <TextBox Text="This is a textbox" Margin="10" />
  <Button x:Name="Button" Content="Button" Margin="10"/>
  <StackPanel.Effect>                              #A
    <BlurEffect Radius="4" />
  </StackPanel.Effect>
</StackPanel>
```

#A Effect on StackPanel

In listing 1, the blur effect is applied to the entire `StackPanel` containing all the UI elements. The net result is to blur everything inside that container. You can also apply a blur to individual elements, of course. The effect is attached to the `StackPanel` using the `Effect` property. The `Effect` property can have only one effect at any point in time. If you want multiple effects on a single element, you need to use nested panels or borders and apply the effects one per panel/border.

The blur effect is useful when combined with things such as a pop-up modal window. In that case, a slight blur of the page contents helps drive home the fact that the pop-up is modal and demands all of your attention.

The second built-in effect is the drop shadow.

Drop shadow effect

The *drop shadow effect* is one of those effects that's best used in moderation and used subtly when used at all. Not only is there a performance and rendering quality concern, but, aesthetically, those of us who aren't designers tend to use bold shadows more often than looks good in an application.

The `DropShadowEffect` class has several knobs you can use to fine-tune the effect. Table 1 shows the five properties that alter the appearance of the effect.

Table 1 Important `DropShadowEffect` properties

Property	Description
`Color`	Specifies the color of the shadow. Default is `Black`.
`ShadowDepth`	Distance in pixels to displace the shadow relative to the element the effect is applied to. Default is 5 pixels.
`Direction`	An angle in degrees from 0 to 360 (counterclockwise), indicating where the shadow lies relative to the element the effect is applied to. The default is 315, which places the shadow in the lower-right corner.
`BlurRadius`	Controls the blurriness of the shadow. A double value between 0 and 1,

with 1 being the softest. Default is 0.5.

`Opacity`	Specifies how opaque the shadow is. A double value between 0 and 1, with 1 being fully opaque. Default is 1.

When playing with shadows, I've found it more aesthetically pleasing to have the `ShadowDepth` be 0 or close to 0, the `Opacity` set to some value around 0.5 or so, and the `BlurRadius` set to a value that spreads out the effect—10 usually works well. That gives you a light shadow that bleeds around all the edges. Listing 2 shows these settings in use in the effect.

Listing 2 A subtle drop shadow

Result:

XAML:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid Background="White" Width="180" Margin="25">      #1
    <StackPanel x:Name="Elements" Margin="10">
      <TextBlock Text="Hello World"
                 Margin="10" />
      <TextBox Text="This is a textbox"
               Margin="10" />
      <Button x:Name="Button" Content="Button"
              Margin="10"/>
    </StackPanel>
    <Grid.Effect>
      <DropShadowEffect BlurRadius="10"
                        Opacity="0.5"
                        ShadowDepth="1" />
    </Grid.Effect>
  </Grid>
</Grid>
```

Note that the effect is applied to a grid with an opaque background (#1). If the grid had a transparent background, the effect would be applied individually to each of the items inside the grid.

As described before the listing, this example uses a large blur radius, 50 percent opacity, and a shadow depth of only 1 pixel. This provides a more pleasing and subtle effect than the default shadow appearance. Compare that to figure 1, with the properties all left at their default values.

Figure 1 The default appearance of the `DropShadowEffect`

Most people would find the default appearance a bit jarring, or at least a little outdated. Fortunately, the Silverlight team gave us all the tweaks we need to be able to make the shadow look better.

In addition to these designer-type recommendations, you should keep a few other things in mind when using effects.

Tricks and considerations

The built-in effects perform well, but they'll tax your system resources if you apply them to a really large area and/or animate any of the values on the effect. For example, one thing I did early on was animate the background blur from 0 to 5 when displaying a new dialog. It worked, but it was a processing hog.

In addition to the processing time, another consideration is the quality drop in the result. Any elements with an effect applied to them are rendered out to a bitmap. That means you automatically lose ClearType font rendering and fall back to grayscale rendering. One way to get around this is to apply the effect to a shape of the same size that sits behind the elements. Listing 3 shows how to use a rectangle behind the grid to ensure that the grid contents stay at top rendering quality.

Listing 3 Applying the drop shadow to a background `Rectangle`

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid Width="180" Margin="25">
    <Rectangle Fill="White">                                #A
      <Rectangle.Effect>
        <DropShadowEffect BlurRadius="10"
                          Opacity="0.5"
                          ShadowDepth="1" />
      </Rectangle.Effect>
    </Rectangle>
    <StackPanel x:Name="Elements" Margin="10">
      <TextBlock Text="Hello World" Margin="10" />
      <TextBox Text="This is a textbox" Margin="10" />
      <Button x:Name="Button" Content="Button"
                    Margin="10"/>
    </StackPanel>
  </Grid>
</Grid>
```

#A Background Rectangle

This example removes the effect from the grid and places it on a background rectangle sitting behind the elements. Because the rectangle isn't a parent of the elements, the effect isn't applied to them. The only thing that's rasterized in this example is the `Rectangle`. The text retains ClearType font rendering.

The Silverlight team may add more effects over time. Requests include true multipass effects such as glow. In the meantime, it's possible to create your own single-pass effects using a little Silverlight code and the shader language.

Creating custom pixel shaders

Most people who know about pixel shaders have run across them in game development. Games and various types of shaders have gone hand in hand because video cards became powerful enough to offload most or all of the shader calculation and logic. Most work done with pixel shaders is performed using the DirectX SDK and optionally XNA.

WPF also supports pixel shaders. Entire libraries of transitions and effects are available on CodePlex, all built using hardware-accelerated shaders.

Pixel shaders in Silverlight are a simplified form of the full pixel shaders used in games or in WPF. For security reasons, the shaders are all run in software and currently support only Pixel Shader level 2. By not running them in hardware, Silverlight can sandbox the code and avoid someone running malicious code on your video card. But as technology progresses, the Silverlight team will likely consider allowing the shaders to run on hardware in selected scenarios.

How pixel shaders work

Pixel shaders perform per-pixel processing on input. That input can be anything you see in the visual tree in Silverlight, including images, video, and controls. Pixel shaders in Silverlight are created using two main files. The first is a .NET class that's used to wrap the shader functionality and expose it to the rest of Silverlight. The second is the pixel shader itself, written in HLSL as an .fx file and compiled into a .ps file as a resource in the Silverlight project.

Pixel shaders are written in HLSL, a C-like language optimized for pixel processing. The language is geared toward running on video card hardware, so you have to deal with things such as registers, fixed numbers of variables, and limitations on overall complexity. In some of those ways, it's like working in assembly language. You can find a reference on HLSL syntax on MSDN at http://bit.ly/HLSLReference.

Pixel shaders in Silverlight are software-rendered, but are parallelized. Although they don't take specific advantage of capabilities of video hardware, they're executed using the CPU's fast SSE instruction set.

Silverlight supports the `ps_2_0` profile of the Shader Model 2 specification. A *shader profile* is the target for compiling a shader, whereas a *shader model* is a specification for capabilities of the shader. You'll need to understand this when looking at existing shader implementations to port to Silverlight or learning about HLSL syntax. In the case of Shader Model 2, the limitation you're likely to hit is the 96-instruction limit. That limit is broken down into 64 arithmetic instructions and 32 texture-sample instructions. The 64 arithmetic instruction limit will almost certainly be a bounding limit for shaders of any complexity. In addition, if you manually compile the shaders using the DirectX SDK, you'll need to know what profile to use.

Environment setup

The most difficult part of writing a pixel shader is setting up the environment to allow them to compile. There are three main options:

- Download the DirectX SDK, and use the compiler there to build the shader.
- Repurpose the WPF pixel shader build step.
- Use a tool such as Shazzam to create and compile the shader.

You can download the DirectX SDK and use its command-line tools to compile the shader. The SDK is roughly 500 MB and may be a bit much just to compile a shader.

Option 2 is to repurpose the WPF pixel shader build step. Tim Heuer put together a great blog post covering the steps required to set up your environment for developing pixel shaders. It'd be too much to include here, so I refer you to his post here: http://bit.ly/SLPixelShaderCompile. I chose option 2, using a build task. It involves some configuration as well as a template for the shader development.

Another option is to use a tool such as Shazzam (http://shazzam-tool.com) to compile the shader and manually add that into your project. Most Silverlight and WPF developers doing serious work with pixel shaders use this tool. It also includes a number of training videos to help you get started with pixel shader development. Finally, Shazzam includes a bunch of existing shaders in source form that you can learn from.

Despite its hackish nature, if you want everything to happen inside Visual Studio, I think you're better off starting with Tim's approach for the project structure. If you don't need everything integrated into Visual Studio and can add the files manually, you'll find that Shazzam is the best long-term solution. In either case, you'll likely have Shazzam open while you explore pixel shader development.

When your environment is set up and you can compile pixel shaders, you're ready to develop one of your own. Go ahead and set up your environment now. I'll wait.

Shader code

Pixel shaders are typically fairly complex; they do things such as alter the visual location of pixels based on a complex algorithm. Learning how to write shaders is like learning any other programming language, but with a heavy focus on performance and optimization.

A good place to learn is the WPF Pixel Shader Effects library on CodePlex: http://wpffx.codeplex.com. Although originally intended for WPF use, Silverlight effects were added once Silverlight supported HLSL-based pixel shaders.

Listing 4 shows the HLSL source code for a pixel shader that takes a color and multiplies every pixel by that color. The result is an image that appears to have been photographed through a tinted lens.

Listing 4 A simple pixel shader that applies a color filter

```
//-----------------------------------------------------------------------
//
// Silverlight ShaderEffect HLSL -- ShaderEffect1
//
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
// Constant register mappings (float,double,Point,Color,Point3D...)
//-----------------------------------------------------------------------
float4 colorFilter : register(C0);
//-----------------------------------------------------------------------
// Sampler Inputs (Brushes, including ImplicitInput)
//-----------------------------------------------------------------------
sampler2D implicitInputSampler : register(S0);
//-----------------------------------------------------------------------
// Pixel Shader
//-----------------------------------------------------------------------
float4 main(float2 uv : TEXCOORD) : COLOR                          #A
{
  float4 color = tex2D(implicitInputSampler, uv);
  return color * colorFilter;
}
```

#A Standard main function

The shader first maps values into registers supported by the shader model. Each input and constant must be mapped to a register. A *register* is a well-known place in hardware (virtual hardware in the Silverlight case) that can be used to store a value. Registers are much faster than regular RAM when it comes to accessing values. If you've ever done any x86 assembly language programming, or even any old DOS interrupt programming, you know well the concept of registers.

The section comments aren't required, but you'll find them in almost every pixel shader implementation.

The actual code starts under the Pixel Shader section. Like all pixel shaders in Silverlight, this has a main function that takes in a UV *coordinate* (a standard way of referring to an x and y position on a texture or image, but normalized into the range of 0 to 1 rather than absolute pixels) and returns a `float4` color.

When you have the HLSL source for your shader, you'll need to write a .NET class to expose it in your project.

Wrapper class

To use a pixel shader, you need to provide a way for the rest of .NET to interact with it. The wrapper class (often called just the *pixel shader class*) is responsible for loading the compiled shader code and for exposing properties used to tweak the shader. The Pixel Shader file template includes a wrapper class. In addition, Shazzam will generate the wrapper class for you. The wrapper class for this example is shown in listing 5.

Listing 5 A pixel shader wrapper class

```
public class ShaderEffect1 : ShaderEffect
{
  private static PixelShader _pixelShader = new PixelShader();
  static ShaderEffect1()
  {
    _pixelShader.UriSource = new
        Uri("/SilverlightApplication61;component/ShaderEffect1.ps",
        UriKind.Relative);
  }
  public ShaderEffect1()                    #A

  {
    this.PixelShader = _pixelShader;
    UpdateShaderValue(InputProperty);
    UpdateShaderValue(ColorFilterProperty);
  }
}
public Brush Input   #B
{
    get { return (Brush)GetValue(InputProperty); }
    set { SetValue(InputProperty, value); }
}
public static readonly DependencyProperty InputProperty =
    ShaderEffect.RegisterPixelShaderSamplerProperty("Input",
                  typeof(ShaderEffect1), 0);

public Color ColorFilter
{
    get { return (Color)GetValue(ColorFilterProperty); }
    set { SetValue(ColorFilterProperty, value); }
}
public static readonly DependencyProperty ColorFilterProperty =
    DependencyProperty.Register("ColorFilter", typeof(Color),
    typeof(ShaderEffect1), new PropertyMetadata(Colors.Yellow,
                  PixelShaderConstantCallback(0)));

}
```

#A Public instance constructor
#B Implicitly mapped

The static constructor (#1) loads the pixel shader resource into a static `PixelShader` typed property. Note the .ps extension: it's loading the compiled resource. The `PixelShader` is static because only one copy of the compiled code is needed within an application.

Each of the dependency properties maps to a register in the shader. One of the properties, of type `Brush`, is mapped implicitly. Any additional properties must be mapped directly to registers. In this source, you can see that the `ColorFilterProperty` maps to constant register zero in the pixel shader. The `PixelShaderConstantCallback` takes the register number as a parameter. In the HLSL source, constant register zero is mapped to the variable `colorFilter`.

But how did a `Color` property become a `float4`, and what's a `float4` anyway? Those are built-in vector types in the language. Table 2 has the mapping.

Table 2 Mapping from .NET types to HLSL types

Shader type	Description	.NET type
`float`	A single floating-point number	`double`, `single`
`float2`	A vector with two floating-point numbers	`Point`, `Size`, `Vector`
`float3`	A vector with three floating-point numbers	(Unused in Silverlight)

`float4`	A vector with four floating-point numbers	`Color`

The member names for the individual floats depend on their usage. For example, a color has the properties `r`, `g`, `b`, and `a`.

HLSL is interesting in that it can perform multiplication and other operations on whole structures. In that way, the number of instructions is reduced, but it can be hard to understand when you first look at it. For example, the exmple multiplies together two `float4` values.

Using the shader

With the shader compiled and the wrapper class in place, it's time to try the shader in your own application. Like any other element used in XAML, you must either include an implicit namespace in your application settings or map a namespace in the XAML file. In this case, because the shader is in the project with the XAML, you'll use an explicit map in the XAML.

Listing 6 shows the effect of using the shader with a red tint. It'll look gray in print, but you can tell there's a tint over the whole image.

Listing 6 Using the pixel shader effect in XAML

Result:

Code:

```
<UserControl x:Class="SilverlightApplication61.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:local="clr-namespace:SilverlightApplication61"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">
    <Grid x:Name="LayoutRoot">
        <Grid Background="White" Width="180" Margin="25">
            <StackPanel x:Name="Elements" Margin="10">
                <TextBlock Text="Hello World" Margin="10" />
                <TextBox Text="This is a textbox" Margin="10" />
                <Button x:Name="Button" Content="Button"
                        Margin="10" />
            </StackPanel>
            <Grid.Effect>
                <local:ShaderEffect1 ColorFilter="Red" />     #A
            </Grid.Effect>
        </Grid>
    </Grid>
</UserControl>
```

#A Colorfilter Property

This example uses the pixel shader with a parameter of `Red` for the `ColorFilter` property. The end result is an angry red form. The use of a pixel shader has reverted the text back to grayscale font smoothing.

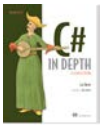
Pixel shaders are a great way to provide your own custom effects or to use effects developed by others. Learning HLSL can be difficult at times, but the payoff is worth it: you can use pixel shaders in Silverlight, in WPF, and, of course, in DirectX and XNA. Pixel shaders, even the software-rendered ones in Silverlight, are extremely efficient as well. When considering pixel-manipulation strategies in an application, the creation of a pixel shader should be high on your list of options.

Summary

Effects augment both graphics and controls. The use of a subtle drop shadow or a blur can help users focus their attention on a specific part of the screen. If those effects aren't sufficient, you also have the option to create your own effects in the form of pixel shaders.

Here are some other Manning titles you might be interested in:

Entity Framework 4 in Action
Stefano Mostarda, Marco De Sanctis, and Daniele Bochicchio



C# in Depth, Second Edition
Jon Skeet

Dependency Injection in .NET
Mark Seemann

Last updated: November 18, 2011