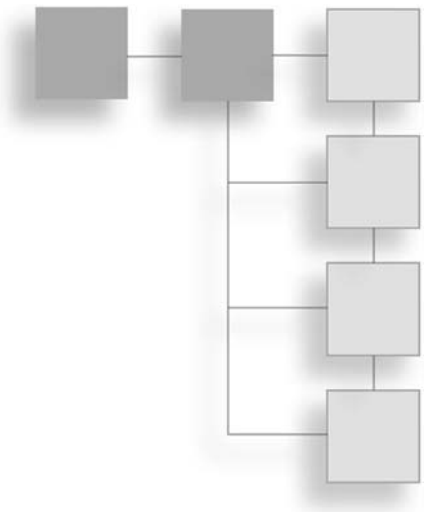
CHAPTER 8



# Making Your Day Brighter

There have been many debates in the past about whether eight bits of resolution per color component are enough to represent all the colors the human eye can see. Generally speaking, eight bits are plenty, but this does not take into consideration something that is of crucial importance in photography.

Although 256 colors per component is sufficient to represent color, what about light intensity? The difference in intensity between candlelight and sunlight is on the order of thousands to one, which obviously can't be represented with only eight bits worth of resolution. You may think this is of little importance, but here are a few real-life examples of how this makes a difference.

Let's say you are outside in the sunlight looking into a torch-lit cave. From this position, you most likely cannot see anything inside the cave, and it appears dark. If you flip the situation around and you are inside the cave looking out, you can see the inside of the cave fine, but the outside appears overly bright, and you cannot see much of what is out there. The reason behind this phenomenon is that both cameras and the human eye can only see a certain range of brightness, and both have exposure control mechanisms, which control the amount of light energy that can be seen based on the average light of their surroundings.

Another example is putting an object between the viewer and a bright source of light. In this case, it seems as if the object is overcome by the light, even though it should normally block the light out. This is because the bright light tends to oversaturate the light receiver, causing some of the light energy to spread onto anything else in your field of vision.

As you can see from these examples, taking light intensity into account can be a strong factor in creating more realistic and immersive renderings. Because of this, many studies have been done on the phenomenon called *high dynamic range*, or HDR. In this chapter, you will

learn about this effect and its implications. You will also learn a few techniques that can be used to re-create this phenomenon on today's and tomorrow's rendering hardware.

## What Is High Dynamic Range?

High dynamic range is the science of recognizing the different intensity levels of light. Under normal rendering circumstances, where the average level of lighting is similar throughout the scene, eight bits of color precision are enough to represent the different colors and light intensities. However, in real life, there are many situations where the levels of lighting vary significantly throughout the scene, leading to a scenario where there isn't enough precision to represent all the phenomena that occur from differences in intensities.

The general process used in HDR rendering is simple and involves the use of render targets like the ones used in previous chapters. This is illustrated in Figure 8.1.

### Glare

Probably the most common effect that comes out of bright intensity light is the glare effect. A bright source of light exhibits a blooming effect, where it can even take over neighboring regions that stand between the light and the receiver.

This effect is caused by the way the human eye and photography equipment work; excess lighting energy affects not only a particular point on the receiver but also neighboring points. In fact, the excess light energy leaks onto its surroundings, creating a glow-like effect often referred to as glare, or *blooming*.

In the past, this effect was reproduced using billboard polygons on top of the source of light with the clever use of alpha blending. This approach is simple and effective but

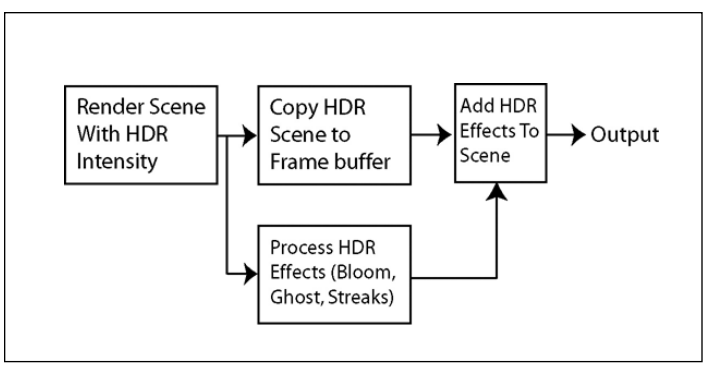



**Figure 8.1** Diagram of the general process used to render high dynamic range effects.

suffers from one major shortcoming. It cannot account for the varying intensities that occur when your source of light is partially occluded. Later in this chapter, I will show you how this effect can be accomplished with blur filters and render targets with proper high-dynamic range considerations.

There is another side effect worth mentioning that comes out of this oversaturation of the receiver. Because a bright source of light overexcites the receiver, once the bright light goes away, it takes some time for the receiver to go back to its normal state. This is the phenomenon that occurs when you see a white blob for a while after seeing a bright flash of light. This effect will not be implemented in this chapter, but it is analogous to setting up an intensity-based motion blur filter, where the current image affects subsequent images while the excitation dissipates.

### Streaks

You may have noticed when driving in a car that sometimes street lights and headlights seem to have a star-like glow around them. This phenomenon is generally due to internal reflections and refractions caused by microscopic scratches on the glass surface of the camera lens. The same thing happens through your car windshield because it is not a well polished surface.

Later in this chapter, I will explain how you can re-create this effect using specifically designed blur filters. These filters blur the bright portions of the render target in a diagonal manner instead of the standard way it has been done with the regular blur filters.

### Lens Flares

Lens flares, or *ghosts*, are commonly seen effects when you see film or photography of a bright source of light such as the sun. This is probably one of the first HDR effects ever used in video games. This phenomenon comes from the fact that the bright light source is reflected between the different lenses of a camera, creating ghost images of the bright lights.

Video games have reproduced this effect by creating fake billboard geometry, placed suitably depending on where the source of light is. In this chapter, you will learn how to generically reproduce this with proper filters.

## A Few HDR Basics

Before you get started on discovering how to re-create all the high dynamic range effects discussed earlier, we have to look at a few basic topics that will come in handy. These topics range from the use of floating-point textures to exposure control.

## What About Floating-Point Textures?

The main reason behind the introduction of the new floating-point textures is to enable support for features that require a wider range of values and precisions than that offered by regular eight-bit textures. High dynamic range rendering is a prime example of where such high-precision textures can be used.

**note**

Because floating-point textures are not yet widely supported, it is likely you will have to use the Hardware Emulation Layer (HEL) mode in RenderMonkey to develop the following high dynamic range shader. Later in this chapter, you will learn how to use regular render targets to estimate some of the HDR effects.

Some hardware implementations do not support bilinear filtering when using floating-point textures. If you have such hardware, you might need to switch to software emulation to get proper results from the HDR shaders developed in this chapter.

Floating-point textures come in two flavors. In fact, you can elect to use either a 16- or 32-bit precision texture. Although 32 bits offer great precision, they also require twice the memory and bandwidth. What this means for you is that the color values in a floating-point texture are represented in a way that enables you to represent any numerical value. Because HDR does not require such a great range, 16-bit textures will be used for the render targets throughout this chapter, especially the `A16R16G16B16` format.

Because floating-point textures are ideal for high dynamic range rendering, it makes sense to show you the basic techniques using these high-precision textures. If your hardware does not support them, however, you can use software emulation. Later in this chapter, you'll learn how to use non-floating-point textures to approximate the same effects.

## Exposure Control: The First Step

The first thing to consider when dealing with high dynamic range is exposure control. Both the human eye and cameras need to be able to adjust to different lighting conditions. You do not wish to have a bright outdoor scene being washed out and an indoor scene being totally dark. This is why, for example, the iris of the human eye adjusts to different lighting environments.

In rendering high dynamic range scenery, you need a similar mechanism to control the average lighting intensity on your scene. This mechanism is commonly called exposure control. The framework shader introduced here includes a basic scene with a background and teapot object and incorporates the needed components to handle exposure control.

Because we want to develop a simple shader with an environment and an object, we'll start by using the `shader_1.rfx` template developed in Chapter 7. The first step to make this

shader HDR-ready is to add some brightness to the scene. For this example, we will be adding HDR information to the environment map.

The ideal solution for this would be to adjust or create the environment as a floating-texture that contains full brightness information for the scene. Although this is doable, few software solutions are available to do such a task at the moment. An easier solution is to take the current environment map and use the alpha channel as a brightness multiplier. A 0 alpha indicates the lowest brightness level, and 1 is the maximum intensity. You would then use this value from the alpha channel in your rendering to multiply with the color intensities, essentially creating a higher range of colors than normally available.

Using image editing software, such as Adobe Photoshop, you can add alpha values to your environment map so that brighter components, such as the sun, have a greater value in their alpha channel. Figure 8.2 shows the snow environment side-by-side with its HDR alpha channel.

**note**

For more information on how to build high dynamic range environment maps, you can refer to Paul Debevec's high dynamic range imaging website at: http://www.debevec.org.

With a high dynamic range environment map, the first step needed to create an HDR shader is to convert the render targets to use floating-point textures. To do so, simply double-click on every target and change the texture format to `A16R16G16B16F`:

The next step is to use the alpha channel from the environment map to calculate the real HDR values. This is done by multiplying the environment color by the alpha channel when each object is rendered, considering the alpha value as being within the zero to 64 range. This leads to the following pixel shader code snippet:

```
color = color * (1.0+(color.a*64.0));
```

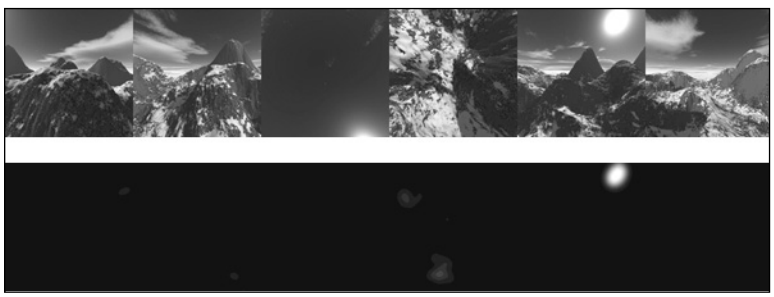


**Figure 8.2** Color and alpha channel values for the HDR-ready environment map.

This little piece of shader code takes the alpha value from a zero to one range and converts this value to an integer multiplier in the range of one to 256. This value is then used as a multiplier for the incoming color values from the environment map.

The only aspect missing in this shader is exposure control. For this, you first need a variable to define the exposure value. Create a new variable called `Exposure_Level`, which is of type `SCALAR`. This value controls the exposure by serving as a scaling factor for the render target colors when it is copied to the screen buffer.

For this, you need to modify the pixel shader we developed earlier to scale down the values based on the exposure value. This change yields the following code:

```
sampler Texture0;
float4 ps_main(float3 dir: TEXCOORD0) : COLOR
{
   // Read texture and determine HDR color based on alpha
   // channel and exposure level
   float4 color = texCUBE(Environment, dir);
   return color * ((1.0+(color.a*64.0))* Exposure_Level);
}
```

With this shader, you can now see your HDR environment being rendered. Try different values of exposure control and see how the scene changes. Figure 8.3 shows different rendered outputs for different exposure values.

The complete version of this shader is included on the CD-ROM as `shader_1.rfx`.

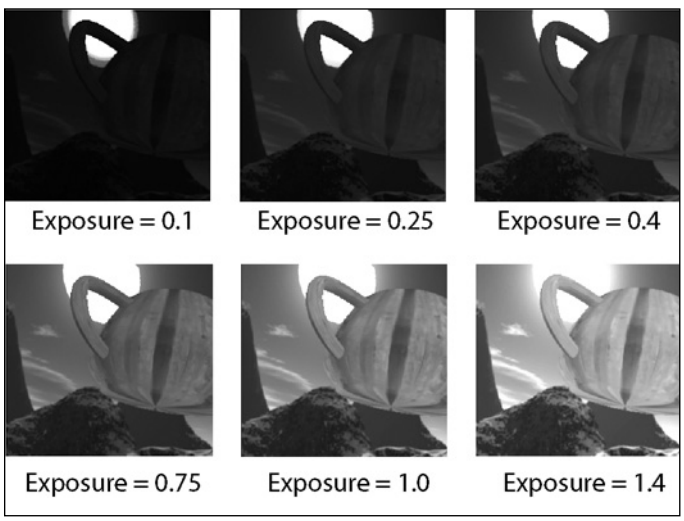



**Figure 8.3** Different exposure control values and their effect on a rendered scene.

### A Note on Automatic Exposure Control

The human eye adjusts its iris automatically to adapt to the surrounding light. This is a behavior you might want to emulate in your rendering.

Automatic exposure control aims to adjust the average brightness of the scene. The idea is that the average brightness of the scene should be around 0.5 because the displayable intensity range is from zero to one. If the average brightness is known, the exposure value can be determined with

```
Exposure = 0.5 / Average_Brightness;
```

Keep in mind that you wish the exposure to slowly adapt and not change instantaneously. To do this, you could use the following code:

```
Exposure = lerp(Exposure, 0.5 / Average_Brightness, Exposure_Adjust_Speed);
```

You need to determine the average brightness of the scene, which isn't an easy task. One way you can do it within a DirectX or OpenGL application is to lock the texture and calculate the average procedurally by going through all the pixels and manually computing the average. However, on most hardware, this has severe performance implications.

Another approach to approximate the same result is to use render targets to compute the average by taking advantage of the rendering hardware. If you have a successive set of renderable textures, each one being half the size of the preceding one, you can use a simple box filter to copy the initial scene from render target to render target. After you reach a one-by-one render target, the single pixel remaining will be the average value for the initial scene. You can then use this final one-pixel texture as an input to your shader to do your automatic exposure control.

## Time for Some High Dynamic Range Effects

Now that you know a few of the important basics required to implement high dynamic range effects, it is time for the meat of this chapter. Over the next few sections, I will teach you how to implement most of the common HDR effects, including glare, lens flare, and streaks.

### Your First HDR Shader: The Glare!

One of the most commonly seen high dynamic range effects is the glare, or glow, or even bloom . . . In this section, I will show you how to create a shader to emulate this effect in an easy to use way.

The glare phenomenon is caused by the energy from a bright source of light not only exciting the area it contacts but also leaking onto neighboring areas. Because of this leaking, this phenomenon can easily be simulated using a properly selected sequence of blur filters.

The first aspect to consider is which blur filter to use. Because you will want the blurring to be smooth and consistent, a 9-samples Gauss blur filter will do fine. You'll recall that we used this filter in Chapter 7. The other advantage to using this Gauss filter is that you can control the blurriness of the output by deciding in how many passes you will apply it to your scene.

Because you wish to have the scene sufficiently blurred and to lessen the impact on the graphics processor, all the glare blurs will be done on a 1/4-by-1/4 size render target. When enough blur has been applied to the original render target, the only thing needed is to additively blend this blurred image onto the screen in a way where only the brightest parts show up.

**note**

Because RenderMonkey does not offer the functionality to select a render target of a size that is proportional to the size of the screen, you can approximate by making an estimation—which should be sufficient—and setting this value manually in RenderMonkey.

Starting with the blur filter, the use of floating-point textures has no real impact, and the pixel shader for a specific blurring pass should be as follows:

```
float viewport_inv_width;
float viewport_inv_height;
sampler Texture0;
const float4 samples[9] = {
    -1.0,      1.0,      0,       1.0/16.0,
    -1.0,      1.0,      0,       1.0/16.0,
     1.0,     -1.0,      0,       1.0/16.0,
     1.0,      1.0,      0,       1.0/16.0,
    -1.0,      0.0,      0,       2.0/16.0,
     1.0,      0.0,      0,       2.0/16.0,
     0.0,     -1.0,      0,       2.0/16.0,
     0.0,      1.0,      0,       2.0/16.0,
     0.0,      0.0,      0,       4.0/16.0
};


float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   float4 color = float4(0,0,0,0);

   // Sample and output the averaged colors
   for(int i=0;i<9;i++)
```

```
    {
        float4 col = samples[i].w*tex2D(Texture0,texCoord+
                     float2(samples[i].x*viewport_inv_width,
                     samples[i].y*viewport_inv_height));
        color += col;
    }
    return color;
}
```

The next consideration is how many blurring passes are needed to get a good result. This is essentially a trial and error type of task. To save you time, I have included results for different numbers of blur passes in Figure 8.4.

Because a single blur pass isn't enough to get the whole blurring task done, you need to repeat the process multiple times. For this shader, I have done six passes, keeping the blur for the past two blurring passes. These intermediate blurring results will be used within the glare shader a little later. Keep in mind that you need to create multiple render targets to account for this and for the intermediate results needed.

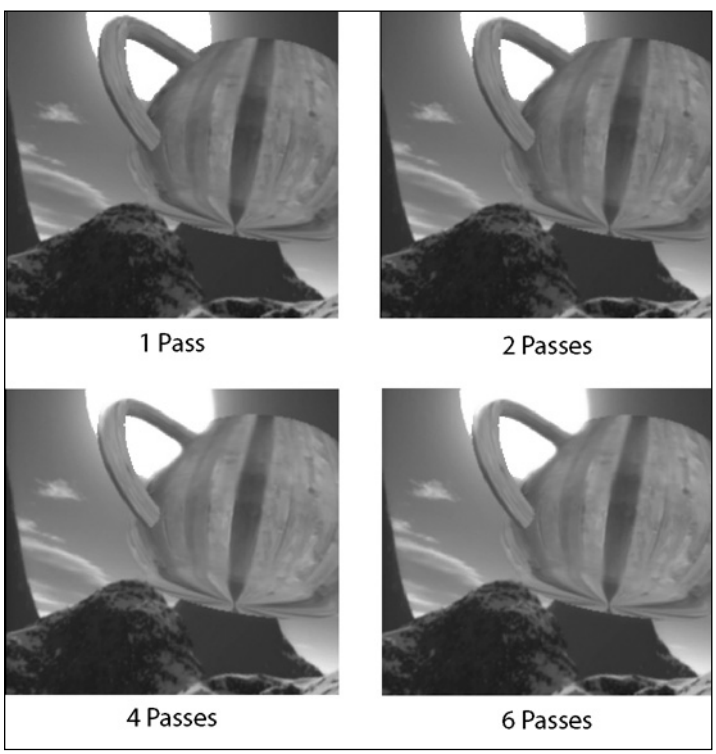



**Figure 8.4** Different levels of blurring for use in the glare effect.

The last missing component to this shader is the final pass where the blurred and regular scene render targets are combined. Although this could be done as separate alpha blending passes for each of our source textures, the easiest way to do this under a pixel shader 2.0-compatible configuration is to sample both render targets in a single pass and use the pixel shader code.

To combine the three blurred versions of your HDR render target, you simply need to combine the values within the pixel shader. To do this, you must define three constants named `Glow_Factor`, `Glow_Factor2`, and `Glow_Factor3`. These variables determine the contribution of each blurred render target. The following pixel shader code shows how this can be done:

```
float Glow_Factor;
float Glow_Factor2;
float Glow_Factor3;
sampler Texture0;
sampler Texture1;
sampler Texture2;
float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   // Sample 3 glow blurring levels and combine them together
   return
     float4((tex2D(Texture0,texCoord).xyz)*Glow_Factor,1.0) +
     float4((tex2D(Texture1,texCoord).xyz)*Glow_Factor2,0) +
     float4((tex2D(Texture2,texCoord).xyz)*Glow_Factor3,0);
}
```

The final rendering results for the shader are shown in Figure 8.5. The final version of this shader is included on the CD-ROM as `shader_2.rfx`. In the first exercise at the end of this chapter, you will be invited to expand on this shader by using a more complex blur filter than a 9-sample Gauss filter.

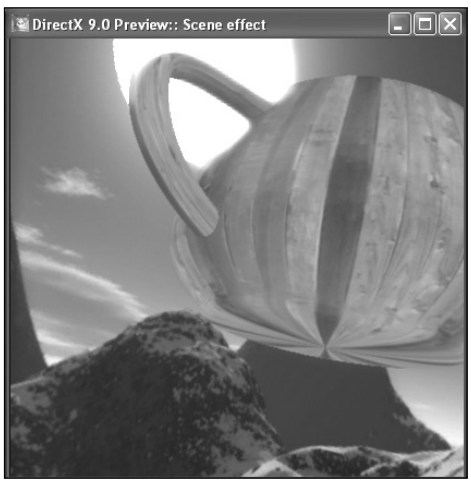



**Figure 8.5** Final rendering results for the Glare HDR effect.

## Time for Some Streaking!

You have probably noticed a star-shaped pattern through your car's windshield that occurs when driving at night. This effect is the result of tiny scratches in the surface of the glass, which cause light to reflect and refract on the surface. The physics behind this phenomenon is fairly complex, but it isn't necessary for our purposes to discuss the underlying cause. We will just develop an approximation, which should produce sufficiently convincing results.

To estimate this effect, you need to create a star pattern. This can be achieved by using a specific blur filter, which acts in a specific direction, causing a streak in that direction. This filter is simple and involves the use of four samples, which are taken along a specific diagonal. Figure 8.6 illustrates how this filter works.

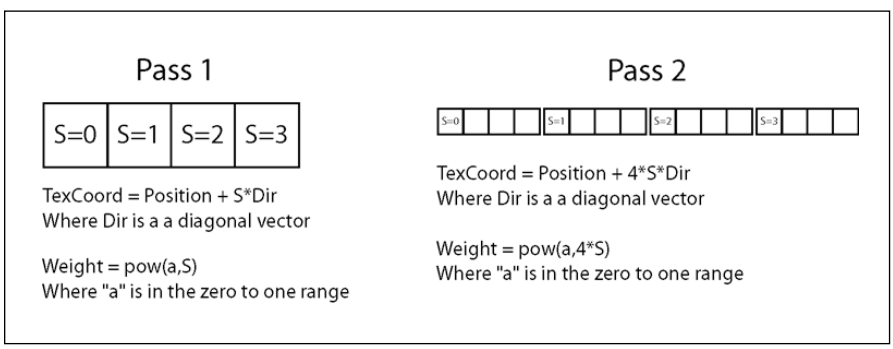



**Figure 8.6** Diagonal filter used to perform the streaking effect.

As you can see from the figure, the position of the sample for each pass is taken at an offset that is proportional to the rendering pass and the sample number. Also, each sample has a weight of `blurFactor`, which is raised to a power that is proportional to the sample number and pass. This allows you to attribute less weight to samples that are farther from the pixel you are rendering. This combination allows the creation of a filter that creates a progressively longer streak in a specific direction with each pass. For example, with one pass, the streak would be 4 pixels long, and with two passes, it would be 16 pixels long.

Translating this basic filter into a pixel shader is simple using the basic array-based approach taken with the other blur filters. Keep in mind that you need the `blurFactor` and `offsetFactor` variables to control the filter for each pass. The result should be a pixel shader similar to the following:

```
float viewport_inv_width;
float viewport_inv_height;
sampler Texture0;
const float  blurFactor = 0.96;
```

```
const float  offsetFactor = 1;
const float4 samples[4] = {
    0.0,  -0.0,   0,   0,
    1.0,  -1.0,   0,   1,
    2.0,  -2.0,   0,   2,
    3.0,  -3.0,   0,   3
};

float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   float4 color = float4(0,0,0,0);

   // Sample and output the averaged colors
   for(int i=0;i<4;i++)
   {
      float4 col = pow(blurFactor,offsetFactor*samples[i].w)*
                   tex2D(Texture0,texCoord+
                   offsetFactor*float2(samples[i].x*viewport_inv_width,
                   samples[i].y*viewport_inv_height));
      color += col;
   }
   return color;
}
```

Figure 8.7 shows the blurred results for a single streak. You need to do the same process for all four directions by changing the signs in the filter table and putting in the suitable `offsetFactor`. Then you only need to combine the four streaks in a way similar to the Glare shader. Note that this time you will need a single glow factor variable because each streak has the same combination. You may also wish to reduce the alpha blending on the final result so it does not oversaturate the scene. Doing so gives the following shader code:


**Figure 8.7** Rendering the shader results for a single streak.

```
float Glow_Factor;
sampler Texture0;
sampler Texture1;
sampler Texture2;
sampler Texture3;
float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   // Combine 4 streak directions
```

```
    return
      min(1.0,
      float4((tex2D(Texture0,texCoord).xyz)*Glow_Factor,0.15) +
      float4((tex2D(Texture1,texCoord).xyz)*Glow_Factor,0.15) +
      float4((tex2D(Texture2,texCoord).xyz)*Glow_Factor,0.15) +
      float4((tex2D(Texture3,texCoord).xyz)*Glow_Factor,0.15)
      );
}
```

The final rendering results for the shader are shown in Figure 8.8. Also note that the final version of this shader is included on the CD-ROM as `shader_3.rfx`.

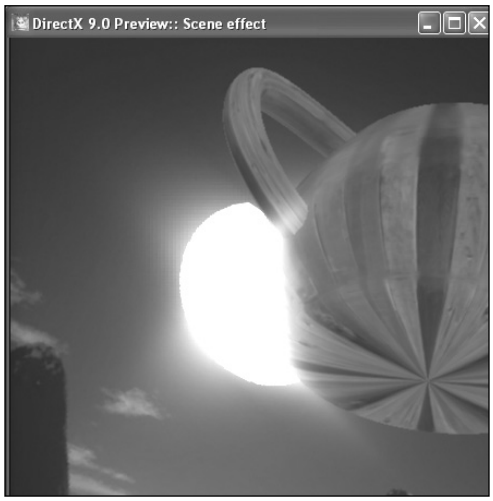



**Figure 8.8** Rendering the Streak HDR shader.

## Lens Flare Free-for-All

The last of the high dynamic range effects worth exploring is the lens flare effect, also called ghost. This effect comes from bright light reflecting between the lenses of the camera, which creates ghost images of the bright components of the image. In this section, I will show how this effect can be re-created with the proper shader code.

Before you write any shader, you need to understand how this effect happens in the first place. Because lens flares originate from reflections between lenses, you need to create mirror images and combine them. The first thing you need to consider is that lenses make rays converge, which has the side effect of creating upside-down mirror images.

With that in mind, you must be able to access render targets in a way that can be scaled and flipped. This can easily be done by changing the way the texture coordinates are passed to the pixel shader. Keeping in mind that a texture is centered around (0.5,0.5), you can scale and flip a texture by using the following vertex shader code:

```
texCoord = (texCoord-0.5)*(Scale) + 0.5;
```

From this code, you can see that the `Scale` variable controls the scale of the texture. If you wish to flip the texture, you simply need to input a negative scale to this equation.

**note**

Keep in mind that because scaling the texture can lead to accesses outside of the texture, you need to set the texture addressing state to `CLAMP` to avoid repeating the render target.

For the effect to look good, you should repeat the same process multiple times. Here, we'll be doing it four times for each pass. You will also need to pick out various scale factors to use. In this example, I have used 2.0, –2.0, 0.6, and –0.6. You may wish to experiment and pick your own values.

Making these adjustments to the vertex shader yields the following code:

```
float4x4 view_proj_matrix;
struct VS_OUTPUT
{
   float4 Pos: POSITION;
   float2 texCoord:  TEXCOORD0;
   float2 texCoord1: TEXCOORD1;
   float2 texCoord2: TEXCOORD2;
   float2 texCoord3: TEXCOORD3;
   float2 texCoord4: TEXCOORD4;
};

VS_OUTPUT vs_main(float4 Pos: POSITION)
{
   VS_OUTPUT Out;

   // Simply output the position without transforming it
   Out.Pos = float4(Pos.xy, 0, 1);

   // Texture coordinates are setup so that the full texture
   // is mapped completely onto the screen
   float2 texCoord;
   texCoord.x = 0.5 * (1 + Pos.x - 1/128);
   texCoord.y = 0.5 * (1 - Pos.y - 1/128);
   Out.texCoord = texCoord;

   // Compute the scaled texture coordinates for the ghost images
   Out.texCoord1 = (texCoord-0.5)*(-2.0) + 0.5;
   Out.texCoord2 = (texCoord-0.5)*(2.0) + 0.5;
   Out.texCoord3 = (texCoord-0.5)*(-0.6) + 0.5;
   Out.texCoord4 = (texCoord-0.5)*(0.6) + 0.5;

   return Out;
}
```

The pixel shader side of this effect is essentially a matter of sampling the texture four times and combining the results. There is one little issue, however, that is a result of the scaling. If you simply sample the texture four times, you end up with hard borders on the edges

of the render target. To remove this effect, you must apply a circular mask, like the one shown in Figure 8.9, which masks out the pixels on the edges.

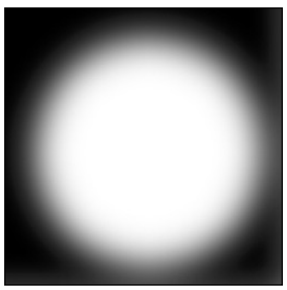


**Figure 8.9** Texture used to mask out hard edges on the lens flare effect.

With this, you need to sample four render targets and four mask values. The mask values are modulated by the render target color for each sample, and the final results are added together and scaled by `Glow_Factor`. This factor helps control the intensity of the glow effect and is determined through experimentation. Applying these changes to the pixel shader yields the following code:

```
float viewport_inv_height;
float viewport_inv_width;
float Glow_Factor;
sampler Texture0;
sampler Texture1;
float4 ps_main(float2 texCoord:  TEXCOORD0,
               float2 texCoord1: TEXCOORD1,
               float2 texCoord2: TEXCOORD2,
               float2 texCoord3: TEXCOORD3,
               float2 texCoord4: TEXCOORD4) : COLOR
{
   // Sample all ghost pictures
   float4 col1 = tex2D(Texture0, texCoord1)*tex2D(Texture1, texCoord1).a;
   float4 col2 = tex2D(Texture0, texCoord2)*tex2D(Texture1, texCoord2).a;
   float4 col3 = tex2D(Texture0, texCoord3)*tex2D(Texture1, texCoord3).a;
   float4 col4 = tex2D(Texture0, texCoord4)*tex2D(Texture1, texCoord4).a;

   // Combine the ghost images together
   return (col1+col2+col3+col4)*Glow_Factor;
}
```

Because you want multiple ghosts, you must repeat this process a second time using the result from the first pass as the source of the following pass. The two passes combined yield 16 ghost images, which should be plenty.

The last step needed is to render the results onto your scene. This can be done through the use of alpha blending and the following pixel shader code:

```
float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   return float4((tex2D(Texture0,texCoord).xyz),0.8);
}
```

With these changes, you can compile the shaders and see the results. The final rendering results for the shader are shown in Figure 8.10. Also note that the final version of this shader is included on the CD-ROM as `shader_4.rfx`.

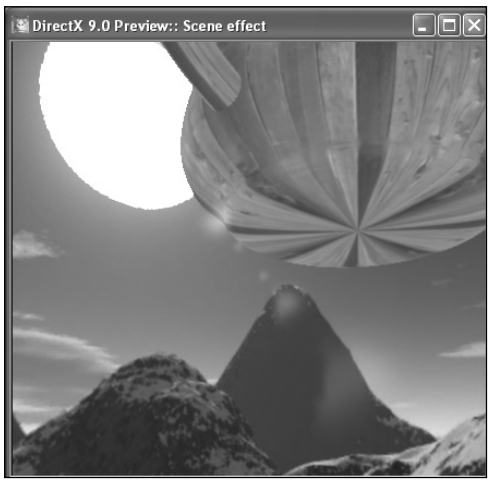



**Figure 8.10** Rendering the ghost HDR shader.

## Putting It All Together

Putting all the effects together is simply a matter of combining all the render targets and render passes of each individual effect. This can be done with the Copy and Paste options on the right-click menu.

Beyond this, all that is needed is a new final pixel shader that takes the results of each individual effect and combines them. The following pixel shader code shows how this can be done:

```
float Streak_Factor;
float Ghost_Factor;
float Glow_Factor;
float Glow_Factor2;
float Glow_Factor3;
sampler Texture0;
sampler Texture1;
sampler Texture2;
sampler Texture3;
sampler Texture4;
sampler Texture5;
sampler Texture6;
sampler Texture7;
```

```
float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   float4 col;

   // Glow
   col = float4((tex2D(Texture1,texCoord).xyz)*Glow_Factor,1.0) +
         float4((tex2D(Texture2,texCoord).xyz)*Glow_Factor2,0) +
         float4((tex2D(Texture3,texCoord).xyz)*Glow_Factor3,0);

   // Ghost
   col += float4((tex2D(Texture0,texCoord).xyz),0);

   // Streak
   col +=
     float4((tex2D(Texture4,texCoord).xyz)*Streak_Factor,0) +
     float4((tex2D(Texture5,texCoord).xyz)*Streak_Factor,0) +
     float4((tex2D(Texture6,texCoord).xyz)*Streak_Factor,0) +
     float4((tex2D(Texture7,texCoord).xyz)*Streak_Factor,0);

   return col;
}
```

This is all that is needed to get this working. You may need to adjust the different variables so that all of the effects blend well together. You can see a tuned version of the output in Figure 8.11. This shader is included on the CD-ROM as `shader_5.rfx`.


**Figure 8.11** Rendering the final HDR shader.

## Solutions for Today's Hardware

Although the use of floating-point textures and render targets is the ideal way of implementing high dynamic range effects, it is not necessarily well suited to today's hardware. At the time of this writing, support for such features is sparse, and its speed is not so good. Although in the future, floating-point textures will be the way to go, you need a solution that works on most of today's hardware architectures.

Because you cannot use floating-point textures, you need a way to represent light intensity with less precision. If you are willing to make the concession that all the color components are at the same intensity level, you can use the alpha channel of your render targets to carry the overall brightness. In other words, the alpha channel will contain a multiplier to be applied to all the color components. This is similar to the way the brightness information was encoded into the environment map earlier in this chapter.

This approach has still another issue. Because the alpha channel values are represented in a zero to one range, you need to redefine what range these values correspond to. For this example, I will assume the range of alpha values is zero to 64, which should give sufficient intensity range and fractional precision. This is simply a matter of multiplying the values from the alpha channel by 64 before using them.

With this basic knowledge, you can now convert the glare or bloom effect to make use of this new approach. Although it may seem like a complex task, it is easier than it looks. Because the glare blurring filters blur both the alpha and the color components of the render targets, there is no need to take any special consideration for this new approach. In fact, the only changes needed are making sure the render targets are of `A8R8G8B8` format and changing the object rendering passes so that the intensity values are scaled down to fit within the alpha channel. The pixel shader code for the object rendering pass becomes the following:

```
float Exposure_Level;
sampler Environment;
float4 ps_main(float3 dir: TEXCOORD0) : COLOR
{
   float4 color = texCUBE(Environment, dir);
   return float4(color.rgb,((1.0+(color.a*64.0))* Exposure_Level)/64.0);
}
```

With this, no other changes are needed to any of the shaders to do the effects. The only exception to this is the final pass where you will need to make modifications to decode the intensity of the pixel. This is done by taking the zero to one alpha value and multiplying it by 64 and using this value to scale the incoming color values. The resulting pixel shader code is as follows:

```
float Glow_Factor;
```

```
float Glow_Factor2;
float Glow_Factor3;
sampler Texture0;
sampler Texture1;
sampler Texture2;
float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{
   // Sample 3 glow blurring levels and combine them together
   float4 col1 = tex2D(Texture0,texCoord);
   float4 col2 = tex2D(Texture1,texCoord);
   float4 col3 = tex2D(Texture2,texCoord);

   // Mix the three glows together
   return
     float4(col1.rgb*(col1.a*64.0)*Glow_Factor,1.0) +
     float4(col2.rgb*(col2.a*64.0)*Glow_Factor2,0.0) +
     float4(col3.rgb*(col3.a*64.0)*Glow_Factor3,0.0);
}
```

As you can see in Figure 8.12, the rendering results are similar to the one obtained earlier in this chapter. I have also included this shader on the CD-ROM as `shader_6.rfx`. You will be invited in the second exercise at the end of this chapter to apply the same process to the streaking shader developed earlier in the chapter.


**Figure 8.12** Rendering the final HDR shader using non-floating-point textures.

## It's Your Turn!

High dynamic range is one of the effects that can add so much realism to your scene. It is now your turn to learn a little more on the topic of HDR through a couple of exercises. The solutions to these exercises can be found in Appendix D.

### Exercise 1: USING A BIG FILTER

For this first exercise, I invite you to improve on the glare filter developed in this chapter. Your task is simple: take the shader and use the 49-samples Gauss filter used in Chapter 7 instead of the 9-samples filter. This will involve changing the glare filter passes and determining the appropriate number of blur passes needed.

### Exercise 2: STREAKING ON TODAY'S HARDWARE

Throughout this chapter, I have demonstrated how the glare HDR effect could be accomplished through the use of non-floating-point textures. For this exercise, I invite you to use the same process for the floating-point streaking effect developed earlier.

## What's Next?

High dynamic range, or HDR, is a great way to enhance your renderings. Instead of taking into account color only, now you can account for the intensity of light. This enables you to create a richer environment and take into account the wide range of lighting that occurs in real life.

At this point, we have covered many screen-based effects. Such effects are powerful in their ease of use and visual appeal.

In the next chapter, I will be covering the topic of lighting. From the basics to more complex approaches, you will explore how to use light to create a much richer environment for your renderings.