A Comprehensive Guide to Creating HLSL Pixel Shaders for WPF and Silverlight Applications

HLSL and Pixel Shaders

for XAML Developers

O'REILLY®

Walt Ritscher

1

Shader 101

It seems an obvious question to ask at the beginning of an *HLSL* and shader book; what exactly is a shader? It's a small program or algorithm written explicitly to run on a computer Graphics Processing Unit (*GPU*). It provides a way for developers to extend the rendering capabilities of the GPU. Any program that works closely with graphics will benefit from using shaders. The video game industry spins off custom shaders by the thousands, they are as vital to game projects as business entity classes are to line of business applications. Nothing prohibits business programmers from experimenting with shaders in their line of business (LOB) applications, in fact recent trends in user interface (UI) design and information visualization cry out for shader use.

Because shaders run at the kernel level of the GPU they are automatically parallelized by the GPU hardware and are extremely fast at manipulating graphic output. Typically, the GPU can process shaders several orders of magnitude faster than if the shader code is run on a CPU.

Why XAML developers should learn HLSL?

If you are a *XAML* developer, I'll wager you've heard about pixel shaders. In fact, you may be using some of these effects in your application already. WPF introduced the *DropShadowEffect* and *BlurEffect* in .NET 3.5 SP1 and both of these classes take advantage of pixel shaders. *Silverlight* added pixel shaders in Silverlight 3. The Windows Phone team disappointed developers by dropping support for shaders before the final release of their hardware. Microsoft had good reason to ditch phone shaders as they caused a significant drag on performance , but their loss is still lamentable, To make up for that setback the Silverlight 5 release includes support for XNA models and shaders.

This is awesome news as it means that you can mix XNA and Silverlight 5 together in the same application and that gives you access to another essential shader type; the *Vertex* shader.

> XNA is a Microsoft framework that facilitates game development on the PC, the Xbox 360, and Windows Phone 7. It give you access to the power of DirectX without having to leave the comfort of your favorite

> .NET programming languages. To learn more about XNA get a copy of Learning XNA 4.0 by Aaron Reed from: http://shop.oreilly.com/product/0636920013709.do

As a XAML developer, do you need to write your own shaders? No, not really, you may spend your entire career without ever using a shader. Even if you use a shader you may never have the need to write your own as there are free shader effects included in Microsoft Expression Blend and also in the .NET framework. While it's nice to have these prebuilt effects, they represent only a fraction of the possibilities discovered by graphics programmers. Microsoft is not in the shader business, at least not directly. A core part of their business is building flexible programming languages and frameworks. The DirectX team follows this path and provides several shader programming languages for custom development. So if you have an idea for an interesting effect or want to modify an existing effect you'll need to write a custom shader. When you cross that threshold and decide to build a custom shader, you have some learning ahead of you. You need to learn a new programming language called HLSL.

> I've started using the term XAML development in the last year. Extensible Application Markup Language (XAML) is the core markup for Windows Presentation Foundation, Microsoft Surface, Silverlight and Windows Phone applications. There are differences between these technologies but they all share a common markup in XAML. Even the new Metro application framework for Windows 8 uses XAML as its primary markup implementation. I find that WPF and Silverlight developers have more in common with one other than they have differences. Since there is so much overlap in skills between these XAML based systems I think XAML developer is a suitable umbrella term that symbolizes this commonality.

The Tale of the Shader

To understand the history behind shaders we need to go back a few decades and look inside the mind of George Lucas. Thirty years ago, George had produced the first movies in his highly successful Star Wars series. These first movies relied on using miniaturized models and special camera rigs to generate the futuristic effects. Lucas could already see the limitations of this camera based system and he figured that generating his models in software would be a better approach. Therefore, he established a software division at LucasFilm and hired a team of smart people to build a graphics rendering system. Eventually the software division he created was sold off and became Pixar.

The engineers hired by Lucas took their responsibilities seriously and were soon generating 3-D objects within software. But these computer generated models failed when spliced into the live action as they suffered from a lack of realism. The problem is that a raw 3D object looks stark and unnatural to the movie viewer, and won't blend with the rest of the movie scene. In other words, it will be painfully obvious that there is a computer-generated item up on the big screen. In the quest to solve this problem an engineer named Rob Cook decided to write a 'shading' processor to help make the items look more realistic. His idea was to have software analyze the 3D object and the surrounding scene and determine where the shadows fell and light reflected onto the model. Then the shader engine could modify the film output to imitate the real world placement of the artificial artifact. To be fair, there were existing shade tools available

but they were primitive and inflexible. Rob's breakthrough idea was to make a scriptable pipeline of graphic operations. These operations were customizable and easy to string together to create a complex effect. These "shaders" eventually became part of an infamous graphics program called Renderman, which remains the primary rendering tool used for every Pixar movie. While you may not be familiar with the Renderman name you certainly have seen the output from this phenomenal program in movies like Toy Story 3.

Pixar has an informative section devoted to Renderman on their website at http://renderman.pixar.com/products/index/renderman.html

The beginnings of this shader revolution started back in the early 1980's and ran on specialized hardware. But the computer industry is never idle. By the late nineties 3D graphics accelerator cards started to show up in high end PCs. It wasn't long before card manufacturers figured out how to combine 2D and 3D circuits into a single chip and the modern Graphics Processor Unit (GPU) was born. At this same time, the GPU manufacturers came up with their own innovative idea -- real-time rendering -- which allows processing of 3D scenes while the application is running. Prior to this breakthrough, the rendering was performed off-line. The burgeoning game development industry embraced this graphics advance with enthusiasm and soon 3D frameworks like OpenGL and Microsoft Direct3D were attracting followers. This is the point in the story where HLSL enters the picture.

HLSL and DirectX

In the early days of GPUs, the 3D features were implemented as embedded code within the video card chips. These Fixed Functions, as they were known, were not very customizable and so chipmakers added new features by retooling the chips and throwing hardware at the problem. At some point Microsoft decided this was solvable with software and devised an assembly language approach to address the problem. This worked and made custom shaders possible but you needed developers who could work in assembly language. Assembly language is notoriously complex and hard to read, for example here is a small sample of shader assembly code for your reading pleasure.

Example 1-1. Shader written in Assembly Language

[c-objdump]

```
; A simple pixel shader
; Use the ps 2.0 instruction set and registers
ps_2_0
;
; Declare a sampler for the s0 register
dcl_2d s0
; Declare t0 to use 2D texture coordinates
dcl t0.xy
; sample the texture into the r1 register
texld r1, t0, s0
; move r1 to the output register
mov oC0, r1
```

> DirectX 8.0 was the first version to include programmable shaders. It first appeared in 2000 and included the assembly level APIs.

Working in assembly takes a special breed of programmer and they are in short supply. NVidia and Microsoft saw this as an opportunity to bolster the PC graphics market and collaborated to create a more accessible shader language. NVidia named their language Cg while Microsoft chose the name High Level Shader Language (HLSL) for their version. Cg and HLSL have virtually identical syntax; they are branded differently for each company. Both languages compile shaders for DirectX. Cg has the additional benefit of compiling shaders for the OpenGL framework.

> The Open Graphics Library, AKA OpenGL, is an open source, cross platform 2D/3D graphics API.

These higher level languages are based on the C language (in fact the name Cg stands for *C for Graphics*) and use curly braces, semicolons and other familiar C styled syntax. HLSL also brings high-level concepts like functions, expressions, named variables and statements to the shader programmer. HLSL debuted in DirectX 9.0 in 2002 and has seen steady updates since its release.

Let's contrast the assembly language shown in Example 1-1 with the equivalent code in HLSL.

Example 1-2. Shader written in HLSL

[C#]

```
sampler2D ourImage;

float4 main(float2 locationInSource : TEXCOORD) : COLOR
{
  return tex2D( ourImage , locationInSource.xy);
}
```

Here, the first line is declaring a variable name *ourImage* which is the input into the shader. The next line defines a function called *main* that takes a single parameter and returns a value. That return value is vital, as it is the output of the pixel shader. That float4 represents the RGBA values that are assigned to the pixel shown on the screen.

This is about the simplest pixel shader imaginable. Trust me, there are more details ahead. This is a preliminary look at shader code; there are detailed discussions of HLSL syntax throughout the remainder of this book.

> This is the first HLSL example in the book but it should be obvious to anyone with a modicum of programming experience that the HLSL version is easier to read and understand than the assembly language version.

Understanding the Graphics Pipeline

HLSL is the shader programming language for Direct3D, which is a part of Microsoft's DirectX API. Appendix A contains a detailed account of Direct3D and the graphics-

programming pipeline. What follows is a simplified account of the important parts of the shader workflow.

To understand pixel shaders in the XAML world requires a quick look at how they work in their original Direct3D world. Building a 3D object starts with defining a model. In DirectX, a model (aka mesh) is a mathematical representation of a 3D object. These meshes are defined as arrays of vertexes. This vertex map becomes the initial input into the rendering pipeline.

> If you studied geometry, you've seen the term vertex. In solid geometry, a vertex represents a point where three planes meet.
>
> In the DirectX realm, a vertex is more than a 3D point however. It represents a 3D location so it must have x, y and z coordinate information. Vertices may also be defined with color, texture, and lighting characteristics.

The 3D model is not viewable on screen without conversion. Currently the two most popular conversion techniques are ray tracing and rasterization. Rasterization is widespread on modern GPUs because it is fast, which enables high frame rates – a must for computer games.

As I mentioned before, the DirectX graphics pipeline is complex, but for illustration purposes, I'll whittle it down to these few components:

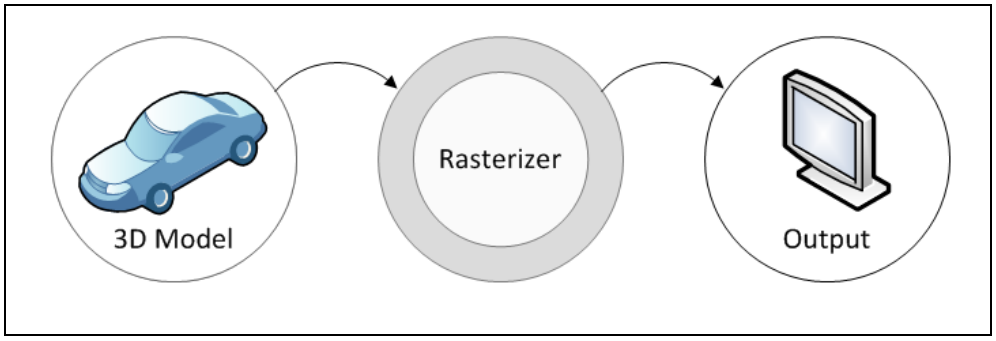

Figure 1-1. Three DirectX pipeline components

DirectX injects two other important components into this pipeline. Between the model and the rasterizer lives the vertex shader. Vertex shaders are algorithms that transform the vertex information stored in the model before handoff to the rasterizer.

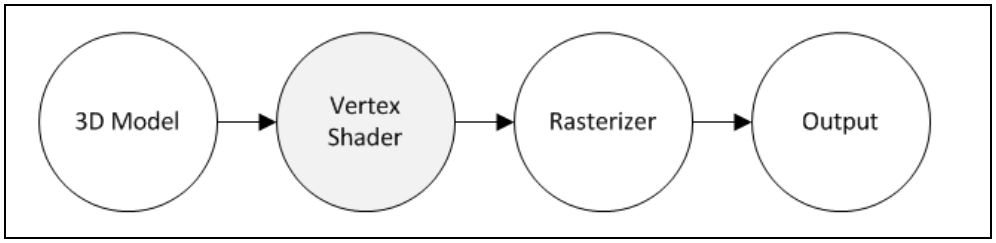

Figure 1-2. The vertex shader in the pipeline

Vertex shaders get the first opportunity to change the initial model. Vertex Shaders simply change the values of the data, so that a vertex emerges with a different texture,

different color, or a different position in space. Vertex shaders are a popular way to distort the original shape and are used to apply the first lighting pass to the object. The output of this stage is passed to the rasterizer. At this point in the pipeline the rasterized data is ready for the computer display. This is where the pixel shader, if there is one, goes to work.

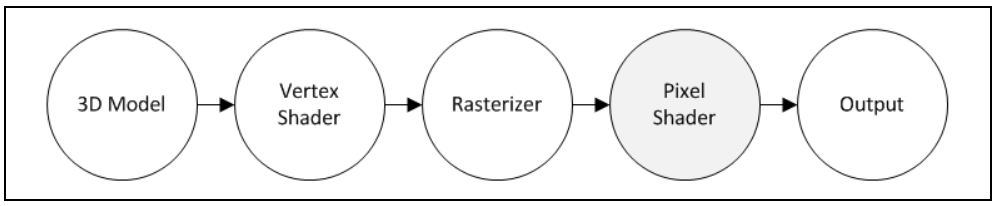

Figure 1-3. The pixel shader in the pipeline

The pixel shader examines each rasterized pixel, applies the shader algorithm, and outputs the final color value. They are frequently used to blend additional textures with the original raster image. They excel at color modification and image distortion. If you want to apply a subtle leather texture to an image the pixel shader is your tool.

XAML and Shaders

Now that you've learned the fundamentals of the DirectX pipeline you're ready to take a closer look at how Silverlight and WPF use shaders. Let's examine what happens in the WPF world first. In WPF, the underlying graphics engine is DirectX. That means that even on a simple business form consisting of a few text controls the screen output travels through the DirectX pipeline. The very same pipeline described above. WPF takes your XAML UI tree and works its magic on it, instantiating the elements, configuring bindings and performing other essential tasks. Once it has the UI ready it passes it off to DirectX which rasterizes the image as described earlier. Here's what the process looked like in the first release of WPF.

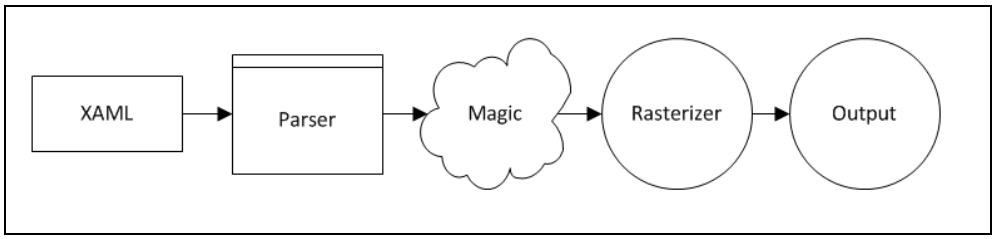

Figure 1-4. The .NET 3.5 render process

As you can see, there were no vertex or pixel shaders available. It took another couple years for Microsoft to add shaders to the mix. Pixel shaders appeared in .NET 3.5 in 2007 and now the process looks like this.

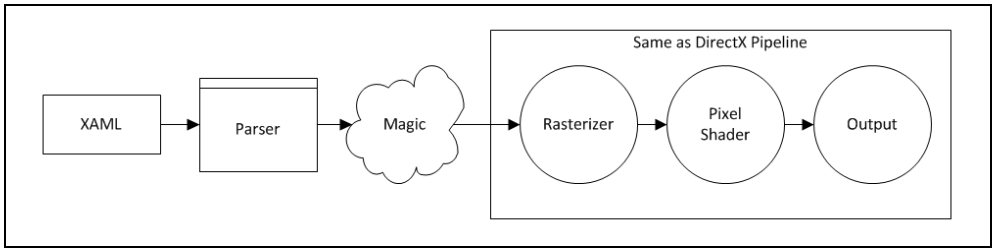

Figure 1-5. *.NET 4.0 adds pixel shader to the render process*

Notice how the end of this pipeline is identical to the 3D model pipeline cited earlier. As you can see the input data for a pixel shader is the output from the rasterizer. It really doesn't matter to the shader whether that information is a rasterized version of a complex 3D shape or the output from a XAML visual tree. The shader works the same way for both, since it is only 2D information at this point.

You might notice that there are no Vertex shaders in the WPF pipeline. That's not an omission on my part. Vertex shaders are not available to WPF and there are no plans to add them to WPF. The likely reason for this oversight was the release of XNA, Microsoft's managed game development platform. XNA has a tight relationship with DirectX/Direct3D and treats 3D and models nearly the same as native DirectX.

Don't be too sad over the loss of vertex shader, pixel shaders are still a powerful technique and can create a variety of useful effects. In fact, since current PC hardware is so powerful, game developers often prefer using pixel shaders for lighting calculations, a job that use to be handled by vertex shaders.

Silverlight is similar to WPF in many respects when it comes to shaders. Silverlight supports pixel shaders like WPF. It doesn't support vertex shaders directly. Instead, it uses XNA integration for 3D rendering. The Silverlight team chose to embrace the XNA framework and integrate it into their specifications rather than write their own 3D engine. If you are an experienced XNA developer, you should have no problem adapting to the Silverlight version.

In Silverlight, pixel shaders are always executed on the CPU. In fact, the rasterizer also runs on the CPU.

WPF, on the other hand, runs the shaders on the GPU, falling back to CPU only in rare cases. Because Silverlight uses the CPU, you might worry about performance. You may suspect that Silverlight is slower when processing shaders and you'd be correct. Silverlight mitigates some of the performance woes by running shaders on multiple cores (when available) and by using the CPUs fast SSE instruction set. Yes, Silverlight shaders are slower than their WPF counterparts. When it comes to pixel manipulation, Silverlight shaders are still the fastest option though, beating other venues like WriteableBitmap by a substantial margin. If you want to see the performance ramifications for yourself, René Schulte has an illuminating Silverlight performance demo that you should check out when you have the time.

http://kodierer.blogspot.com/2009/08/silverlight-3-writeablebitmap.html

Summary

Pixel shaders have revolutionized the computer graphics industry. The powerful special effects and digital landscapes shown in modern movies and games would not be possible

without them. Adobe Photoshop and other designer applications are jammed with effects that are implemented with pixel shaders. They are a magnificent way to create your own custom effects. Granted, the HLSL syntax is a bit cumbersome and difficult to understand at first, but it’s worth learning. Once you master HLSL, you can create shaders for DirectX, XNA, WPF, Silverlight and Windows 8 Metro. In the next chapter, I’ll show you how to create your first XAML shader project. By the end of this book, you’ll be able to add the title “HLSL ninja” to your resume.

2

Getting Started

In this chapter, you get your first look at using shaders in an XAML application. Using the prebuilt shaders in .NET is a snap. It's not much harder than using a drag and drop UI designer. You will also get a miniature tutorial on creating a simple custom shader.

Setting up your development computer

If you are a .NET developer, you know a lot about managed code and the .NET framework libraries. If Microsoft statistics are accurate, you write your code in either C# or Visual Basic. Moreover, if you are like me, you are a Visual Studio junkie, spending countless hours living inside the Visual Studio environment. Given these facts and the possibility that you are also an experienced XAML developer it's likely that you already have your computer ready to create HLSL shaders. But I'm going to be methodical and show you what you need to make sure your development computer is set up correctly.

Silverlight development

One thing you can say about the Silverlight team; they produce high quality releases on a tight schedule. Silverlight 5 is the newest version available at this time. It requires a Visual Studio 2010 installation in order to build a Silverlight 5 project. If you are cheap, all you need is a copy of the free Visual Web Developer 2010 Express edition (http://www.microsoft.com/express/web/) to be ready to create Silverlight applications. If you have access to your corporate credit card, buy Visual Studio 2010 pro, premium or ultimate. You get a lot more tools in these editions and they are indispensable for real world development. To be fair though, there is nothing in the more expensive editions that makes HLSL development any easier.

Since Silverlight 5 shipped after Visual Studio 2010 you need to visit http://www.silverlight.net/getting-started and install the Silverlight 5 tools and SDK before you are completely ready to start coding.

WPF development

To get the most out of your shader code use the .NET 4.0 version of WPF. That's because 4.0 supports a more recent shader specification (PS_3_0) and that gives you more shader

power. For the skinflints in the audience, look at the Visual C# 2010 Express (http://bit.ly/VCS2010Express) or Visual Basic 2010 Express (http://bit.ly/VB2010Express) editions. Both express editions are fully capable of creating WPF applications and incorporating your shader code. Just like with Silverlight 5, you can use the commercial editions of Visual Studio for development (http://bit.ly/wEUD17).

The Visual Studio install takes about an hour. I suspect most readers have gone through the installation process many times so I'll assume you know what you are doing and don't need step by step instructions.

Expression Blend 4

I highly recommend that XAML developers learn Expression Blend (http://bit.ly/expressionblend4/). It contains dozens of tools that simplify XAML UI development and it is a perfect companion for Visual Studio. For the shader developer, it is useful for two reasons. First, it ships with nice set of prebuilt shader effects. Second, it provides a preview feature, making it easy to see the effect without having to run the application first.

Installing Blend is a ten-minute exercise. Download the installer from the Microsoft site and follow the prompts.

Choosing a Shader Compiler

Your HLSL shader source code is just text, any text editor will suffice for code creation. Before you can use the shader it must be compiled into a binary file and added the GPU during runtime. Thus, you need a compiler.

DirectX Compiler

Visual Studio is such a powerhouse that many assume it has compilers for any programming language, but neither Visual Studio 2010 nor Expression Blend 4 includes a shader compiler. There is good news on the horizon though; the next version of Visual Studio has some remarkable 3D editors, and it will have a shader compiler. In the meantime, you need to find a compiler before you can continue.

Since HLSL is a part of DirectX you can use the FXC.exe compiler in the DirectX SDK. FXC.exe is a small file; it's less than 200KB in size. Regrettably, the FXC compiler is only available as part of the SDK and that SDK is a monster, using up over one gigabyte of your hard drive space.

I use the FXC compiler for some of the examples in this book. If you want to follow along you can find the DirectX SDK at http://msdn.microsoft.com/directx/.

WPF Build Task

The good folks on the WPF team have a few open source projects on Codeplex (http://wpf.codeplex.com/). If you snoop around their Codeplex site you'll find a shader build task that doesn't require having the DirectX SDK installed (http://wpf.codeplex.com/releases/view/14962/). Here's what is does. If it is installed it enhances the normal MSBuild build process. It looks for any files with an .fx extension within your WPF project. It compiles the source in that .fx file into a binary file (*.ps). It has two exasperating limitations however; it not available for Silverlight projects and it doesn't compile to the newer PS_3_0 specifications.

Shazzam Shader Editor

Shazzam Shader Editor is a free stand-alone tool for writing XAML shaders. If you install Shazzam, you don't need to install the massive DirectX SDK as it has its own compiler. It contains an HLSL code editor that has intellisense and code completion. That might not seem like a big deal; until you learn that Visual Studio doesn't have intellisense for HLSL files. Earlier in this chapter I mentioned the effect preview feature in Expression Blend. Shazzam does Blend one better, featuring a spate of preview, animation and comparison features. Shazzam has been available for the last four years and has thousands of users around the world. It's the recommend shader tool in nearly every Silverlight book on the market.

I have to tell you, in the interest of full disclosure, I'm the primary developer for Shazzam. Of course I think it's the best tool available for learning XAML specific HLSL. I encourage you to download a free copy from http://shazzam-tool.com and see for yourself. I use Shazzam extensively throughout this book, so be sure and read Chapter 6 to learn more about the tool.

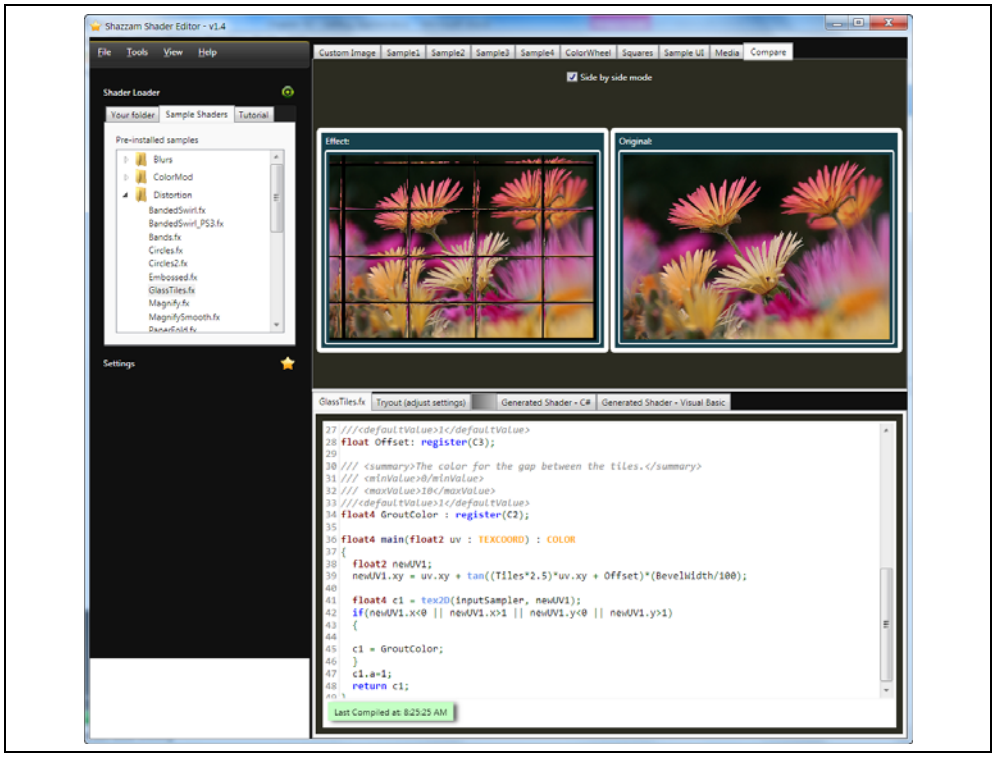

Figure 2-1. Shazzam Shader Editor IDE

To install Shazzam, download the installer from shazzam-tool.com and follow the prompts. The install takes less than a minute on most computers.

Other Tools to Consider

FX Composer

NVidia has a stake in the shader community and have a number of developer tools. One of their more impressive applications is the FX Composer tool. It is aimed squarely at the game development community and has tons of remarkable features. It boasts a shader debugger, supports easy importation of 3D models, has a particle system generator, test harness and many other cool enhancements.

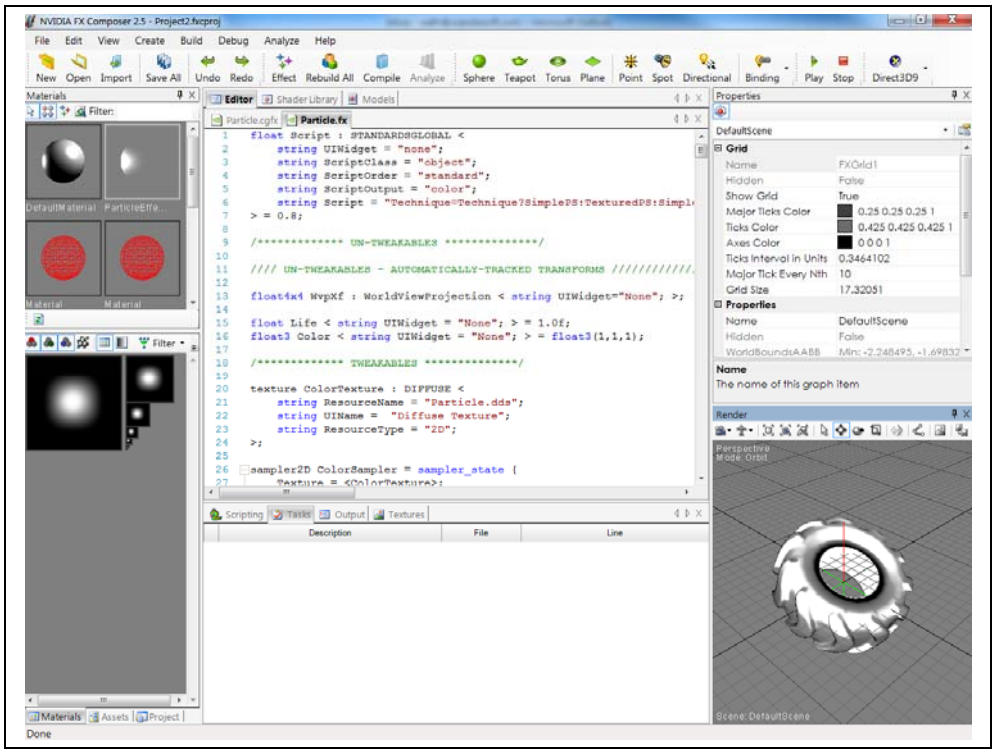

Figure 2-2. NVidia FX Composer IDE

It's an impressive tool but I find it overkill for creating pixel shaders for Silverlight or WPF. You can find it on the NVidia site at http://developer.nvidia.com/fx-composer/ .

NShader

NShader (http://nshader.codeplex.com/) is a Visual Studio extension, which provides syntax highlighting for assorted shader languages including HLSL, Cg and GLSL. If you write HLSL in Visual Studio, you may find this tool useful. It's strangely incomplete though and misses some obvious HLSL functions like **sampler2D()**.

Visual Studio Next

The next version of Visual Studio, code name Visual Studio 11, is available in a developer preview. I've looked at the 3D tools and I'm impressed. Download a free copy of the developer preview (http://bit.ly/vs11devpreview) to see what's coming.

A First Shader Project

Traditionally the first application in most programming books is the ubiquitous "hello world" application. Boring! I'll have none of that dreary code in this book. The cynical reader will point out that writing text to the screen with a shader is nearly impossible but let's not go there. I need a graphical demo that shows a shader in action, but also is graphical in nature. With that in mind I decided to make the first project an image transition application.

> I want to say a couple words about terminology before going any further. The terms *shader* and *effect* are pervasive and often used interchangeably. On the .NET side of the aisle, the common term is effect. Examine the UIElement class and you'll see that it has an Effect dependency property. Tour the class libraries and you'll discover the prebuilt DropShadowEffect and BlurEffect classes tucked amidst the other familiar XAML types. In addition, there is the ShaderEffect base class, which is used when creating your own custom effects.
>
> On the HLSL side of the house the word effect has special meaning; in this realm you can think of an effect as a *shader package* containing multiples shaders targeting different hardware. The shader is the actual algorithm run on the GPU. To summarize, when you write a custom effect you create a .NET effect class and an HLSL shader.

Using prebuilt effects

Silverlight and WPF have two built-in effects: *dropshadow* and *blur*. These effects are applicable to any UIElement, which includes any element in the visual tree. When an effect is applied to an element it affects the element and all of its children elements. In our first project, you'll place the effect on an Image element.

To begin, open your Visual Studio edition and choose File→New Project from the menu bar (or press Ctrl-Shift-N). The New Project dialog opens, offering a number of options, as shown in Figure 2-3.

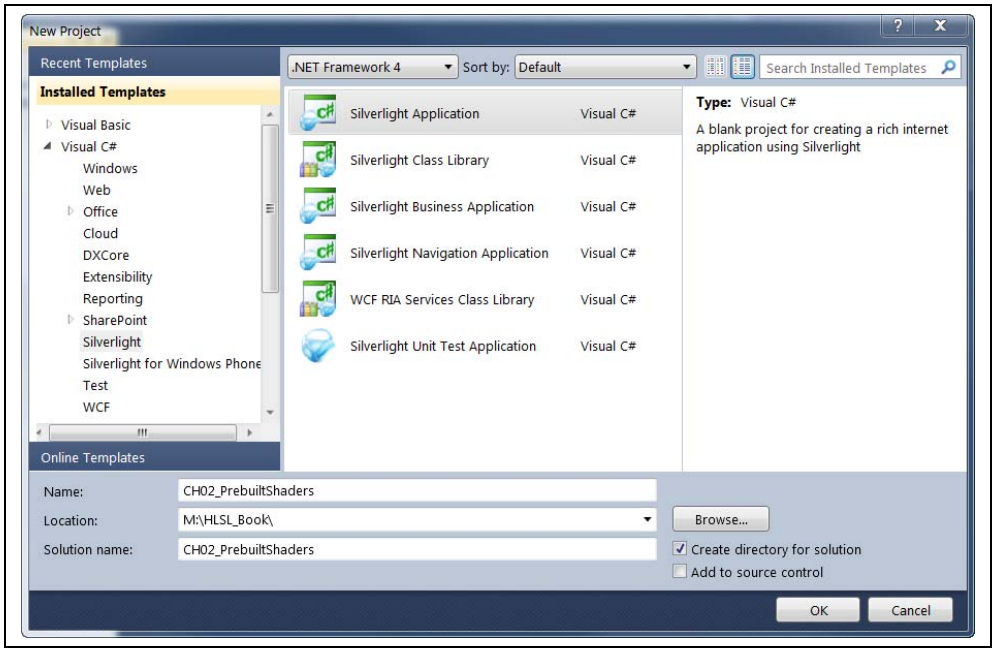

Figure 2-3. The New Project dialog

Select the Silverlight Application template. Enter a name of your choosing and click OK. Visual Studio will start creating a Silverlight application for you.

The New Silverlight Application dialog box will appear, asking whether you want to host the application in a new Web site (Figure 2-4). For simple Silverlight applications, I rarely create a host Web Project so I recommend unselecting that check box. Leave the other options set to their defaults for now and click OK.

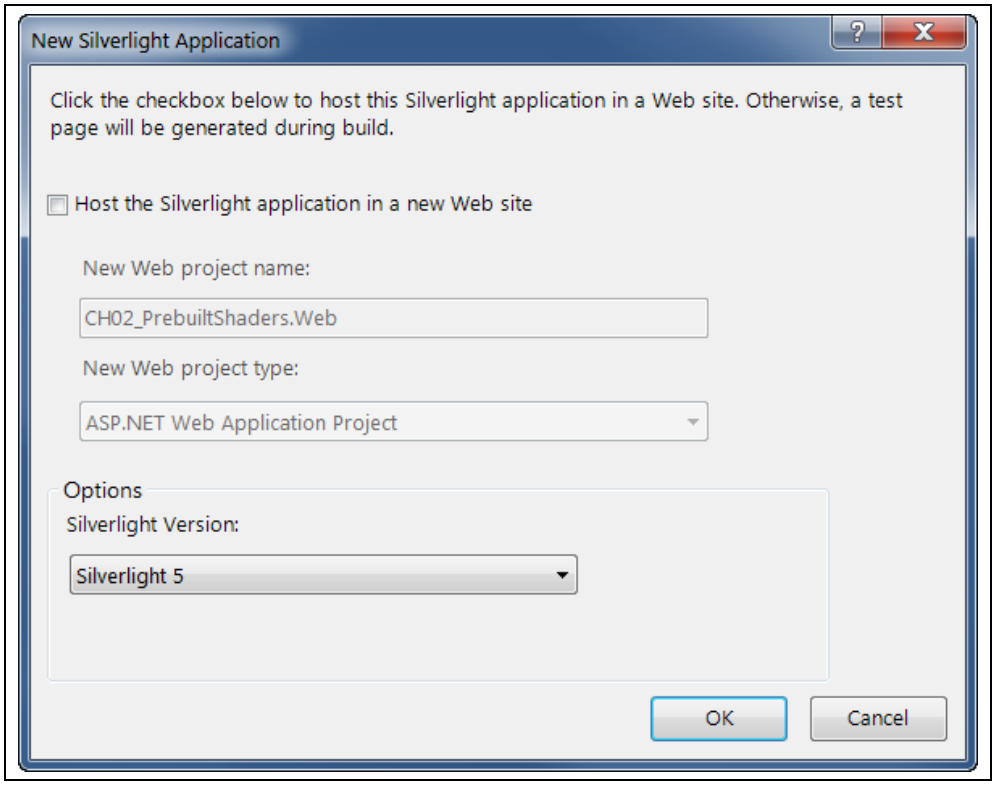

Figure 2-4. New Silverlight Application dialog

Use the Project→Add Existing Item menu bar to add a couple image files to the project.

Next, open the MainPage.xaml file and modify the following values on the UserControl element.

[XML]

```
    d:DesignHeight="600" d:DesignWidth="800"
```

Setting the DesignHeight and DesignWidth properties make it easier to see the images on the Visual Studio designer.

Add the following XAML to the MainPage.xaml file.

Example 2-1. Add Images and Slider XAML

[C#]

```
<Grid x:Name="LayoutRoot"
      Background="White">
  <Grid.RowDefinitions>
    <RowDefinition Height='380' />
    <RowDefinition Height='40' />
  </Grid.RowDefinitions>
  <!--  set the Source to a valid path in your project -->
  <Image x:Name='StartImage'
         Source='garden1.jpg'
         Width='500'
```

```
          Opacity='1'></Image>

    <!--  set the Source to a valid path in your project -->
    <Image x:Name='EndImage'
           Source='garden2.jpg'
           Width='480'
           Opacity='0'></Image>
    <Slider x:Name='TransitionSlider'
            Grid.Row='1'
            Width='500' />
</Grid>
```

The XAML in Example 2-1 creates two Image elements, one superimposed over the other. The width of the **EndImage** is smaller than the **StartImage** to accentuate the transition. There is also a **Slider** element, located at the bottom of the grid, which is used to control the transition amount.

Here is what the UI looks like at this stage.

Figure 2-5. Transition project, phase 1

In the next phase you will add a couple lines of code to fade between the two images when the slider is moved. Start by adding a **ValueChanged** event handler to the existing XAML. Experienced XAML developers know that Visual Studio shows a *New Event Handler* prompt (Figure 2-6) when adding event attributes in the XAML editor. Pressing **Tab** at the prompt stubs in the correct attributes value and writes an event procedure in the code behind.

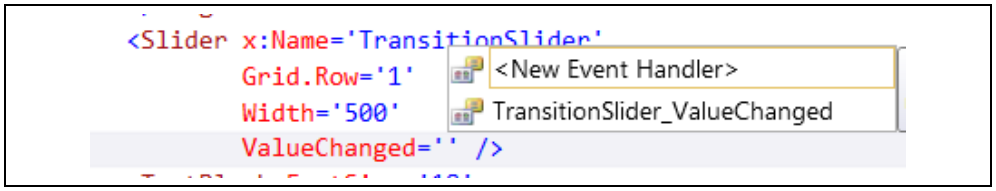

Figure 2-6. The Insert New Event Handler prompt

When you are done, your XAML for the slider should look like Example 2-2.

Example 2-2. ValueChanged event text

[XML]

```
<Slider x:Name='TransitionSlider'
        Grid.Row='1'
        Width='500'
        ValueChanged='TransitionSlider_ValueChanged' />
```

Press **F7** to switch to the code behind view and add the following code to the C# file.

Example 2-3. The TransitionSlider_ValueChanged event code

[C#]

```
private void TransitionSlider_ValueChanged(object sender,
                RoutedPropertyChangedEventArgs<double> e) {
  // transition the images
  var max = TransitionSlider.Maximum;
  EndImage.Opacity = e.NewValue / max;
  StartImage.Opacity = 1 - EndImage.Opacity;
}
```

As you can see, this code changes the opacity of the two Image elements. Opacity accepts a value between 0.0 and 1.0 so the code uses a calculation to normalize the current slider value to that range.

[C#]

```
e.NewValue / max
```

The last line ensures that when StartImage.Opacity is at 0.0, the EndImage.Opacity is set to 1.0 and vice versa.

[C#]

```
StartImage.Opacity = 1 - EndImage.Opacity;
```

Run the application and drag the slider to change the value. The first image gradually disappears as the second image fades into view.

Adding Effects

To make the transition more interesting you can apply a BlurEffect during the fade-in and fade-out. The BlurEffect is nice choice for your first look at a built-in effect. It's one of the built-in effects, it's available for Silverlight and WPF, and is quite simple to use. There are different types of blur effects used in the graphics industry (motion blur, zoom blur, Gaussian blur). The BlurEffect class is one of the simplest implementations,

providing a uniform unfocused look to the affected pixels. If you've ever looked through a frosted translucent screen, you've seen the effect.

> The BlurEffect uses a simplistic blur algorithm, which is effective but slower than other potential blur implementations. A custom BoxBlur or optimized Gaussian Blur outperforms the built-in WPF blur.

Each **UIElement** has an **Effect** property. In this project, the potential candidates for the effect are the UserControl, Grid, Slider and the two Image elements. When you apply an effect to a parent element, like the Grid, it affects all the children elements. Each element can have only one effect set directly in its Effect property but can inherit other effects from its parents. Imagine applying a BlurEffect to the parent Grid (LayoutRoot) and an **InvertColorEffect** to **StartImage**. **StartImage** would have both effects applied while **EndImage** would only show the blur effect.

The BlurEffect has one interesting property: *Radius*. You can think of Radius as the strength of the blur effect. The higher the Radius value the fuzzier the output.

Here's how to add a **BlurEffect** in XAML.

Example 2-4. Add BlurEffect in XAML

[XML]

```
<Image x:Name='StartImage'
       Source='garden1.jpg'
       Width='500'
       Opacity='1'>
      <Image.Effect>
        <BlurEffect
          Radius='20' />
      </Image.Effect>
    </Image>
```

Of course, you can apply the effect in the code behind too.

Example 2-5. Add BlurEffect in code

[C#]

```
var blur = new System.Windows.Media.Effects.BlurEffect();
blur.Radius = 20;
StartImage.Effect = blur;
```

Now that the effect is applied, run the application. The UI should look similar to Figure 2-7.

Figure 2-7. Image element with blur effect

Are you ready to add the blur effect to the transition? Begin by setting the Radius value to zero for the existing blur effect. Then add a blur to the EndImage (Example 2-6).

Example 2-6. Blur effect for both images

[XML]

```
<Image x:Name='StartImage'
       Source='garden1.jpg'
       Width='500'
       Opacity='1'>
  <Image.Effect>
    <BlurEffect x:Name='StartImageBlur'
                Radius='0' />
  </Image.Effect>
</Image>
<Image x:Name='EndImage'
       Source='garden2.jpg'
       Width='480'
       Opacity='0'>
  <Image.Effect>
    <BlurEffect x:Name='EndImageBlur'
                Radius='0' />
  </Image.Effect>
</Image>
```

Next, write some code to apply the blur gradually as the opacity level changes. Modify the **ValueChanged** event handler as follows:

Example 2-7. Value Changed event procedure code

[C#]

```
// transition the images
var max = TransitionSlider.Maximum;
EndImage.Opacity = e.NewValue / max;
StartImage.Opacity = 1 - EndImage.Opacity;

// opacity is between 0.0 and 1.0
// we want a max blur radius of 20 so will multiply
// by 20
StartImageBlur.Radius = EndImage.Opacity * 20;
EndImageBlur.Radius = StartImage.Opacity * 20;
```

The project is finished. Run the application and watch the transition. As the first image fades away it gets more blurry while the second image fades in and snaps into focus. Nice effect! (pun intended). I'll show you how to create a custom effect soon, but first a word about performance.

Debrief

I have a few quibbles with this code, for one, the performance might be improved by consolidating the effects. There are two blur effects applied to an overlapped area of the screen. If you are seeing perf issues during testing this is an area worthy of further research. To consolidate, you could remove the image effects, wrap the two Images inside another container and apply the blur to the parent container. You can't use the current grid (LayoutRoot), because the blur would alter all children including the slider element. The solution is to add another grid and place the images in the new grid. You'd have to change the transition code too.

Custom Shader

Now that you have some rudimentary experience working with a prebuilt effect, it's time to consider writing a custom one. Custom shaders are necessary when an existing effect doesn't do what you need. Let's say you read an article describing a faster blur algorithm and you decide to alter the **BlurEffect** to use the newer algorithm. If you wait for Microsoft to release a new version, who knows how long you'll have to wait. In this situation, you are better off writing your own effect.

For your first custom shader I picked a simple color modification effect. The shader code is childishly simple, just to give you an overview of the custom shader process. I promise there are more details coming as you read deeper into this book.

There are a few common steps necessary in creating a custom shader.

- Create a text file containing your HLSL code
- Compile the HLSL into a binary file
- Add the binary shader file to a XAML project and mark as project resource
- Create a .NET wrapper class to expose the shader to your project

- Compile the project
- Instantiate your shader and assign to an element Effect property

Create a Shader Algorithm

Crafting a custom shader starts by creating a text file and adding some HLSL code. Example 2-8 shows how to write a simple **InvertColor** shader.

Example 2-8. HLSL code for InvertColor shader

[C#]

```
sampler2D InputTexture;

float4 main(float2 uv : TEXCOORD) : COLOR {
   float4 color = tex2D( InputTexture, uv );
   float4 invertedColor = float4(color.a - color.rgb, color.a);

   return invertedColor;
}
```

There is not much code in this example, but it's sufficient to reverse the color on every pixel in the input stream.

The first line declares the input source, *InputTexture*, for the shader.

[C#]

```
sampler2D InputTexture;
```

In your sample application, this **InputTexture** corresponds to pixels contained in the Image elements. Next is a function declaration.

[C#]

```
float4 main(float2 uv : TEXCOORD) : COLOR {
```

As you can see the function is named *main* and returns a *float4* value. That *float4* represents the color value for the modified pixel, which is destined for the computer screen. You can think of a float4 as containing four values corresponding to the color values (red, green, blue, alpha). The next two lines sample a pixel from the input source and calculate the new output color, which is stored in the *invertedColor* variable.

[C#]

```
float4 color = tex2D( InputTexture, uv );
float4 invertedColor = float4(color.a - color.rgb, color.a);
```

Finally, the inverted color is returned from the function call.

[C#]

```
return invertedColor;
```

Compile the HLSL Code

Next, it is necessary to compile the **InvertColor** shader code into a binary file. By convention, this binary file ends with a .ps extension. There are a number of ways to compile shader source code. For this first walkthrough, you will use the FXC.EXE compiler that ships with the DirectX SDK. If you have the SDK installed you can open a DirectX command prompt from the Windows Start menu as shown in Figure 2-8.

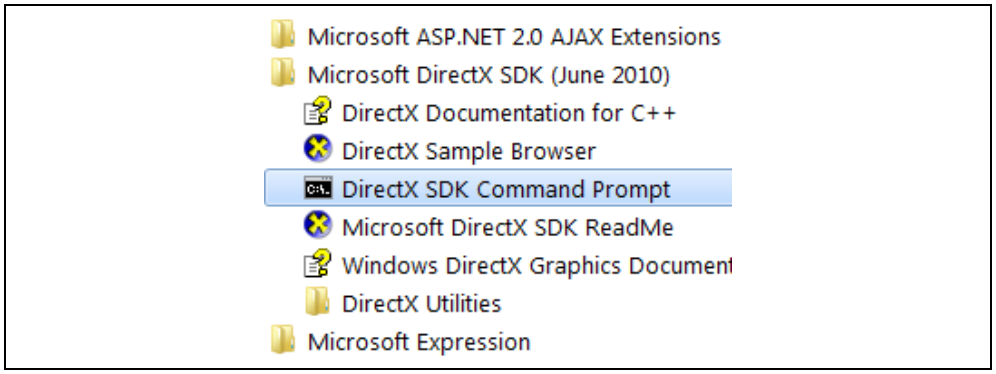

Figure 2-8. The DirectX Command prompt on the Start menu

At the DirectX prompt run the FXC compiler with this entry.

```
fxc /T ps_2_0 /E main /Fo output.ps invertcolor.txt
```

The compiler takes the code in the *invertcolor.txt* file and compiles it into the *output.ps* file. There are various switches specified that tell the compiler to use the *main* function as the shader entry point and to compile with the ps_2_0 shader specification.

Be forewarned, FXE is finicky about encoding types for the input file, it prefers ASCII and doesn't like Unicode encoded text files.

Add to Visual Studio XAML Project

The next step in the process is to add the ps file to a XAML project. Be sure and set the *Build Action* for the file to *Resource* as shown in Figure 2-9 .

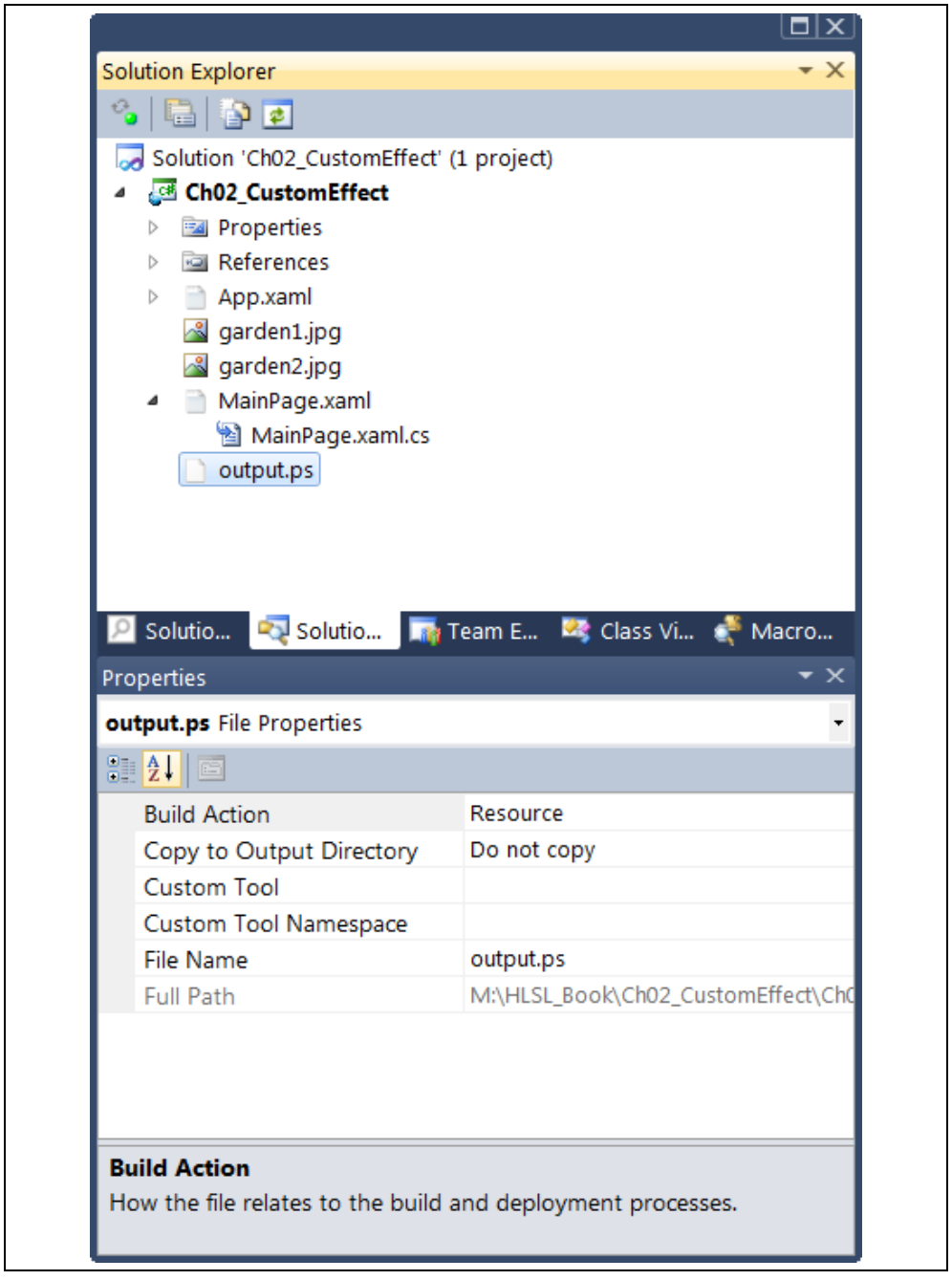

Figure 2-9. Set the Build Action property to Resource

Create a .NET wrapper class

To use a pixel shader you need a way for .NET code to interact with it. The prescribed means to accomplish this is to write a wrapper class. The class derives from the ShaderEffect base class and exposes dependency properties to manipulate shader

properties. The simple **InvertColor** shader doesn't provide any shader properties so the example wrapper class will be small.

Add a class to the project and insert the following code.

Example 2-9. InvertColorEffect wrapper class

[C#]

```
public class InvertColorEffect : ShaderEffect
  {
   private PixelShader pixelShader = new PixelShader();

    public InvertColorEffect() {

      pixelShader.UriSource =
        new Uri("/Ch02_CustomEffect;component/output.ps", UriKind.Relative);
      this.PixelShader = pixelShader;

      this.UpdateShaderValue(InputProperty);
    }

    public static readonly DependencyProperty InputProperty =
      ShaderEffect.RegisterPixelShaderSamplerProperty("Input",
        typeof(InvertColorEffect), 0);

    // represents the InputSource for the shader
    public Brush Input {
      get {
        return ((Brush)(this.GetValue(InputProperty)));
      }
      set {
        this.SetValue(InputProperty, value);
      }
    }
  }
```

You may recall that the .NET wrapper must instruct Silverlight or WPF to load the binary resource. In WPF, the binary is loaded into the GPU, in Silverlight it's loaded into the virtualized GPU. You can see the loader code in the constructor.

[C#]

```
pixelShader.UriSource =
        new Uri("/Ch02_CustomEffect;component/output.ps",
            UriKind.Relative);
```

There is also a single dependency property name **InputProperty**. This property represents the input into the shader. In other words, it's the data provided by the rasterizer. The dependency property follows the XAML convention and looks like you would expect; with one small difference.

[C#]

```
public static readonly DependencyProperty InputProperty =
      ShaderEffect.RegisterPixelShaderSamplerProperty("Input",
        typeof(InvertColorEffect), 0);
```

The *ShaderEffect.RegisterPixelShaderSampler* property is how the dependency property is associated with the pixel shader sampler register. Don't fret too much about the details of this class for now. It's ready to compile.

Compile the project

Build the project and verify that everything compiles.

Instantiate the shader

How do you want add the **InvertColorEffect** to the image? If you want to add it in code, you just instantiate the effect and assign it to the correct element property.

[C#]

```
var invert = new InvertColorEffect();
StartImage.Effect = invert;
```

To add the effect in your XAML file, add a custom XML namespace to the **UserControl**. This xmlns should reference the .NET namespace that contains your wrapper class.

[XML]

```
<UserControl x:Class="Ch02_CustomEffect.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:Ch02_CustomEffect'
             mc:Ignorable="d" >
```

Then apply the effect to the image.

[XML]

```
<Image x:Name='StartImage'
        Source='garden1.jpg'
        Width='500'
        Opacity='1'>
      <Image.Effect>
        <local:InvertColorEffect />
      </Image.Effect>
</Image>
```

Be sure and comment out the code in the **TransitionSlider_ValueChanged** event handler or you will get a runtime error.

Here's what the InvertColorEffect looks like when you run the application.

Figure 2-10. InvertColorEffect applied to image

Summary

Effects are one of the strong points of XAML development. As a former Windows Forms developer, I struggled with the severe limitations of GDI programming and I never want to go back. I embrace the new rendering capabilities in WPF and Silverlight with gusto and appreciate the benefits provide by the custom shader API.

Welcome to the marvelous world of shader development.

Shader Scenario

You are working on a Silverlight project for a large media client. Their lead designer is fascinated with steam punk and insists that the spring campaign reflect his new love. In conjunction with the promotion is a user submitted product video contest. Winners of the best video will take home a cash prize. Your job is to create the video viewer for the contest entries. But there is a requirement that is making your manager anxious. All the videos need to look like movies from the early 1900's. None of the submitted videos have the correct look and there is no budget to for video post-production on the submissions

Shader Solution

The solution, of course, is to create an old movie shader. The shader takes the video output, converts it to a monotone color and tints it an antique brown tone. Next, it apply a vignette mask to the output. Vignette is a photography term that describes an effect often seen in wedding shots and other glamour pictures. The corners of the picture are tinted slightly darker than the center of the image.

Just apply the old movie shader to the video player output and you are done.

3

Commonplace Pixel Shaders

"What is an example of a real world pixel shader?" That's a question I hear all the time when teaching the concepts of shaders to XAML programmers. The trouble, as I see it, is that most computer users wouldn't recognize a pixel shader if they saw one. Make no mistake, they are out there — pixel shaders are found in a wide range of software products. Most consumers happily use them without knowing that they are working with a feature that was implemented with an effect.

Certain application categories are prime candidates for pixel shaders. The game development world is one obvious example. Image manipulation software is another fertile area for shaders. Academics have studied the problem of image filtering and pixel manipulation for decades. All modern image processing software, think Photoshop or Paintshop Pro, have ranks of shaders hiding behind the application facade.

Common effects include blurring, distorting, image enhancement and color blend. This chapter provides an overview of the types of effects that are common in the shader realm.

> Shaders are often applied after the render phase, so are also known as post-processing effects.

Pixel shaders fall into a few general categories.

- Color modification / Color transformation
- Displacement / Spatial transformation
- Blurs
- Generative / Procedural
- Multiple Inputs

A Word about Pseudocode

A couple words before diving into the shader descriptions. It is still early in the book and you haven't seen a lot of HLSL syntax. With that in mind most of the code listed in this

chapter is pseudocode which is useful for explaining the general idea behind a shader implementation but doesn't use the official HLSL syntax. Here is a simple example.

Example 3-1. Pseudocode and HLSL compared

```
// pseudocode
originalColor = color.r
average  = (color.r + color.b + color.g)/3

// HLSL syntax
float4 originalColor = tex2D(InputTexture, uv);
originalColor = color.r ;

float average;
average = color.rgb/3;
```

The HLSL in this example is more precise than the pseudocode version of that same code. First, it gets the color using the text2D function. Then, it defines the variables with the *float4* and *float* keywords. Leaving the last line to employ the HLSL **swizzle** syntax to calculate the average.

> Swizzling, in HLSL, provides a unique way to access members of a vector. While the word conjures images of fruity drinks around a tropical pool, it's just a way to perform operations on any combination of vector items.

Our Sample Image

For many of the examples in this chapter, I will use the color photo you see in Figure 3-1. I picked this photo because it is colorful and has a nice composition. Furthermore, it has a black background which works well with some of the color replacement shaders.

Figure 3-1. Flower photo with no effect applied

Color Modification

Color modification is a simple concept. Take an incoming pixel color, and shift that color according to some formula. What's key to understanding this category of pixel shaders is that the basic image structure stays the same. The algorithm doesn't move or reposition pixels on the screen. Take a picture of a cat for example; you'll know that it’s a cat, even though the colors are translated into shades of purple and blue.

Common Techniques

Color correction is one use of color modification that is indispensable to the film and print industry. Most of the images you see on television or in a magazine have been color corrected during post-production. Whether it was to adjust contrast on a marginal exposure, create unified skin tones for a fashion spread or to apply a gloomy color palette to a gothic novel poster, most commercial imagery undergoes color alteration before it is released for public consumption. Color corrections largely come in two flavors: per-channel corrections, which alter red, green, and blue components independently; and color-blending procedures, in which the output value is determined based on combined values in the RGB channels.

Stylizing a photo is another popular color modification technique such as utilizing sepia tone and tinting, introducing sixties-era colors, making an image look like it’s from an old movie, inverting colors, or using color channels.

Color removal is another commonplace technique. Grayscale and desaturation are popular methods that fall into this category.

Photo editing applications are rife with tools in this category. Open your copy of Adobe Photoshop and look in the Image→Adjustments menu (See Figure *3-2*). It's packed with color adjustment tools and it's likely that a pixel shader or two is employed behind the scenes to do the real work.

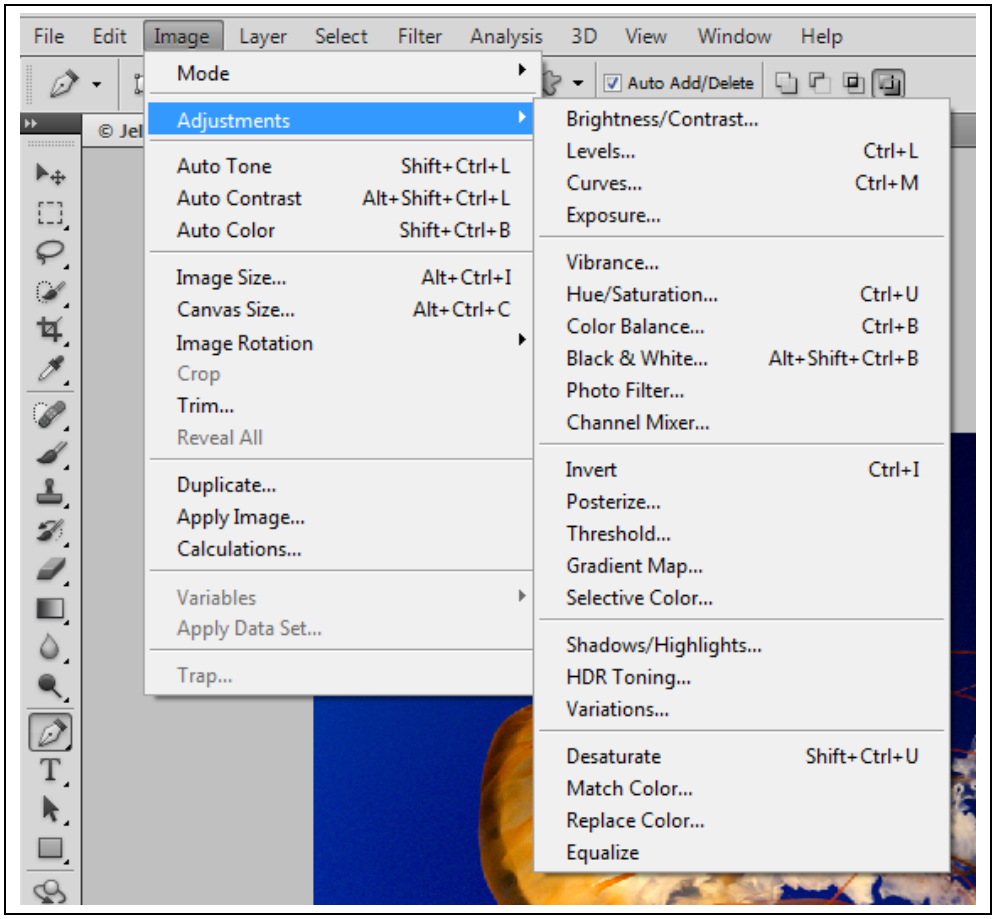

Figure 3-2. Photoshop Adjustments Menu

Black, White and Gray

All the shaders in this subsection remove the natural colors from the image and replace them with a monotone value. Creating a grayscale image, is accomplished by setting a color value that consists of equal parts of the red, green and blue channels.

It's likely that you know this already; nevertheless, a quick review is in order. A pixel with each RGB color set to zero results in a black pixel.

```
newColor = color.rgb(0,0,0); // pseudocode
```

When each RGB value is set to one the output renders as a white pixel.

```
newColor = color.rgb(1,1,1); // pseudocode
```

Whenever all three RGB values are set to the same value, a gray pixel is produced.

```
newColor = color.rgb(0.2, 0.2, 0.2); // pseudocode
```

HLSL shaders normalize color values to the range 0-1, as you can see in the following example (Example 3-2).

Example 3-2. Setting grayscale values

```
// pseudocode
RGB(0, 0, 0); // black
RGB(1, 1, 1); // white
RGB(0.8, 0.8, 0.8); // light gray
RGB(0.2, 0.2, 0.2); // dark gray
```

Black-White

You've seen black and white photos in fine art magazines or your local gallery. In most cases, they are not pure black and white though; they are composed of multiples shades of gray. True black and white is rare but is occasionally used for dramatic special effect.

Technique: This shader works by setting all pixels over a certain threshold to white and the rest of the pixels to black.

Example 3-3. Black-White pseudocode

```
if (color.r > .5) // use the red channel for threshold trigger
{ RGB(1,1,1);}
else
{ RGB(0,0,0);}
```

Check out Figure 3-3, a reductionist version of our original image created with a shader using the black/white technique.

Figure 3-3. Black and White effect

Grayscale

Grayscale images appear to have no color, only tones of gray. Other common names for this image type are monochromatic and monotone.

Technique: The shader must examine the pixel and decide what value to assign to the RGB channels. One approach is to use a single channel at the filter (See Figures 3-4 (a) and (b)). For example, take the red value and assign to the green and blue channels.

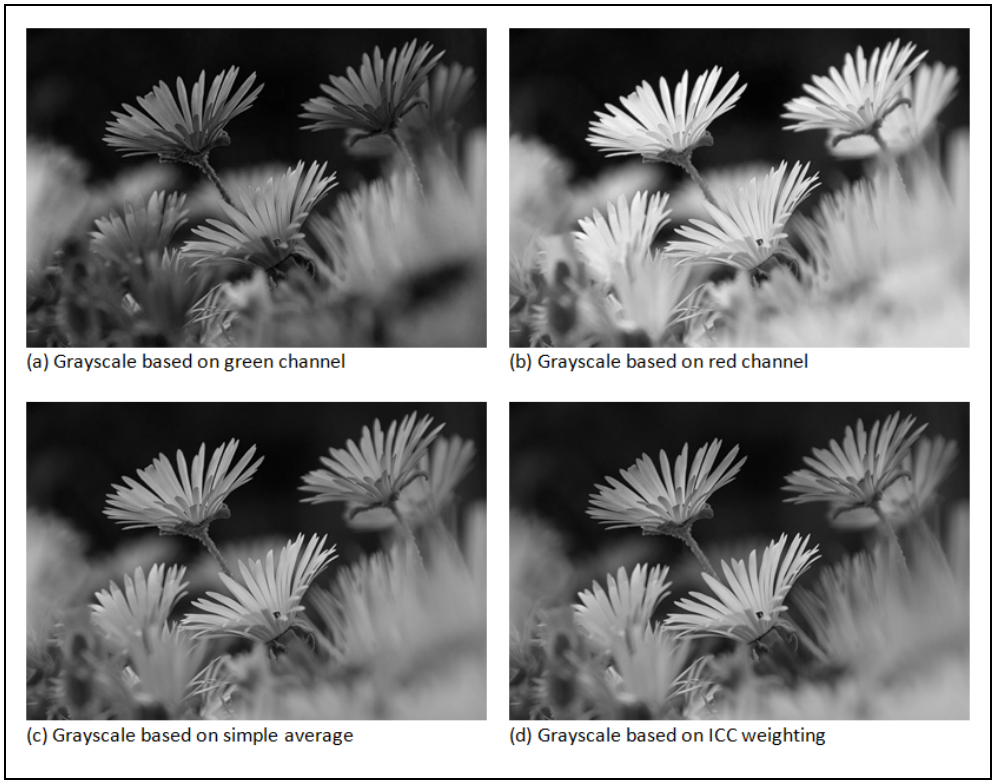

Figure 3-4. Grayscale effect, based on different algorithms

Another approach is to average the colors channels and apply the average to each RGB value (see Figure 3-4(c)). For the most realistic approach however, don't use a simple average calculation. The lab-coated masterminds at ICC provide a nice weighting formula that makes your grayscale feel authentic (see Figure 3-4(d)).

```
// pseudocode
gray = RGB(inputColor * 0.21, inputColor * 0.71, inputColor * 0.07)
```

> The human eye is more sensitive to the green spectrum than other colors. Using a grayscale formula that enhances greens results in an image that looks closer to a picture taken with black and white film.

Color Replacement

Effects in this category seek out a color and substitute another in its place. These effects usually have a color sensitivity dial, which tweaks how precisely you wish to match the target color.

Other names for these effects are ColorKey and ChromaKey. In the samples included with this book is one called ColorKeyAlpha. It transforms any pixel of the target color to a transparent value (by setting the Alpha channel to zero).

All sci-fi and action movie buffs are familiar with green screen effects, whereby the actors work in front of a dazzling lime green screen and the bellowing velociraptor is added to the scene in post-production. A ChromaKey effect is a form of color replacement. Because the replacement data is not a simple color, but a complex image ChromaKey can also be categorized as a multi-input effect. That's because the effect relies on having at least two samples passed into the shader; the foreground image where the actors live and the background image where the dinosaurs roam.

Color Enhancement and Correction

How many times have you taken a picture indoors and been dismayed to see a blue cast on the finished picture. Fluorescent lights are to blame, as they use an unnatural light spectrum. Professional photographers understand the nuances of light and learn how to exploit it for their craft. But even they occasionally make mistakes during photo shoots and need to correct the image in post-production.

Color enhancement and correction can be achieved by applying one or several of the following techniques:

- Gloom
- Hue
- Saturation
- Contrast
- Brightness
- Bright Extract
- Bloom
- Color Balance
- Vibrance
- Thresholding
- LightStreak
- Tone Mapping

If you are familiar with image processing techniques you will recognize some of the names in this list and perhaps know what they do when applied to a picture. Other terms, like Gloom, may be unfamiliar. In case you are curious, the Gloom effect intensifies the dark areas of the image

This chapter mentions a lot of shader effects by name but there isn't enough room to provide a comprehensive catalog of every effect cited. They are all included in Shazzam; I encourage you to try them for yourself.

Distinctive Effects

Sometimes you just want to do something wacky to the output, like make the image look frosty or apply a cartoon color palette. Shaders can handle the job with ease.

Some examples are:

- Frosty Outline
- Old Movie
- Vignette
- ParametricEdgeDetection
- Tinting
- Color Tone
- Pixilation
- Sketch
- Pencil
- Toon

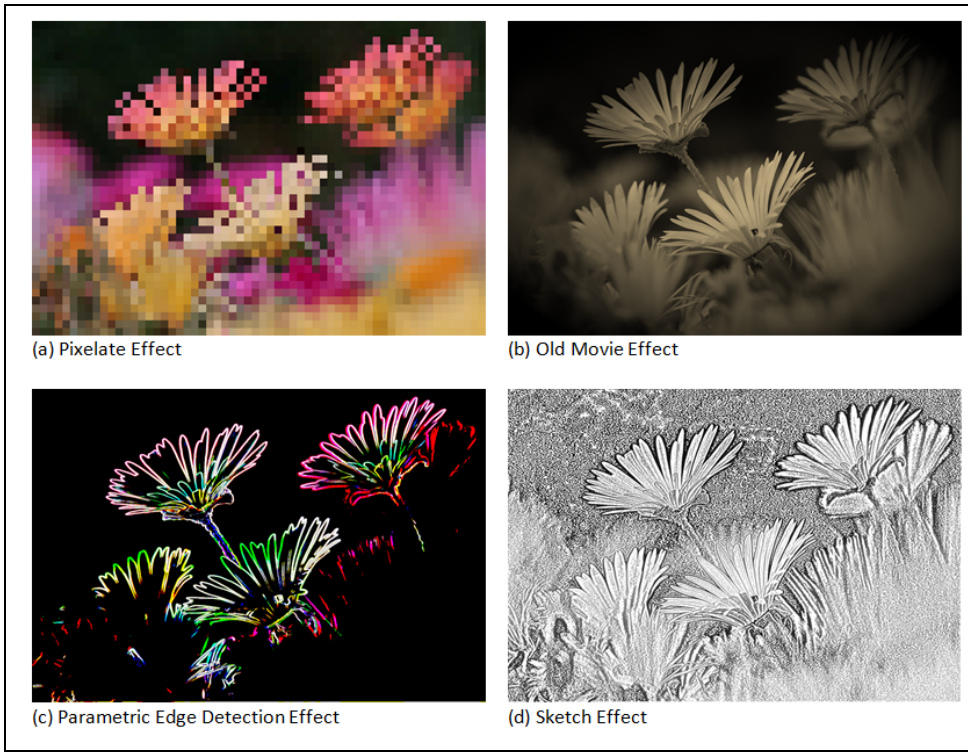

Figure 3-5. Shader Effects

The Pixelate effect (Figure 3-5(a)) is used to simulate a close-up of an image, causing it to appear coarser, blockier and less detailed. The Old Movie effect (Figure 3-5(b)) is a combination effect. It applies a sepia tone to the image. It also uses a vignette effect, whereby the pixels closer to the image perimeter are darkened. The Parametric Edge

Detection effect (Figure 3-5(c)) uses a convolution algorithm to detect edges of items within the image. In this version it colors the edges with vibrant colors to make them stand out. The last effect (Figure 3-5(d)) transforms the image into a version that looks like it was hand sketched with a pencil.

Distortion and Displacement

Shaders in this category are less subtle than the color modification examples. They aggressively move pixels around the screen, causing bulges and ripples to appear on an otherwise flat output. Generally, the objective is to trick the user's brain into seeing a real-world distortion where none exists.

The techniques used are diverse but all have one thing in common. When determining the color to use for an output pixel, they don't use the color from its current position. They borrow the color from a neighboring pixel or interpolate a new output color based calculations run on nearby locations.

> Be attentive when looking at HLSL code on the Internet, as a good portion of your search results are likely vertex shader examples. This is particularly true when researching displacement shaders and lighting shaders. As you may recall, vertex shaders are a companion type of shader that are part of the DirectX input pipeline. Whereas pixel shaders change pixel colors, a vertex shader interacts with the 3D model vertex points. Typically, a vertex shader precedes a pixel shader in the shader pipeline. Many Internet examples consist of a mixture of the two with the majority of the work implemented in the vertex portion, which does us no good as we can only work with the pixel shader portion of the pipeline.

The terms applied to this technique are varied: distortion shaders, displacement mapping, per-pixel displacement, relief mapping, spatial transformation and parallax mapping are some of the more common names. Let's look at some of these effects.

Magnify

The magnify effect enlarges a subset of the image, zooming the user in for a closer look. The enlarged area is often elliptical, though I've seen other implementations.

Figure 3-6. Two implementations of a magnify effect

Figure 3-6 shows the flower image, with two different magnification effects applied. The effect on the left has a hard edge to the enlargement area, while the one on the right uses a smoothing algorithm. The smoothing makes it appear more natural and glass-like. You may also see this called a bulge effect.

Embossed

The embossed shader is a simple displacement shader. The photo processing community calls this the inset effect. It provides a mono-color 3D appearance to the image. This is accomplished by making two separate monochrome versions of the image and making the first copy lighter in color than the second copy. Then the images are displaced a few pixels away from each other (See Figure 3-7(a)). For example, the light version is moved two pixels to the left, the dark version two pixels to the right.

Figure 3-7. More Distortion Effects

Figure 3-7 presents a few more distortion effects. The Bands effect (Figure 3-7(b)) appears to draw vertical blinds over the image. For a glassy tiled look, chose the Glass Tiles effect (Figure 3-7 (c)) and the Ripple effect (Figure 3-7 (d)) provides the ubiquitous and overused water droplet effect.

Testing distortion effects

I love the colors and composition of the flower image but it is a poor choice for testing certain distortion effects. However, it works well for testing the following mirror effect (See Figure 3-8).

Figure 3-8. Mirror effect

For effects like bulge or pinch it's better to test with a hard edge geometric pattern — I find a checkerboard pattern works best. Figure 3-9 shows some sample effects applied to a black and white checkerboard.

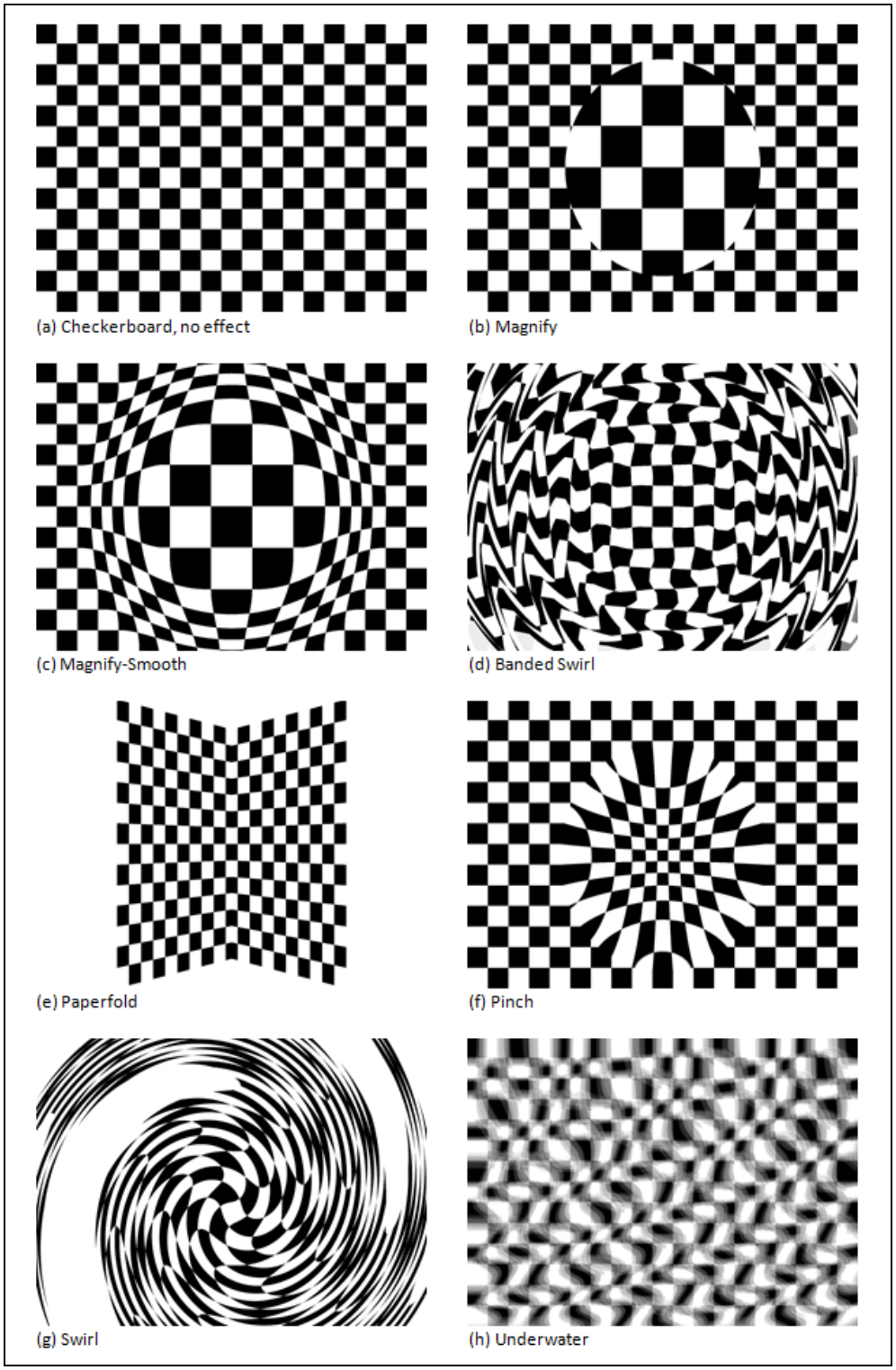

Figure 3-9. Checkerboard with various distortion effects

Other Displacement Effects

Displacement shaders are a fertile area of development. Here is a listing of some of the more interesting effects in this category. You can see a few examples in Figure 3-9.

Just a few displacement shaders are:

- Paperfold
- Pivot
- Pixelate
- Splinter
- Camera Lens Correction
- Bulge
- Pinch
- Ripples
- Swirl
- Banded Swirl
- Bands
- Tiles
- Glass Tiles
- Mirror

Once again, I must omit the detailed descriptions of each of these effects. There isn't enough room in this book to cover them all.

Blurs

Blurs are type of a displacement effect. Rather than being dramatic, like a ripple effect, they tend to be subtler and affect a smaller area. Some of the common blurs are named after the technique used to create the blur. The Gaussian blur and GrowablePoissonDisk effect fall into this bucket.

> The Gaussian and Poisson algorithms are named after their inventor's surnames, Johann Carl Friedrich Gauss and Siméon Denis Poisson

Both Silverlight and WPF have a built-in blur effect. While it is serviceable, you might consider some of the alternative blur algorithms listed below for parts of your interface.

Motion blur

Motion blur is the apparent streaking of quickly moving objects in a photo or video frame. It is an artifact of camera hardware limitations and appears when to trying to capture a moving object while using a too-slow shutter speed. The graphics industry makes frequent use of this handy effect during post-production work. You'll find it used in computer graphics, video games, traditional and 3D animation and movie special effects. This effect is also called a directional blur.

Zoom blur

This blur mimics the action of taking a picture while zooming the camera lens closer to the subject. This effect is also known as the telescopic blur.

Sharpening

This effect attempts to remove blurriness from the source image by increasing the edge contrast ratio (a well-known process for changing perceived image sharpness). This effect is also called the unsharp mask effect.

Generative Effects

Most of the time your effect is dependent on the inbound raster image. There are times when it is advantageous for an effect to ignore the inbound pixels and generate its own independent output as in Figure 3-10. Choose this route when the effect generated can benefit from the multi-core and parallelization power of the Graphics Processing Unit (GPU). There are examples found across the Internet showing blazing fast fractal generators. Fractal algorithms are a dead-end for the XAML developer however. We are stuck with the PS_2 and PS_3 specification, which has an inadequate number of instructions available for recursive functions. These limits mean that you can create a Mandelbrot quickly, but it will be a superficial portrayal of the beauty of fractals.

> Each new version of the Pixel Shader specification provides significant upgrades over the previous version. To write HLSL code to exploit the newest PS specification (PS_5_0) you must use the DirectX API itself as there is no support for it in Silverlight/WPF. Silverlight supports the PS_2_0 spec and WPF supports the PS_2_0 and PS_3_0 versions. To learn more about the difference in these specifications try this website. http://en.wikipedia.org/wiki/HLSL

Even though robust fractal patterns are out of the question, you can generate gradients and fancy patterns.

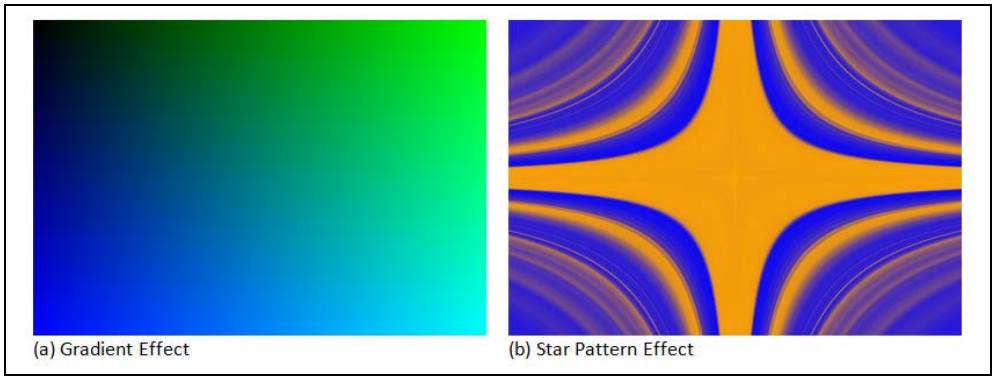

Figure 3-10. Generative effects

Multiple Inputs

Most of the effects seen in this chapter have been single input. To be more accurate they have a single sampler input. Samplers, as you may recall, are the inbound pixels passed to the shader from the DirectX pipeline. You don't have to be clubbed with the obvious stick to recognize that if a shader has no input samples it doesn't have much to work with.

> The generative shaders are the only useful effect that ignore input samplers.

The PS_2 and PS_3 specifications allow additional input samplers-- up to a maximum of four in PS_2 and a maximum of eight in PS_3. You'll learn more about input samplers later in the book.

What can you do with multiple inputs? Here's a short list, though there are many more ideas to explore on your own.

- Sampler Transition
- Color Combination
- Texture Map
- Photoshop Blend modes

Sampler Transition

The last decade has seen pervasive changes in UI metaphors. Touch input is without a doubt the most transformative shift I've seen since starting my tech career. Another trendy change is transitional animation, whereby the user is moved from one state to another though a helpful animation. In the XAML world this is frequently done with a storyboard. Let's envision a fade animation that transitions from a list of products to a products detail page. The accepted way to accomplish this in XAML is to layer the list UI over the detail UI and animate the opacity value of the top layer.

That's one way to accomplish a UI transition. Let's consider how you could implement this transition as a multi-input shader. You'd provide two inputs to the shader, on for each UI section. In the HLSL, you take the two inbound samplers and output one result. That result is either the first sampler, the second sampler or some transitional value between the two. In your .NET effect class provide a Progress dependency property. Then animate the Progress property and let the shader provide the fancy transition effect. Figure 3-11 shows a few stages of a transition effect implement in this manner.

Figure 3-11. Multi-input shader, showing four stages of transition

Texture Map

Texture mapping takes the values stored in one input texture and uses them to manipulate other input textures. Let's look at an example that uses the values in the a geometric pattern image to manufacture a glass like embossing effect on the flowers image.

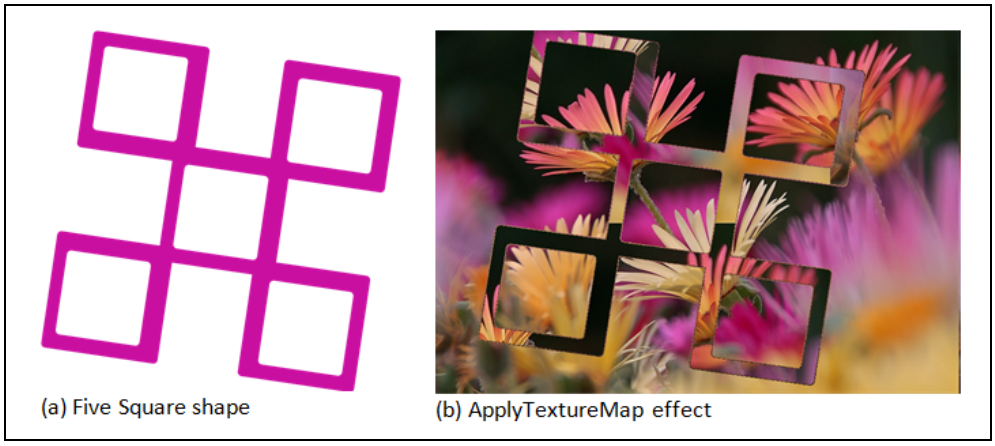

Figure 3-12. Shape and final output for ApplyTextureMap effect

In Figure 3-12 example, the Texture map is the five squares shown in Figure 3-12(a). The effect combines the purple pixels of the texture map with the flower pixels from the other image to create a glass like texture overlay (Figure 3-12(b)).

Mapping can get more sophisticated, you could use the colors in the map file to recolor the colors in the target image. For this example I've created a map file containing bands of colors.

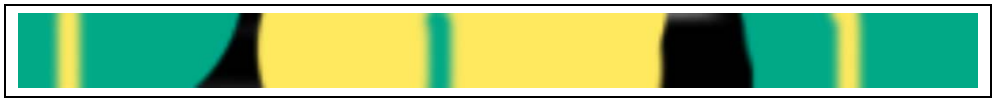

Figure 3-13. The Source map file

Start on the left edge of Figure 3-13 and look at the first pixel on the top row. What color do you see? I see a lustrous emerald green. Continue moving right on the top row of pixels and you see green pixels for a short stretch, then a few yellow pixels, then green again, followed by a strip of black. The colors change as they move rightward but stay limited to the three colors (green, yellow and black).

There are exactly 256 pixels in that first row and they are going to serve as a lookup table for a shader.

> Often times a map file is only one pixel high by 256 pixels wide. It might help to think of it as a one-dimensional array with 256 elements. The image in Figure 3-13 is actually 20 pixels high. That extra height serves no purpose for the shader, but it does make it easier to see the picture in a book!

Some of you are thinking ahead and know why there are 256 pixels in the first row. Yes, that's the number of unique gray values in an 8-bit grayscale color range. Now if you replace each pixel in a grayscale image with a lookup color from the color map you are performing a texture mapping. Black is replaced with pixel[0], which on our texture map is green. White is replaced with pixel[255] which is also green. The other 254 gray values use the lookup color found at their respective location. Figure 3-14 shows before and after versions of three images with the color map applied.

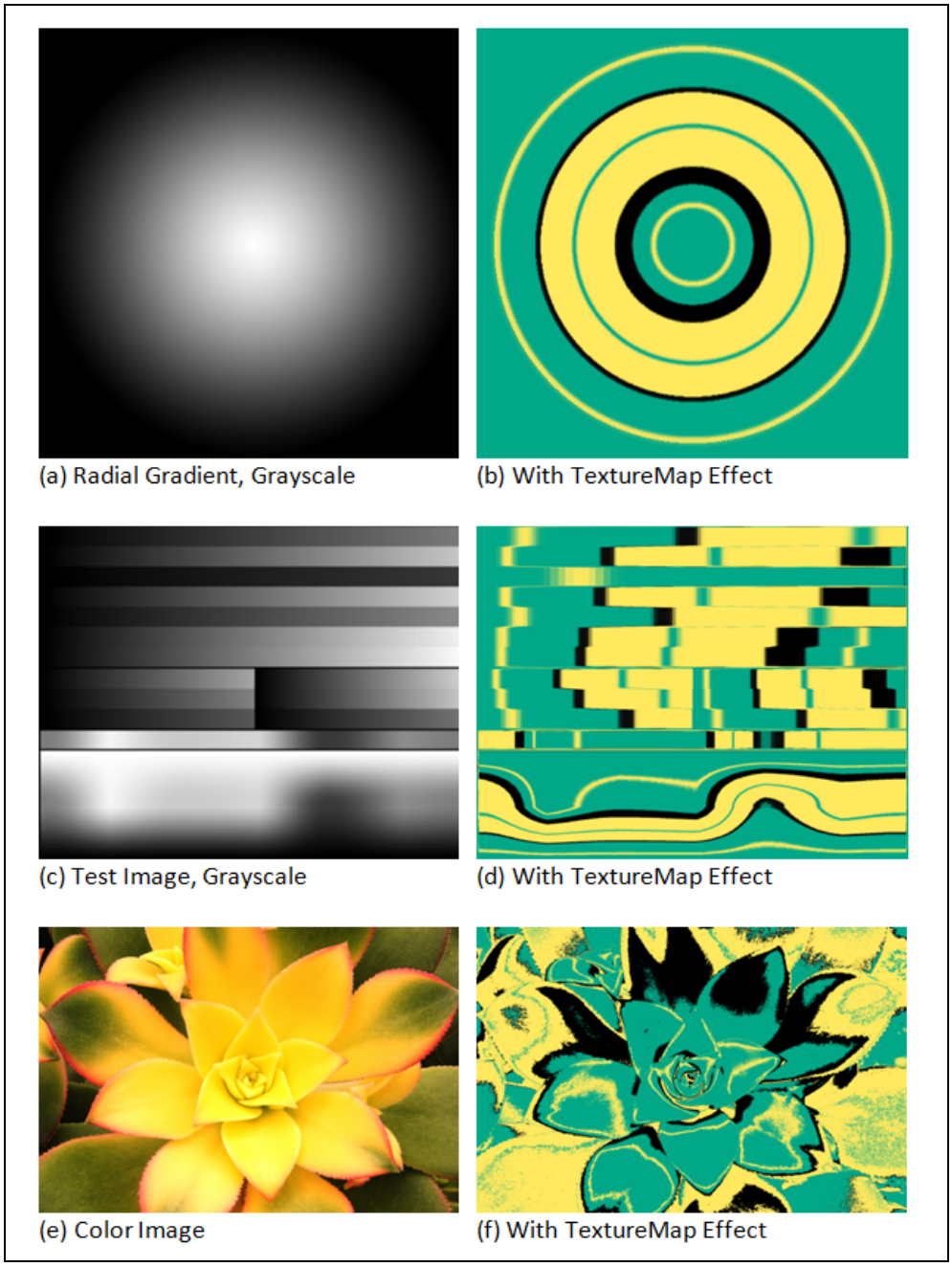

Figure 3-14. Three images with texture map applied

Wait a minute. The picture of the flower on the last row in Figure 3-12 isn't a grayscale. How does that mapping work? It's quite simple if you recall the earlier discussion on creating monochromatic shaders. The shader code converts the pixels to grayscale values before applying the texture map.

Examples, and source code for of each of the shader types discussed in this chapter can be found in the Shazzam Shader Editor.

Color Combination

Color combining is straightforward. Take a pixel from the same location in multiple sources and combine into a new color. The color algorithm here is the key. For an overtly simple algorithm just average the RGBA channels across all input sources.

Example 3-4. Sample color combination shader

```
// pseudocode

color1 = GetColor(sourceA);
color2 = GetColor(sourceB);

combinedColor.r = (color1.r + color2.r) /2;
combinedColor.g = (color1.g + color2.g) /2;
combinedColor.b = (color1.b + color2.b) /2;
combinedColor.a = (color1.a + color2.a) /2;
```

This works but produces a low contrast and muddy output. A more desirable approach is to determine what kind of color combination is best for the intended image effect. Luckily, there is a set of well-tested and respected formulas available. I'm referring to the Adobe Photoshop blend-mode algorithms. You see, the Photoshop developers have been thinking about the problem for a long time. As a result, their color blend-mode implementation is top-notch.

In many situations, your input samplers will not have the same dimensions. In that case, there won't be a one-to-one relationship between pixels. WPF and Silverlight handle this situation by enlarging the smaller source to match the larger.

Photoshop Blend Modes

Adobe Photoshop is considered by most design shops to be 'the' premier photo-editing tool and knowing how to exploit its toolset is a considered a badge of honor in the designer community. If you ask me to name the single most important feature in Photoshop, I would vote for the layers feature. Without layers, you'd be hard pressed to make editable parts of your image and get any work done.

When you have multiple layers in a Photoshop project, you can configure how the pixels in an upper layer combine with the pixels on the next layer down the stack. This feature is called *Blending Mode*. On a many-layered project conceptualizing each layer and its blend can get complex so for this discussion I'll assume that there are only two layers.

Before explaining the modes, it's helpful to define the three colors as defined in Photoshop documentation. The lower layer contains the *base color*, the upper layer contains the *blend color* and the output color is termed the *result color*.

The current version of Photoshop boasts over twenty blend modes (see Figure 3-15) and each one uses a different formula to blend colors between layers. You can read details about each mode on the Adobe website (http://adobe.ly/cs5blendmodes).

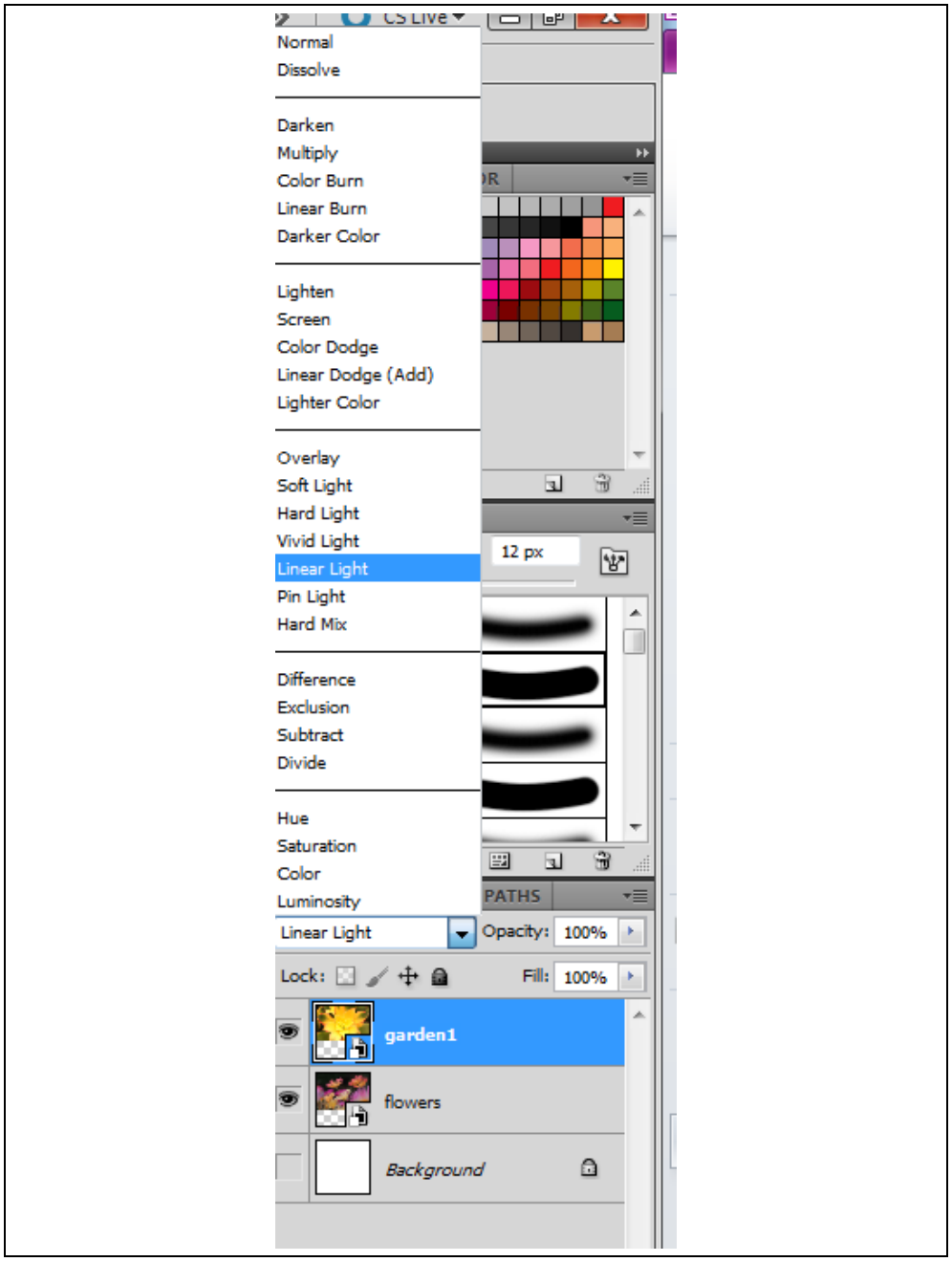

Figure 3-15. Photoshop blend modes

Darken modes

Each of these modes results in of darkening the base image.

- Darken

- Multiply
- Color Burn
- Linear Burn

I'll examine the Darken blend mode here. A detailed discussion of all thirty modes is beyond the scope of this book so I encourage you to learn more about the modes at the Adobe site.

The darken mode examines the base and blend colors for each pixel. The darker of the two is selected as the result color of that pixel. Figure 3-16 show the results from using the four darken effects.

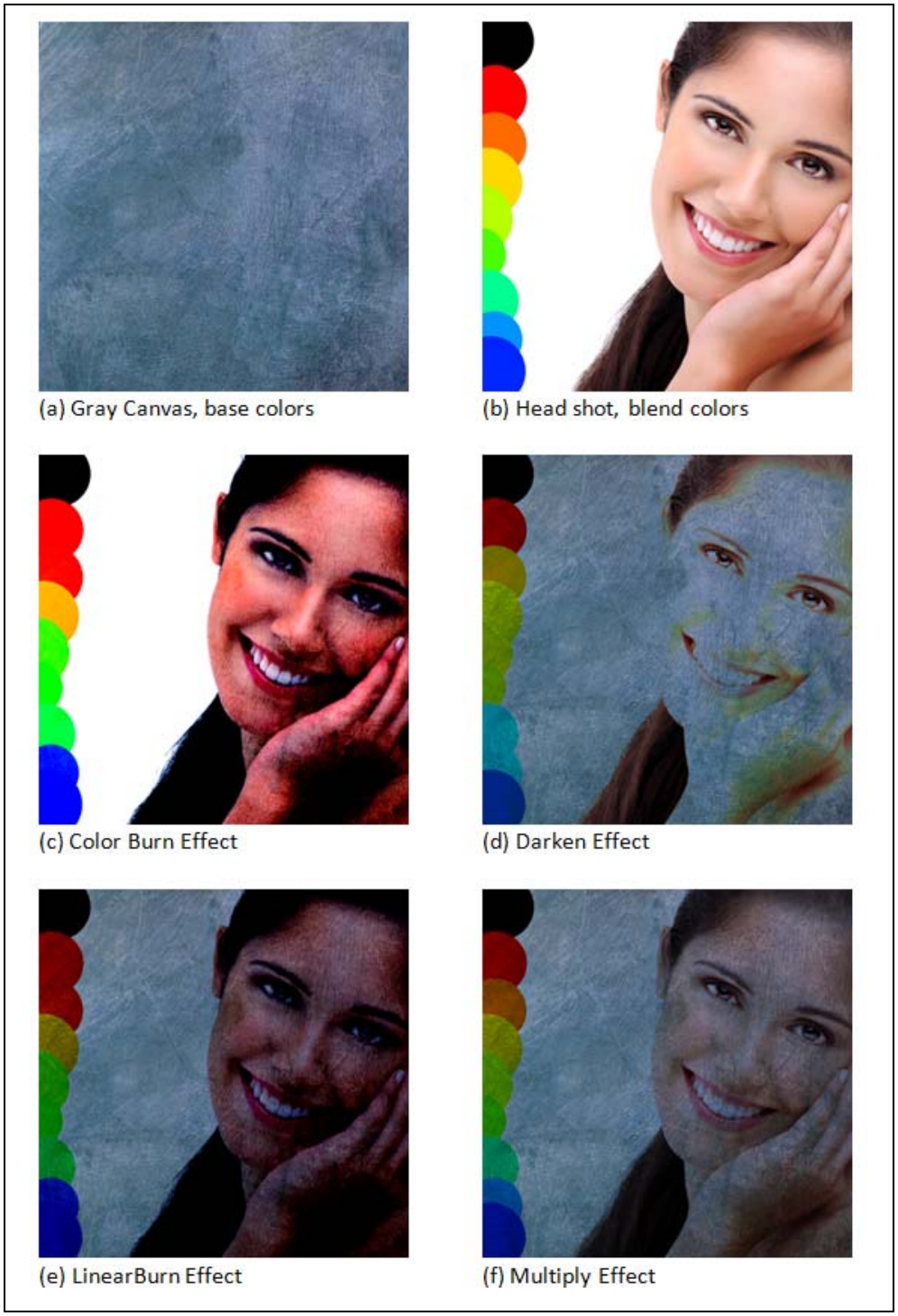

Figure 3-16. Darken blend modes

Lighten modes

Each of these modes results in a lightening of the base image.

- Lighten
- Color Dodge
- Screen
- Linear Dodge

Contrast Modes

Each of the contrast modes lightens some pixels and darkens others in the base image, heightening the contrast.

- Hard Light
- Hard Mix
- Linear Light
- Overlay
- Pin Light
- Soft Light
- Vivid Light

Comparative Modes

Each of these modes compares the two layers, looking for regions that are identical in both.

- Difference
- Exclusion

Other Modes

Each of these modes uses a unique formula to blend the pixels. Some of these are addendums to the Photoshop blend algorithms and were provided by third party tools or community members.

- Glow
- Negation
- Phoenix
- Reflect

Figure 3-17 shows a potpourri of blend effects.

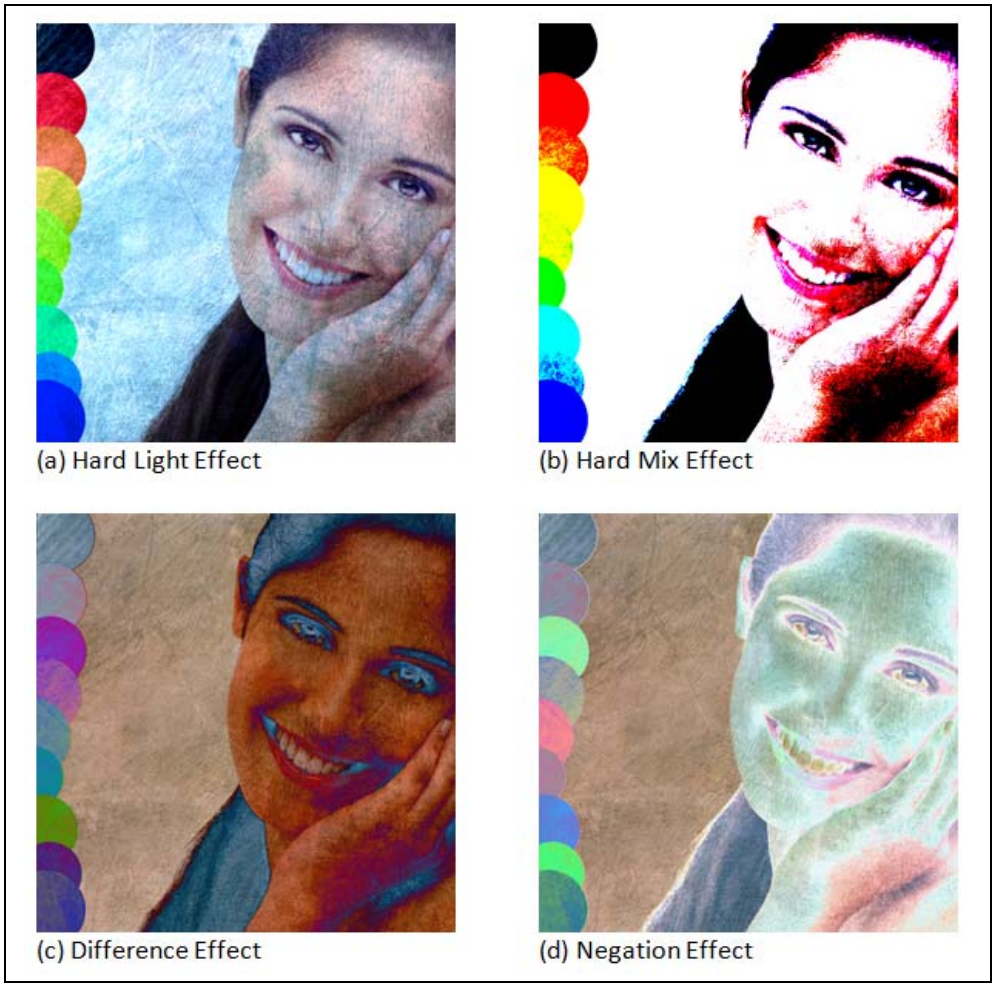

Figure 3-17. Sample of other blend effects

Blend modes in HLSL

Recreating these blend modes in HLSL is a productive exercise and provides a great example of the usefulness of multiple input shaders. My favorite implementation of blend modes comes from Cory Plotts. His original library is available at http://www.cplotts.com/2009/06/16/blend-modes-part-i/ and they are included in the latest Shazzam release.

Practical uses for shader effects

It's a good bet that you are considering using custom effects in you application, otherwise you'd not be reading this book. Effects are exciting and an obvious choice for certain families of applications (I'm thinking of the photo editors and video tools among other "fancy" UX applications).

Still, I suspect some readers are wondering about practical examples in business applications so here are a few ideas to consider before you move into the next chapter.

Motion blur in moving parts

Think about the parts of your UI that move. An obvious place to look is where you have animated elements. Adding a motion blur to the element while it is moving across the screen can make it seem livelier. It's done all the time in the movie and cartoon industry and it's worth considering in your application, too. Don't forget to consider other non-animated areas like listboxes and scrolling areas as candidates for motion blurs.

Blurs to emphasize UI focus

You can use a blur to focus attention toward a section of the UI. For example, when you show a Popup or Dialog in your application apply a blur to the UI in the background. This will draw more attention to the dialog as the rest of the UI is blurry and the background appears to fade into the distance. Note that this technique works best when the main UI is maximized. I've used this technique for the dialogs in Shazzam Shader Editor. Check out Figure 3-18 to see a telescopic blur in action.

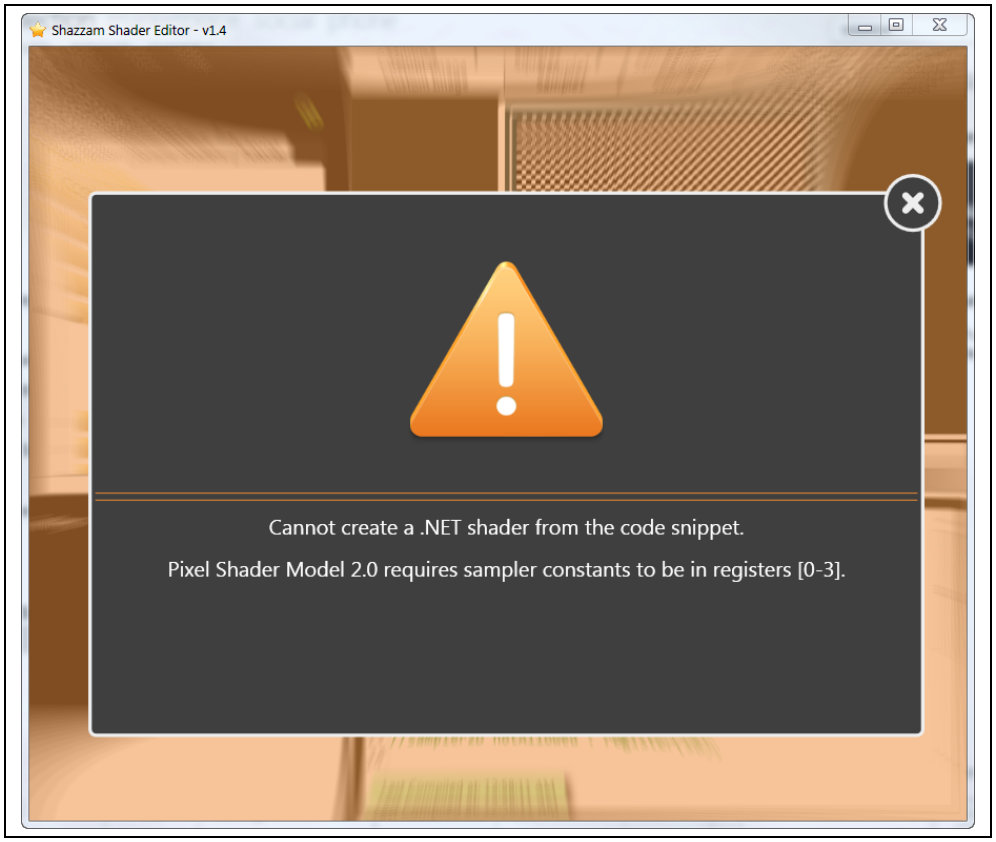

Figure 3-18. Using blur for dialog background

Desaturate for emphasis

The Windows OS used this technique a few years back. When you switched from one OS theme another, the UI would desaturate slowly to gray, then saturate the colors back to your new theme. You could do a similar stunt in your application.

For example, reimagine the dialog shown in Figure 3-18 with a desaturated background, instead of the blurred one.

Summary

Creating a UI that is informative, yet delightful, is a delicate balancing act. On the practical side, you need input controls and access to the business data or the UI is just a pretty plaything. On the designer side, you want to follow time-tested design principles, choose beautiful typefaces and create an inspiring layout for the interface. Shaders are an enhancement to the designer mindset and provide unique tools for enriching an application. Be cautious when using bold effects or you might find your users are nauseated instead of elated with your rippling, oversaturated workspace. On the Silverlight side, you also need to consider the performance ramifications for shaders. Because they run on the CPU, Silverlight shaders are not as fast as their WPF brethren.

Shaders speak to my artistic side; I love the power and beauty inherent in these potent graphical nuggets. When I look at a shader example, my mind's eye sees the UI potential hiding in each effect. I'm sure you felt the tingle of inspiration while thumbing through the assortment of shader effects in this chapter. Now that you know what shaders can do, let's see how to use them in your Silverlight or WPF application.

4

How WPF and Silverlight Use Shaders

You can spend your programming days happily working within the comforting confines of .NET's managed code libraries without ever seeing a smidgen of unmanaged code. The framework team is not stupid though, they know there are times when you have to call out to a COM library or Win32 DLL to get your job done. So they created hooks in the framework to enable the flow of code between the sheltered world of managed code and the mysterious unmanaged realm. It's the same story when interoping between HLSL code and Silverlight/WPF classes.

In this chapter, we look at the .NET parts that facilitate the use of unmanaged HLSL shaders in the visual tree. The **UIElement.Effect** property is our first stop. It provides a way to assign a **ShaderEffect** to a visual element. Next, we look at some of the classes in the **System.Windows.Media.Effects** namespace. These classes (**ShaderEffect**, **PixelShader,** etc.) enable the flow of information to the HLSL world. We'll examine how to create your own managed wrappers for HLSL and investigate the prebuilt effects in the **System.Windows.Media.Effects** namespace and the Expression Blend libraries.

> Remember, on the .NET side the customary term is effect, on the HLSL side the preferred term is shader.

Framework Effects

It's easiest to start our discussion of framework effects by looking at the two shaders included in the **System.Windows.Media.Effects** namespace (see Figure 4-1). By starting with the **BlurEffect** and **DropShadowEffect** we can concentrate on the XAML syntax and not worry about custom classes and resource management.

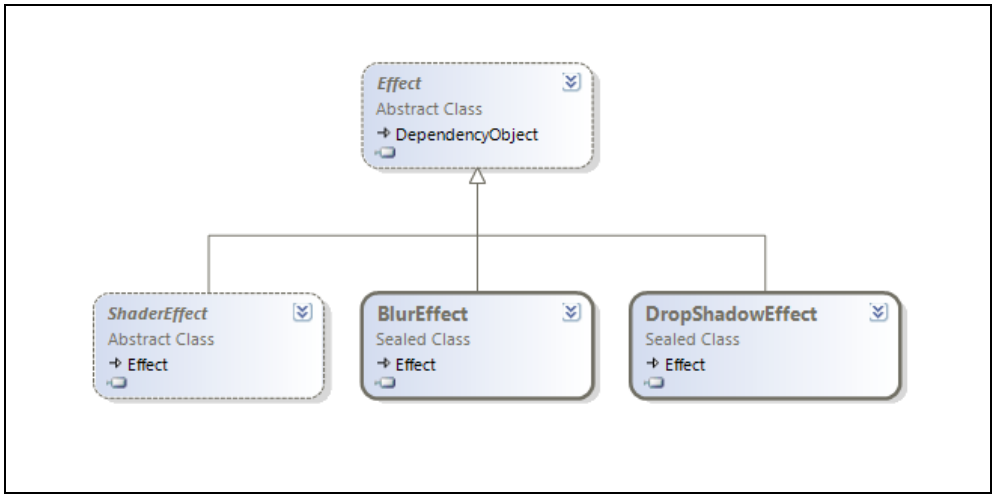

Figure 4-1. Effect classes included in the framework

All visual elements derive from the **UIElement** class, which makes it an ideal location to surface the **Effect** property. With a few lines of XAML you can apply an effect to any **UIElement**, as shown in Example 4-1.

[XML]

Example 4-1. Applying BlurEffect to Image element

```
...
<TextBlock Opacity='1'>
  <Image.Effect>
    <BlurEffect Radius='12' />
  </Image.Effect>
 </Image>
...
```

BlurEffect

In an earlier chapter, I showed how to use the **BlurEffect**. It is one of the simpler effects. It applies a blur algorithm to the output, resulting in; you guessed it, a blurry output. The Silverlight version has one property, **Radius**: which influences the blurriness of the effect.

The WPF version adds two additional properties. The **KernelType** property is used to specify the blurring algorithm. The default algorithm is the infamous **Gaussian** blur. To switch to the simpler and less smooth **Box** kernel type simply change the value as shown here (Example 4-2).

Example 4-2. Setting BlurEffect Properties

[XML]

```
<CheckBox>
  <CheckBox.Effect>
    <BlurEffect KernelType='Box'
                RenderingBias='Quality' />
  </CheckBox.Effect>
```

```
</CheckBox>
```

There are tradeoffs in shaders, just as in other areas of programming. Blur algorithms can affect rendering speed so the WPF **BlurEffect** provides the **RenderingBias** property as a means to choose performance or quality output for the effect. To get better quality output alter the property as shown in Example 4-2.

DropShadowEffect

The UI design community has a turbulent relationship with the drop shadow. One decade it's a beloved tool in UI design and it pervades the popular design metaphors and the next it isn't. Designers are restless and inquisitive and eventually the drop shadow falls from favor and is viewed as an anachronism by the same community. If you long to add a shadowy aspect to your UI, reach for the **DropShadowEffect** class.

The Silverlight version contains a few properties that are self-explanatory (**Color**, **Opacity** and **ShadowDepth**) so I won't burden you with a description. The **Direction** property represents the angled direction of the shadow. A direction of zero draws a shadow to the right of the host element. Higher values rotate the shadow counterclockwise with the default value (315) placing the shadow in the lower right position. The **BlurRadius** property configures the blurriness of the shadow. Set the **BlurRadius** to zero and the shadow has a crisp, sharp edge, crank up the value for maximum shadow fuzziness.

WPF adds one additional property, **RenderingBias**, over the Silverlight version, which provides the same services as seen in the **BlurEffect.RenderingBias** property described earlier.

Nested Effects

When an effect is applied to a framework element, it affects that element and all of its children. In many circumstances, this is the appropriate approach and the UI looks as expected. Other times the nested effects give an undesirable look to the UI. Figure 4-2 shows two stack panels with a drop shadow applied. The first stack panel has the desired look, because its background brush is fully opaque. The second stack panel uses a solid color background brush with the alpha channel set to a non-opaque value. Because the brush is semi-transparent the drop shadows for the child elements are visible.

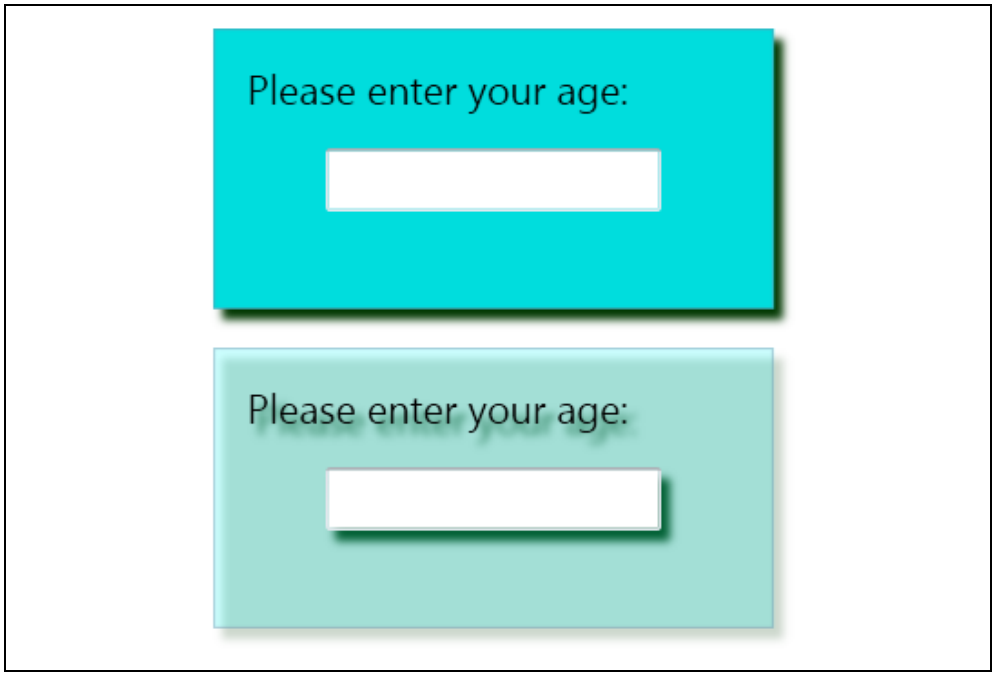

Figure 4-2. Two StackPanels with dropshadow

> Take heed, once an effect is set on a parent element, there is no way to disable the effect on its children elements.

Multiple Effects on One Element

On a sophisticated interface there might be effects applied at different levels of the visual tree. It's likely that at some point you will want to apply multiple effects to a single element. The **Effect** property has some limitations, which you should understand before proceeding. The primary constraint on your creativity is that the **Effect** property can only have a single effect in scope at any time. In other words, there is no collection of effects permitted on a **UIElement**.

Imagine that you want to apply a blur and drop shadow to a button. The work-around for the single effect problem is to nest the button inside another element and apply the second effect to the containing element. Example 4-3 shows some XAML that demonstrates this technique.

Example 4-3. Using a Canvas to add second effect to a Button

[XML]

```
<Canvas>
  <Canvas.Effect>
    <DropShadowEffect />
  </Canvas.Effect>
  <Button Content='Blurred and Shadowed'
          Width='180'
          Height='50'>
    <Button.Effect>
```

```
      <BlurEffect />
    </Button.Effect>
  </Button>
</Canvas>
```

It's a bit underwhelming to learn that Microsoft only includes these two simple effects in the framework. With the vast number of shaders known to the graphics programming crowd I was expecting a lot more out of the box. Fortunately, Expression Blend fills in the gaps and provides many supplementary effects.

Expression Blend Effects

The Expression Blend team is constantly looking for tools to enhance the XAML design experience. A few years ago, they decided to cherry-pick the best shader effects and package them for use in Silverlight/WPF projects (see Figure 4-3). In the Blend interface you can easily add these effects to elements via the Assets panel. You are not limited to using Expression Blend to access them as you can always add a reference to the **Microsoft.Expression.Effects** DLL to bring them into any XAML project.

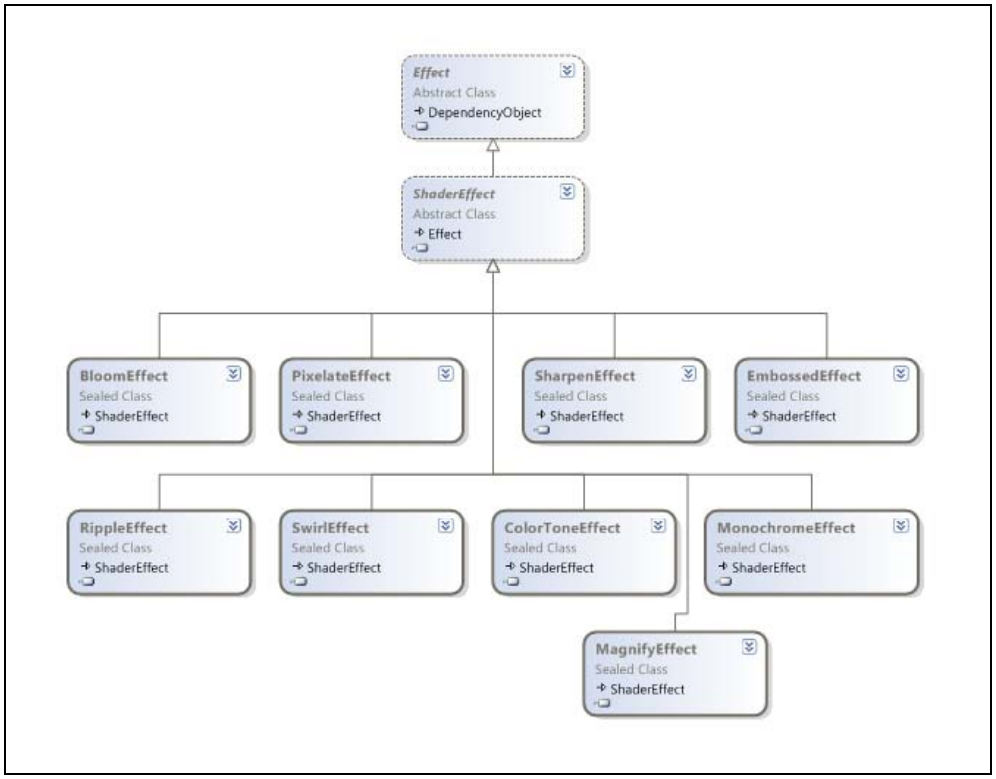

Figure 4-3. Expression Blend effects

Using a Blend Effect

The first step to using a Blend effect is to add a reference to the Blend effect library (*Microsoft.Expression.Effects.dll*). If you have installed Expression Blend in the default location the Silverlight DLL is in the *C:\Program Files\Microsoft*

SDKs\Expression\Blend\Silverlight\v4.0\Libraries directory and the WPF version is in the *C:\Program Files\Microsoft SDKs\Expression\Blend\.NETFramework\v4.0\Libraries* directory.

To use the effect in a XAML file add the Blend namespace as shown in the following XAML (Example 4-4).

Example 4-4. Add Blend effects namespace to XAML file

[XML]

```
<UserControl
  x:Class="Demo.Examples.UseBlendEffectPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:ee="http://schemas.microsoft.com/expression/2010/effects"
...
```

Now it's just a matter of setting the **Effect** property and configuring some parameters as shown here in Figure 4-4.

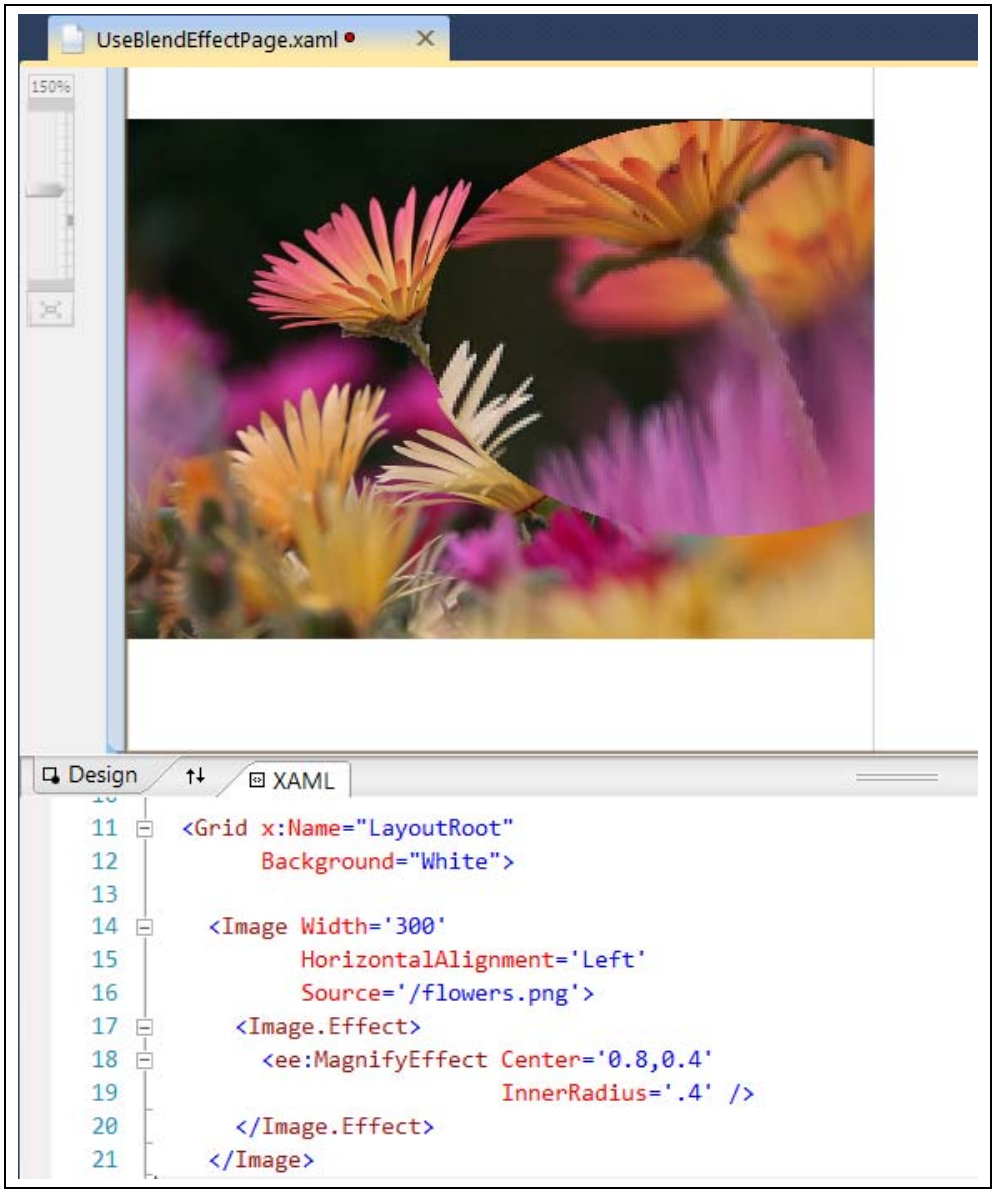

Figure 4-4. Using a Blend effect in the Visual Studio XAML editor

There are about a dozen standard effects in the Blend library. Blend also includes specialized effects known as transition effects. I won't detail either type of effect in this chapter, but you will see more of the standard and transition effects in chapter 5.

You may encounter the BitmapEffect class and its derived types (BevelBitmapEffect, BlurBitmapEffect, DropShadowBitmapEffect, EmbossBitmapEffect, and OuterGlowBitmapEffect) while exploring the WPF libraries. Don't be fooled by the name, these are legacy effects from the early days of WPF; they are not implemented with

> pixel shaders. They are slow and inefficient when compared to their speedy ShaderEffect relatives and are ultimately destined for the .NET dustbin.

Custom Effects

The process of creating a custom effect starts by creating an unmanaged pixel shader. As you may recall, pixel shaders are written in their own quirky programming language called HLSL. Once the HLSL shader code is finished, it is compiled into a binary .ps file. To use the shader, it has to be loaded into the rendering engine input stream. To accomplish this task you need to work with the .NET **ShaderEffect** and **PixelShader** classes.

The **ShaderEffect** is the abstract class that serves as a base for your custom effect class. It is a dependency object, so you can populate it with dependency properties. It works in conjunction with the **PixelShader** class. The **PixelShader** class is a managed wrapper around your HLSL pixel shader. Internally, the **ShaderEffect** keeps a reference to the **PixelShader** class, so that it can inject the unmanaged shader into the graphics pipeline. You will have little interaction with the **PixelShader** class, other than configuring it to load the shader. Most of the customization of your effect revolves around the **ShaderEffect** class.

The **ShaderEffect** offers a handful of member that we'll examine in this chapter.

- RegisterPixelShaderSamplerProperty
- UpdateShaderValue
- PixelShaderSamplerCallback
- PixelShaderConstantCallback
- Padding

Creating a custom ShaderEffect

Consider the following code definition:

```
public class BareBones : ShaderEffect {}
```

While this might technically be considered a **ShaderEffect** it is an empty shell, incapable of influencing any pixels. The first step in turning the class into a useful effect is to load an unmanaged pixel shader file.

> This chapter concentrates on understanding the .NET code and leaves the in-depth discussion of unmanaged pixel shaders for another chapter. To that end, the examples in this section assume that a pixel shader has been compiled into a .ps file and is ready to use in the custom effect.

Loading the .ps file

The compiled pixel shader is stored inside a binary file. It is common to name this file with a .ps extension but that is not a requirement. To make it accessible to your **ShaderEffect** add it to your .NET project and mark it as a project resource. It's still not usable until your **ShaderEffect** extracts the .ps file and associates it with the managed

PixelShader class. The syntax for locating the .ps file is the same as retrieving any other project resource file. Here is some sample code (see Example 4-5) demonstrating how to extract the resource.

Example 4-5. Extracting the .ps file and assigning to PixelShader

[C#]

```
public class LoadingPsFileEffect : ShaderEffect {

  public LoadingPsFileEffect() {

    // the PixelShader class provides a
    // managed wrapper for the unmanaged pixel shader
    var pixelShader = new PixelShader();

    // retrieve the .ps resource with a URI
    // the .ps file needs to be marked as resource in Build Action

    var psFileUri = new Uri
      ("/CustomShaderEffects;component/PsFiles/BlueTintEffect.ps",
        UriKind.Relative);

    pixelShader.UriSource = psFileUri;

    // store the reference to the PixelShader instance
    // in the ShaderEffect.PixelShader property
    this.PixelShader = pixelShader;
  }
}
```

The code starts by creating an instance of the **PixelShader** class in the class constructor. Next, a new URI is created and assigned to the **PixelShader.UriSource**. This example assumes that the assembly containing the resource is named CustomShaderEffects and that the .ps file is in the PsFiles project folder. Finally, the **PixelShader** reference is assigned to the **ShaderEffect PixelShader** property. From this point forward, the **ShaderEffect** will manage the communication with the GPU.

> For simplicity sake, I'll use the term GPU in this chapter to refer to both the WPF and Silverlight rendering engine. The purists in the audience will be offended but it makes it easier to talk about the process in this chapter.

The **LoadingPsFileEffect** class is a functional effect so let's see how to use it in a XAML page.

Use the ShaderEffect

Using your custom effect is similar to working with the Blend effects. Start by compiling your project and then adding a custom **xmlns** namespace to the XAML file. This xmlns attribute indicates which assembly contains the preferred effect. Once you have the xmlns namespace configured you can use it as the following code reveals (Example 4-6).

Example 4-6. Using the effect on an Image element

[XML]

```
...
<!-- In the root element add this namespace-->
  xmlns:effects='clr-namespace:HLSL.Book.Ch04.TheEffects'

<!-- Use the Effect in your application-->
<Image Source='/Images/garden1.jpg'>
  <Image.Effect>
    <effects:LoadingPsFileEffect />
  </Image.Effect>
</Image>
...
```

Once the project is compiled, you can see the effect result by running the application or viewing it in the Visual Studio designer as shown in Figure 4-5.

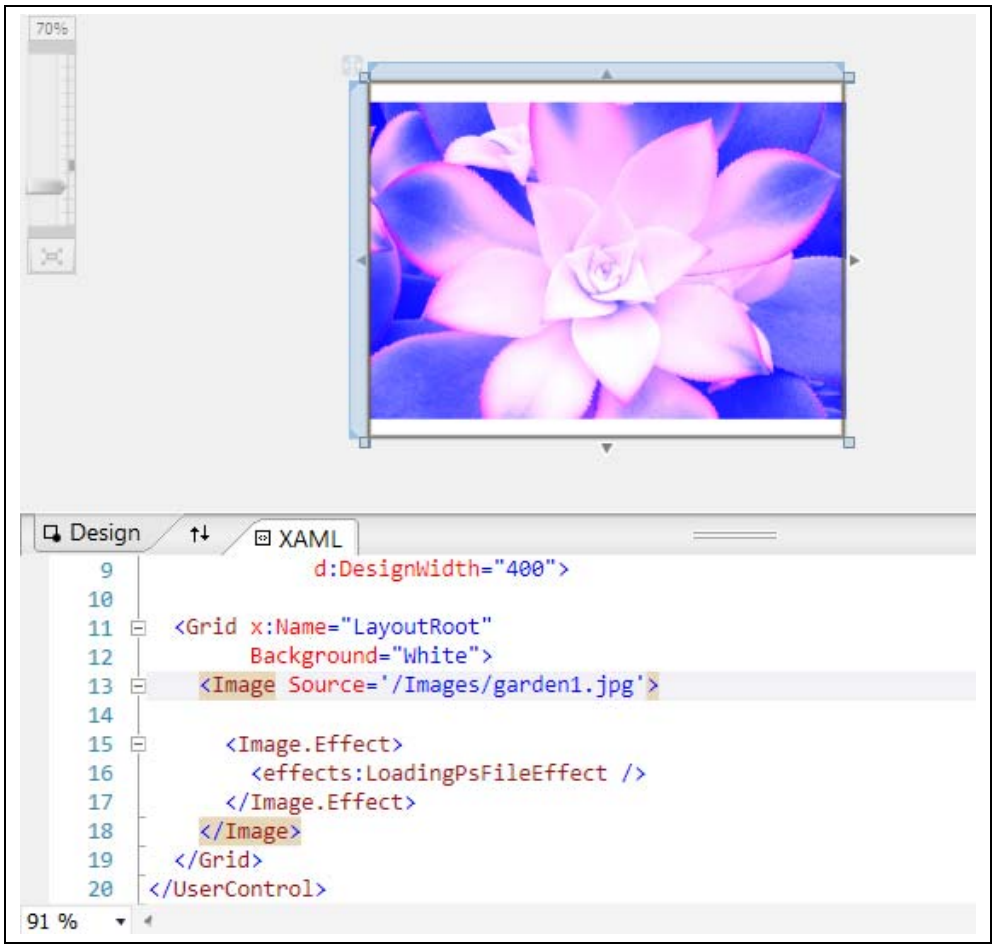

Figure 4-5. Viewing the custom effect in Visual Studio designer

Working with Samplers

In the preceding example, the **ShaderEffect** was applied to the entire image. Clearly, that implies that the pixels from the **Image** element are passed to the HLSL shader. How does that happen?

To understand how this works we need to look at the sampler2D concept in the HLSL specification and the **ShaderEffect.RegisterPixelShaderSamplerProperty** in the managed libraries.

Lets' start by examining the HLSL (Example 4-7) for the BlueTintEffect:

Example 4-7. HLSL code for a blue tint shader

[C#]

```
sampler2D input : register(s0);
float4 main(float2 uv : TEXCOORD) : COLOR  {

  float4 Color;
  Color = tex2D( input , uv.xy);
  Color.b += 1 + uv.y;
  return Color;
}
```

It's a simple color alteration shader. It applies a slight blue tint to each inbound pixel. Direct your attention to the first line of the example. It's in that first line that you see how the HLSL code gets the inbound pixels.

The HLSL specification states that pixel shaders have access to bitmap information via samplers. A **sampler** is a bitmap that is stored in video memory. In the early days of shaders the sampler was often used to store a small texture file (for example an image containing; bricks, stones, moss or cloth) that was mapped or painted onto a 3D object to make the model look realistic. The early graphics pioneers called it a sampler because it was a way to sample a texturemap within the shader. The terminology persists to this day. In a XAML application, the HLSL sampler usually contains the rasterized output of the effected UI elements.

Samplers are passed into the HLSL program by means of the GPU registers. To do this in HLSL you declare a program level variable and associate it with a shader register as shown here:

[C#]

```
sampler2D input : register(s0);
```

In this example, the variable name is **input** and the associated shader register is **s0**. The **sampler2D** variable type signals that the accompanying GPU register contains bitmap data.

> Samplers and other inputs to the shader are declared at the top of the HLSL code and are considered global variables by the HLSL specification. Be aware that the shader term *global variable* has a different connotation here, especially when compared to your favorite .NET language. Global variables are settable during the shader initialization phase, but cannot be changed during the shader execution. This guarantees that the parameter value is constant for all the pixels processed by the shader.

The Pixel Shader 2.0 specification permits up to 16 shader registers. Unfortunately, .NET restricts the number of accessible sampler registers to a smaller number. Silverlight and WPF 3.5 limit you to a maximum of four inputs while WPF 4.0 is more generous and ups the input limit to eight.

Implicit Input from ShaderEffect

We've just seen that the HLSL shader uses the sampler2D type for its texture input. That won't work on the .NET side; we need a Silverlight/WPF specific type instead. The good news is that .NET uses the familiar **Brush** type for this purpose. Several types of XAML brushes can be used as input but we'll start by looking at a special, effect-friendly one called **ImplicitInputBrush**.

Example 4-8 shows one of the most common scenarios for using an effect by setting the **Effect** property on an element.

Example 4-8. Use the ImplicitInput brush

[XML]

```
<TextBox>
  <!-- Use the ImplicitInput brush feature of the Effect base class -->
  <TextBox.Effect>
    <effects:BlueTintEffect />
  </TextBox.Effect>
</TextBox>
```

In this circumstance, the "sampler" that the shader gets as input is the rasterization of the **Textbox**. As mentioned above, a brush is used to send the information to the shader. A close inspection of the XAML in Example 4-8 reveals no trace of a brush however. What's happening?

The **ShaderEffect** base class has some default behavior that creates a special **ImplicitInputBrush** in this situation. This implicit brush contains the rasterized **Textbox** pixels, which are eventually sent over to the shader for processing.

To take advantage of this implicit brush feature requires nothing more than registering the shader **.ps** file as you saw in Example 4-5. To assign any other type of brush to the shader texture requires creating an explicit **DependencyProperty** in your custom effect.

Explicit Input from ShaderEffect

Start by creating a dependency property within the custom **ShaderEffect** and marking the property type as **System.Windows.Media.Brush**. Traditionally this property is named Input, but the choice of name is entirely up to you and your imagination. To integrate this Input property with the HLSL shader you must associate the dependency property with the correct GPU **s** register. For convenience, the **ShaderEffect** class exposes the static **RegisterPixelShaderSamplerProperty** method for this purpose.

Here is the explicit way to achieve the association:

Example 4-9. Writing a DependencyProperty that uses the "s" register

[C#]

```
// the last argument (0) refers to the HLSL s register

public static readonly DependencyProperty InputProperty =
  ShaderEffect.RegisterPixelShaderSamplerProperty("Input",
  typeof(AddingInputTextureEffect), 0);
```

With this dependency property in place, the custom effect is applied to any brush assigned to the Input property

> Even though the effect has an explicit Input property, you can still use the syntax shown in Example 4-8 to apply the implicit brush.

At this point in the story, you know how to create an explicit input property. I'll show how to assign other brushes to it but first let's look at a small scenario that highlights shader input and output within the visual tree.

Pipeline trivia

To explore these concepts I'll use a sample UI with four elements placed inside a Canvas panel. Look at the screenshot of the sample elements in the Visual Studio designer (Figure 4-6).

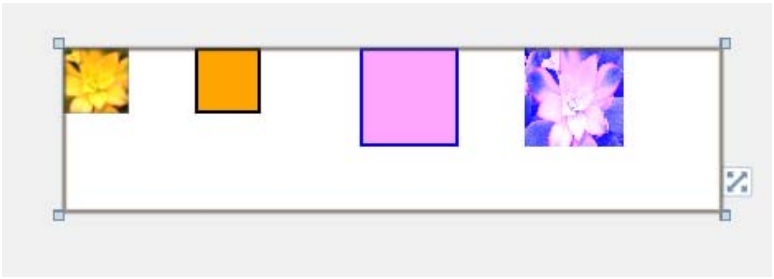

Figure 4-6. Four elements in a canvas

The first two elements on the left side have no effects configured. As you can see from the following XAML snippet (Example 4-10), there is nothing especially notable about these two elements.

Example 4-10. Two elements

[XML]

```
...
<!-- Normal Image.
     Drawn at Location(0,0) Size(40,40) -->

<Image Source='/Images/garden1.jpg'
       x:Name='GardenImage'
       Width='40'
```

```
       Height='40'
       Canvas.Top='0'
       Canvas.Left='0'
       Stretch="UniformToFill" />

<!-- Normal Rectangle.
     Drawn at Location(0,80) Size(40,40) -->
<Rectangle x:Name='RectangleWithoutEffect'
           Fill='Orange'
           Width='40'
           Height='40'
           Stroke='Black'
           StrokeThickness='2'
           Canvas.Top='0'
           Canvas.Left='80' />
...
```

Silverlight/WPF processes these two elements (**GardenImage** and **RectangleWithoutEffect**) during the layout phase. Once that phase is finished, it knows the location and size for both elements and rasterizes their UI for consumption by rendering engine.

It's a similar process for elements with effects. Take, for example, the two rectangles defined in the following XAML snippet (Example 4-11). They are similar to the prior example, but have the distinction of having the BlueTintEffect applied.

Example 4-11. Two rectangles with effects applied

[XML]

```
...
<!-- Rectangle with Effect applied. Output from pixel shader
     is drawn at Location(0,180) Size(60,60)
     Raster input into the pixel shader comes from the Rectangle -->
<Rectangle x:Name='RectangleWithEffect1'
           Fill='Orange'
           Width='60'
           Height='60'
           Stroke='Black'
           StrokeThickness='2'
           Canvas.Top='0'
           Canvas.Left='180'>
  <Rectangle.Effect>
    <effects:BlueTintEffect />
  </Rectangle.Effect>
</Rectangle>

<!-- Rectangle with Effect applied. Output from pixel shader
     is drawn at Location(0,280) Size(60,60)
     Raster input into the pixel shader comes from the ImageBrush -->
<Rectangle x:Name='RectangleWithEffect2'
           Fill='Orange'
           Width='60'
           Height='60'
           Stroke='Black'
           StrokeThickness='2'
           Canvas.Top='0'
           Canvas.Left='280'>
```

```
  <Rectangle.Effect>
    <effects:BlueTintEffect>
      <effects:BlueTintEffect.Input>
        <ImageBrush ImageSource='{Binding
          ElementName= GardenImage,Path=Source}' />
      </effects:BlueTintEffect.Input>
    </effects:BlueTintEffect>
  </Rectangle.Effect>
</Rectangle>
...
```

Once Silverlight/WPF has finished the layout pass, it knows the location and size for **RectangleWithEffect1** and **RectangleWithEffect2**. During the rasterization phase, it passes the rasterized output data into the elements associated shader. The pixel shader does its pixel voodoo and the resultant output is placed in the regions reserved for these two rectangles.

> To hammer home the point -- RectangleWithEffect1 is drawn at the same location and size regardless of whether it has an effect or not.

Explicit Input Revisited

So where do the inbound pixels for the pixel shader come from? That depends on a few factors. **BlueTintEffect** has an Input **DependencyProperty** defined as seen previously in Example 4-9.

Lets' apply the effect and dissect where the input comes from. Example 4-12 shows the **BlueTintEffect** applied to a **Rectangle**.

Example 4-12. Using the BlueTintEffect on an Rectangle element

[XML]

```
<Rectangle.Effect>
  <effects:BlueTintEffect />
</Rectangle.Effect>
```

Even though the effect has an explicit input property, it is not used when using this syntax; instead, it uses the implicit input. You can verify this is true by checking the Input property as seen in the code in Example 4-13.

Example 4-13. Checking explicit Input brush

[C#]

```
var brush =
(RectangleWithEffect1.Effect as CustomShaderEffects.InputTestEffect).Input;

// brush is null, indicating that the Input property was not set
```

Because the **BlueTintEffect** exposes an explicit Input property it's possible to pass in other brushes to the shader input as shown in this XAML (Example 4-14).

Example 4-14. Assigning an ImageBrush to the explicit Input property

[XML]

```
<Rectangle.Effect>
  <effects:BlueTintEffect>
    <effects:BlueTintEffect.Input>
      <ImageBrush ImageSource='{Binding
        ElementName=GardenImage,Path=Source}' />
    </effects:BlueTintEffect.Input>
  </effects:BlueTintEffect>
</Rectangle.Effect>
```

As you can see, the pixel shader input is coming from an **ImageBrush** but you can also use a **VisualBrush**, or **BitmapCacheBrush** in the same manner.

> When an effect is applied to an element, the output of the shader is exactly the same size as the original input size. If the rectangle is 60 x 80 pixels, the output of the shader is also sized at 60 x 80 pixels. Choosing implicit or explicit input has no bearing on the output size.
>
> The only exception to the sizing rule is when an effect uses the effect padding properties.

Multi Input Shaders

A pixel shader can have up to 16 input samplers defined in the HLSL. WPF 4.0 limits you to 8 however.

Here is a HLSL example with two input samplers defined (Example 4-15).

Example 4-15. Pixel shader with two sampler2D inputs

[C#]

```
sampler2D BaseImage: register(s0);
sampler2D TextureMap : register(s1);

float4 main(float2 uv : TEXCOORD) : COLOR
{

  float hOffset = frac(uv.x / 1 + 1);
  float vOffset = frac(uv.y / 1 + 1);
  float2 offset = tex2D(TextureMap, float2(hOffset, vOffset)).xy * 4 - 1/2;

  float4 outputColor = tex2D(BaseImage, frac(uv + offset ));
  return outputColor;
}
```

The first **sample2D** variable is using the s0 register while the second **sample2D** variable maps to the s1 register.

> Be pragmatic and thoughtful when naming your HLSL variables. Readability is just as important in HLSL code as in other programming languages.

In this example, the first sample2D variable name reflects its status as the base image. The second variable name, **TextureMap**, indicates that it holds a bitmap containing lookup textures. The HLSL in the sample uses a simple mapping technique to blend the pixels from the two sampler inputs.

On the .NET side, you need to create two dependency properties and call **ShaderEffect.RegisterPixelShaderSamplerProperty** on both. The registration code will be similar to the code shown in Example 4-9.

To use these inputs in XAML use syntax like this:

Example 4-16. Assigning some ImageBrushes to the input properties

[XML]

```
...
<Rectangle x:Name='RectangleWithEffect1'
           Width='256'
           Height='170'
           Stroke='Black'
           StrokeThickness='2'>
  <Rectangle.Effect>
    <effects:TwoInputEffect>

      <effects:TwoInputEffect.BaseImage>
        <ImageBrush ImageSource='{Binding
                    ElementName=GardenImage2,Path=Source}' />
      </effects:TwoInputEffect.BaseImage>

      <effects:TwoInputEffect.TextureMap>
        <ImageBrush ImageSource='{Binding
                    ElementName=GardenImage1,Path=Source}' />
      </effects:TwoInputEffect.TextureMap>
    </effects:TwoInputEffect>

  </Rectangle.Effect>
</Rectangle>
...
```

This is a beautiful effect as you can see in the screenshot below (Figure 4-7). It shows four images, the left two being the original images and the right two showing the texture mapping.

Figure 4-7. Two original images and two blended images

Understanding Sampler Size

All sampler inputs into the shader are resized by the Silverlight/WPF runtime to match the render size of the host element.

Consider the following XAML:

Example 4-17. Effect brushes with mismatched size

[XML]

```
...
<Rectangle x:Name='Rectangle1'
           Width='400'
           Height='400'>
  <Rectangle.Effect>
    <effects:TwoInputEffect>

      <effects:TwoInputEffect.BaseImage>

        <!-- flowers_wide.jpg is  925 x 260 pixels -->
        <ImageBrush ImageSource='/Images/flowers_wide.jpg' />
      </effects:TwoInputEffect.BaseImage>
      <effects:TwoInputEffect.TextureMap>

        <!-- garden_small.jpg is  150 x 200 pixels -->
        <ImageBrush ImageSource='/Images/garden_small.jpg' />
      </effects:TwoInputEffect.TextureMap>
    </effects:TwoInputEffect>
  </Rectangle.Effect>
```

```
</Rectangle>
...
```

This example uses the **TwoInputEffect** and assigns an **ImageBrush** to each sampler input. During the layout pass, the runtime determines the render size and location for the host rectangle, in this case, a 400 x 400 square. When each **ImageBrush** is readied for the shader, its sized is constrained to the same 400 x 400 size as the host rectangle; causing the larger image to be compressed and the smaller image to be enlarged. As far as the HLSL shader is concerned, it gets two 400 x 400 textures assigned to its **s** registers. If you could debug the shader pipeline and look at the two textures stored in video memory, you'd see that this is true.

Use a transform to manipulate an input brush before the scaling occurs as shown in Example 4-18:

Example 4-18. Transforming a brush before sending to shader

[XML]

```
...
<effects:TwoInputEffect.TextureMap>
  <ImageBrush ImageSource='/Images/flowers_wide.jpg'>
    <ImageBrush.Transform>
      <CompositeTransform ScaleX = '.4'
                          ScaleY = '.4'
                 TranslateX = '100' />
    </ImageBrush.Transform>
  </ImageBrush>
</effects:TwoInputEffect.TextureMap>
...
```

Now that you've seen how to pass bitmap parameters to the shader it's time to expand your horizons and see how to pass other types of parameters into the shader.

Creating Parameterized Effects

Parameters are the lifeblood of a flexible programming model. Can you imagine how dull and impractical it would be to work in a programming language without parameters? Luckily for us, HLSL accepts various types of input data into the shader.

You've already seen how to pass bitmap data to the pixel shader through the GPU registers. To be more precise, we used the sampler registers for this purpose. They are designated with the "s" nomenclature (**s0**, **s1**, **s2**, etc.). You are not limited to passing bitmap data into the shader as HLSL sports another set of registers known as the constant registers (**c0**, **c1**, **c2** etc.). A constant parameter is similar to a **readonly** field in C#. The value is changeable during the pixel shader initialization period, but remains constant throughout the execution of the shader. In other words, once the value is set it will be the same for every pixel processed by the pixel shader. You can have up to 32 constant registers in PS_2_0. PS_3_0 expands that to 224, but is only accessible in WPF 4.0.

Let's rewrite the multi input shader as follows:

Example 4-19. Adding constant registers to the HLSL shader

[C#]

```
sampler2D BaseImage: register(s0);
sampler2D TextureMap : register(s1);
float vertScale : register(c0);
float horzScale : register(c1);
float translateX : register(c30);
float translateY : register(c31);

float4 main(float2 uv : TEXCOORD): COLOR
{

  float hOffset = frac(uv.x / vertScale + translateX);
  float vOffset = frac(uv.y / horzScale + translateY);
  float2 offset = tex2D(TextureMap, float2(hOffset, vOffset)).xy * 4 - (1/2);

    float4 outputColor = tex2D(BaseImage, frac(uv + offset ));
    return outputColor;
}
```

In addition to the sampler2D inputs shown earlier in Example 4-15, the refactored code contains four additional input values declared at the top of the pixel shader. If you look closely, you can see that these new items are float values, which are loaded into registers **c0**, **c1**, **c30** and **c31** and then used inside the **main** function.

The **ShaderEffect** class transmits parameter information to a HLSL constant register through a **DependencyProperty**. It does this by using the special **PixelShaderConstantCallback** method. The trip is one-way, from the effect class to the pixel shader. The parameter value never travels back to the effect class.

Now, let's focus on how to write the effect to take advantage of these parameters. Here is a snippet (Example 4-20) that shows the **DependencyProperty** registration:

Example 4-20. Binding the "c" registers with PixelShaderConstantCallback

[C#]

```
...
public static readonly DependencyProperty VertScaleProperty =
  DependencyProperty.Register("VerticalScale", typeof(double),
    typeof(InputParametersEffect),
      new PropertyMetadata(((double)(0D)),
        PixelShaderConstantCallback(0)));

public static readonly DependencyProperty HorzScaleProperty =
  DependencyProperty.Register("HorizontalScale", typeof(double),
    typeof(InputParametersEffect),
      new PropertyMetadata(((double)(0D)),
         PixelShaderConstantCallback(1)));

// ... continue in this manner for other dependency properties
```

The last argument on each registration line is the important one for this discussion. We call the **PixelShaderConstantCallback** method and pass in the appropriate constant register. **PixelShaderConstantCallback** sets up a **PropertyChangedCallback** delegate, which is invoked whenever the **DependencyProperty** is changed. Example 4-21 shows how easy it is to use these new properties.

Example 4-21. Setting some shader parameters via DependencyProperties

[XML]

```
...
<Rectangle x:Name='RectangleWithEffect2'
           Width='Auto'
           Height='Auto'
           Margin='3'
           Grid.Row='1'>
<Rectangle.Effect>
  <effects:InputParametersEffect
        HorizontalScale='{Binding ElementName= horzSlider, Path=Value}'
        VerticalScale='{Binding ElementName=vertSlider, Path=Value}'
        TranslateX='{Binding ElementName=xSlider,Path=Value}'
        TranslateY='{Binding ElementName=ySlider,Path=Value}'>

    <effects:InputParametersEffect.BaseImage>
      <ImageBrush ImageSource='/Images/Garden1.jpg' />
    </effects:InputParametersEffect.BaseImage>

    <effects:InputParametersEffect.TextureMap>
      <ImageBrush ImageSource='/Images/Garden2.jpg' />
    </effects:InputParametersEffect.TextureMap>
  </effects:InputParametersEffect>
</Rectangle.Effect>
</Rectangle>
...
```

UpdateShaderValue

There is one more step necessary to make a functional **ShaderEffect**. You need to invoke the **UpdateShaderValue** method in the class constructor for every bound **DependencyProperty** otherwise the pixel shader won't be initialized with the default values for the property. Call the method for every effect property, as shown in Example 4-22, to ensure that the initial value for each property is set in the pixel shader.

Example 4-22. Using the UpdateShaderValue method in the effect constructor

[C#]

```
this.UpdateShaderValue(InputProperty);
this.UpdateShaderValue(TextureMapProperty);
this.UpdateShaderValue(VerticalScaleProperty);
this.UpdateShaderValue(HorizontalScaleProperty);
```

Property Types

On the HLSL side, the constant register works with various types of float values. When you register a **ShaderEffect** **DependencyProperty** with the **PixelShaderConstantCallback** method you are limited to a short list of .NET types. Table 4-1 lists the permitted .NET types, and the matching HLSL types.

Table 4-1. Comparing WPF, Silverlight and HLSL property types

WPF	Silverlight	HLSL
Single	Single	float
Double	Double	float

WPF	Silverlight	HLSL
Point	Point	float2
Size	Size	float2
Color	Color	float4
Vector	NA	float2
Point3D	NA	float3
Vector3D	NA	float3
Point4D	NA	float4

Padding

Normally an effect is applied to an elements actual render size. Therefore an effect for a 200 x 200 Image will modify pixels in a 200 x 200 region. Certain affects, like the drop shadow, need additional space outside the normal render area. Use the **ShaderEffect** padding properties (**PaddingTop**, **PaddingLeft**, **PaddingRight**, **PaddingBottom**) to increase the size passed into the pixel shader.

The padding properties are marked as protected scope so you cannot access them outside your **ShaderEffect**. The typical pattern is to set the padding within your type and expose other dependency properties for client code to access. The built-in **DropShadowEffect** uses the **ShadowDepthProperty** in this manner.

Effect Mapping

Distortion effects are a popular use of pixel shaders (see Figure 4-8).

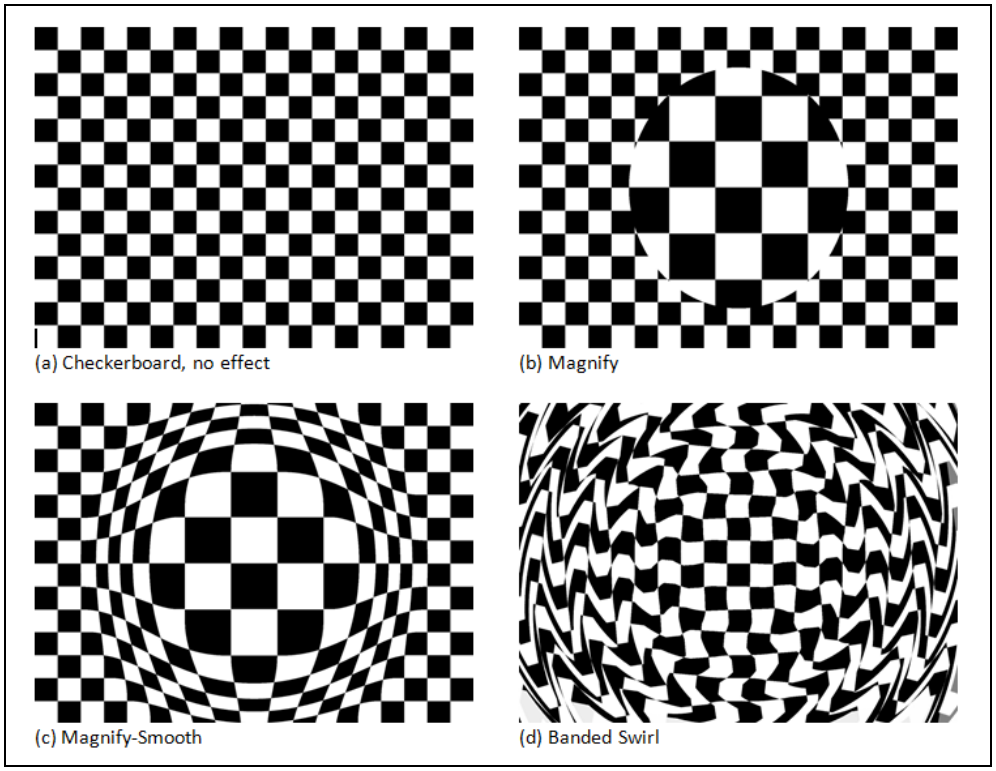

Figure 4-8. Three distortion effects applied to checkerboard.

Distortion effects require extra work if you want them to behave in a predictable fashion. When you apply a distortion effect to an interactive element like a list box (Figure 4-9(a)), the touch, stylus and mouse events won't work as expected. The pixel shader is rearranging the output pixels but the Silverlight/WPF hit-testing infrastructure is unaware that the pixels are in a new location (Figure 4-9(b)).

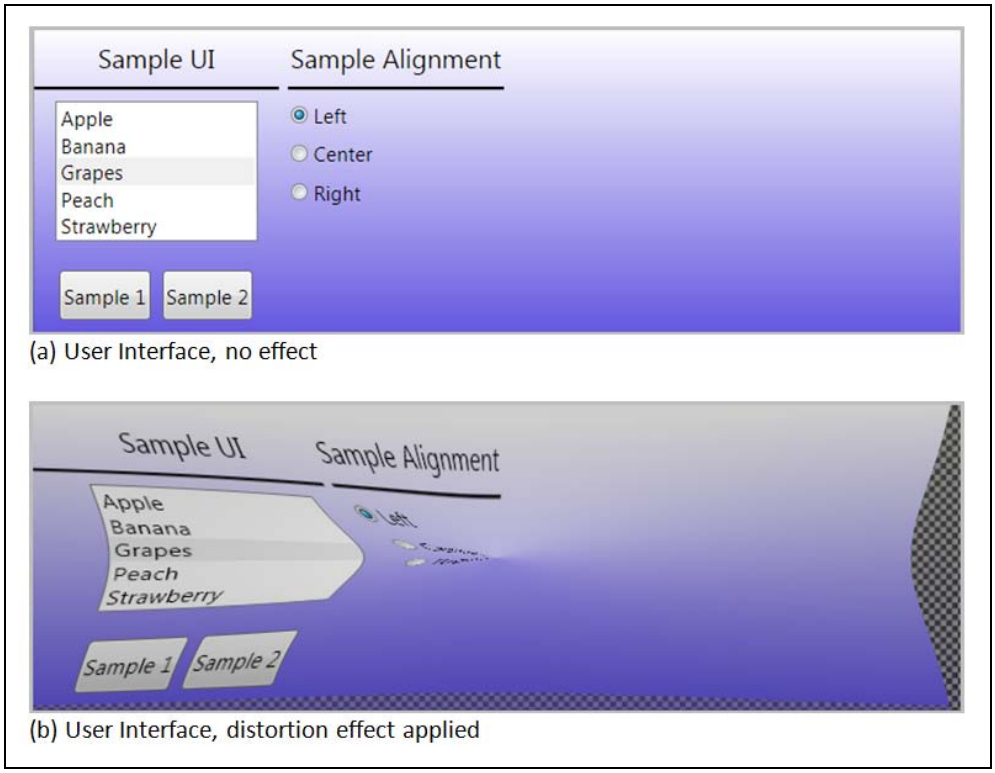

Figure 4-9. UI with Distortion Effect

The **EffectMapping** property provides a way to synchronize the input coordinates between the two worlds. It takes the raw input coordinates and maps them to the pixel shader coordinates. This is accomplished by creating a custom **GeneralTransform** class.

Before we examine the customized **GeneralTransform** let's look at the sample compression shader (Example 4-23) that lives on the HLSL side.

Example 4-23. A compression shader

[C#]

```
sampler2D input : register(s0);
float CrushFactor : register(c0);

float4 main(float2 uv : TEXCOORD) : COLOR
{

  if (uv.y >= CrushFactor )
  {
    float crushAmount = lerp(0, 1, (uv.y - CrushFactor)/(1 - CrushFactor));
    float2 pos = float2(uv.x, crushAmount );
    return tex2D(input, pos);
  }
  else return float4(0,0,0,0);
}
```

This HLSL example takes the incoming pixels and compresses the pixel shader output toward the bottom of the element. The higher the **CrushFactor** property value, the shorter the output image will be.

In the XAML snippet shown below (Example 4-24) the **CrushEffect** causes the **Image** to be rendered at 30% of its original height.

Example 4-24. Applying the CrushEffect

[XML]

```
...
<Border BorderBrush='Red'
        BorderThickness='4'
        Width='240'
        Height='120'
        Margin='5'
        Grid.Row='2'>
  <Image  Stretch='Fill'
          Source='/Images/garden1.jpg'
          MouseMove='distortedImage2_MouseMove'
          MouseLeftButtonUp='distortedImage2_MouseLeftButtonUp'
          Name='distortedImage2'>
    <Image.Effect>
      <effects:CrushWithMappingEffect CrushFactor='.7'/>
    </Image.Effect>
 </Image>
</Border>
...
```

Figure 4-10 shows the output of the **CrushEffect**, when applied to an **Image** element. The image is wrapped in a **Border** element, which shows the size of the **Image** if it didn't have the effect applied.

Figure 4-10. CrushEffect applied to Image element

If there is no **EffectMapping** provided, the image mouse events will fire when the mouse is within the white area, even though it's evident in the screenshot that the image pixels are no longer visible in that region. To fix this shortcoming, create an **EffectMapping** property. The **ImageMapping** property is responsible for returning a custom **GeneralTransform** class to the Silverlight/WPF engine as seen in this code scrap.

Example 4-25. Creating an EffectMapping property

[C#]

```
private CrushTransform _transform = new CrushTransform();
protected override GeneralTransform EffectMapping {
  get {

    _transform.CrushFactor = CrushFactor;

    return _transform;
  }
}
```

GeneralTransform Class

The **GeneralTransform** class is one of the XAML transform classes. Though not as familiar as other transform like **CompositeTransform** it is used by the framework during certain transform actions like **TransformToVisual** and **EffectMapping**. It contains a few members of interest. It has two transform methods, **Transform** and **TryTransform**. Both methods take an incoming point and return a transformed point. The difference between the two is that the **TryTransform** method returns a **Boolean**, instead of throwing an exception if the transform fails for any reason, and it uses an out parameter to deliver the transformed point back to the caller. Example 4-26 shows a few of the member of the **GeneralTransform** class.

Example 4-26. Prototyping the GeneralTransform class

[C#]

```
public class GeneralTransform
{
  // a few of the class members
  public Point Transform(Point point) {
    Point point1;
    if (this.TryTransform(point, out point1)) {
      return point1;
    }
    else {
      throw new InvalidOperationException("Could not transform");
    }}

public abstract bool TryTransform(Point inPoint, out Point outPoint);
}
// sub-classing the GeneralTransform class
public class SampleTransform : GeneralTransform {}
```

Were you to create an instance of the **SampleTransform** class shown in Example 4-26, you could easily get a transformed point with code similar to the following (Example 4-27).

Example 4-27. Getting a transformed point

[C#]

```
var transform = new SampleTransform();
var originalPoint = new Point(10, 20);
Point transformedPoint;
```

```
if (transform.TryTransform(originalPoint, out transformedPoint)) {
  // do something with the out parameter
  Console.WriteLine(transformedPoint.Y);
}
```

The **GeneralTransform** class also has an **Inverse** property. This property is utilized whenever an inverted version of the transform is needed and it is this property that is called during the effect mapping operations. It returns a reference to another transform as shown in Example 4-28.

Example 4-28. Getting the inverse transform from the general transform class

[C#]

```
var t1 = new CrushTransform();
var t2 = crushTransform.Inverse as InverseCrushTransform;
```

GeneralTransform and EffectMapping property

The **ShaderEffect EffectMapping** property tells the Silverlight/WPF framework which **GeneralTransform** class to use during hit-testing and other input events. The framework follows this workflow. When a mouse event is detected (mousemove) the framework get the transform from the **EffectMapping** property. Next, it calls the **Inverse** method to get the undo transform. Finally, it calls the **TryTransform** method on the inverted transform to get the corrected mouse location.

For every distortion action in the pixel shader you provide a undo action in the **Inverse** transformation class. For intricate shaders, the transformation code can get quite complex. The different algorithms available in the HLSL and .NET frameworks exacerbate the problem. Nevertheless, it is your responsibility to write the transform to make hit testing work correctly.

Here is some code (Example 4-29) that demonstrates the transforms that reverse the **CrushEffect**.

Example 4-29. General and Inverse transforms

[C#]

```
public class CrushTransform : GeneralTransform
{
  // create a DependencyProperty that matches the  DependencyProperty
  // in the CrushEffect ShaderEffect class.
  // Is used to pass information from the ShaderEffect to the Transform
  public static readonly DependencyProperty CrushFactorProperty =
    DependencyProperty.Register("CrushFactor", typeof(double),
    typeof(CrushTransform),
    new PropertyMetadata(new double()));

  public double CrushFactor {
    get { return (double)GetValue(CrushFactorProperty); }
    set { SetValue(CrushFactorProperty, value); }
  }
  protected bool IsTransformAvailable(Point inPoint) {
```

```
      if (inPoint.Y < CrushFactor) {
        return false; // No transform available for this point location
      }
      else {
        return true;
      }
    }
    public override bool TryTransform(Point inPoint, out Point outPoint) {
      outPoint = new Point();

      // normal transform actions
      double ratio = inPoint.X;
      outPoint.Y = CrushFactor + (1 - CrushFactor) * ratio;
      outPoint.X = inPoint.X;

      return IsTransformAvailable(inPoint);
    }

    public override GeneralTransform Inverse {
      get {
        // this method is called by framework
        // when it needs a inverse version of the transform
        return new InverseCrushTransform { CrushFactor = CrushFactor };
      }
    }

    public override Rect TransformBounds(Rect rect) {
      throw new NotImplementedException();
    }
  }
  public class InverseCrushTransform : CrushTransform
  {
    public override bool TryTransform(Point inPoint, out Point outPoint) {
      outPoint = new Point();

      // inverse transform actions
      double ratio = (inPoint.Y - CrushFactor) / (1 - CrushFactor);
      outPoint.Y = inPoint.Y * ratio;
      outPoint.X = inPoint.X;
      return base.IsTransformAvailable(inPoint);
    }
  }
```

Summary

Silverlight/WPF has a nice system for integrating shaders and .NET effects. This chapter showed you how to make the managed wrapper for the HLSL shader.

Let's review the steps needed to create your own shaders.

- Write a shader in HLSL
- Compile the shader to a binary file (**.ps**) with FXC.exe or other HLSL compiler
- Add the **.ps** file to your Silverlight/WPF project and set the build action to **Resource**
- Create a .NET effect class that derives from **ShaderEffect**
- Load the **.ps** file into the effect class and assign to its **PixelShader** property

- Setup one or more input dependency properties of type **Brush** and use the **ShaderEffect.RegisterPixelShaderSamplerProperty** method to map the input to the correct GPU **s** register
- If the shader has parameters map each parameter to a dependency property and bind to the correct GPU **c** register with the **PixelShaderConstantCallback** method
- In the effect constructor call **UpdateShaderValue** for each **DependencyProperty** in the class
- For certain shader types create Padding or EffectMapping code
- Apply the effect to any **UIElement**

> The WPF and Silverlight teams took different routes when creating the ShaderEffect and PixelShader classes. Looking at the public interfaces of the implementation, the classes look nearly identical, but a quick look at the internal implementation shows some differences. If you plan on creating shaders that work in both systems be cognizant of the potential internal differences and test accordingly.

As you've seen in this chapter, there are many steps necessary to create a working shader effect class. To ease the development of custom shaders I created specialized utility called Shazzam Shader Editor. It automates most of the steps needed to make effects. A detailed tour of Shazzam is imminent but first comes a chapter showing how to use Expression Blend to add effects to any Silverlight/WPF project.