8.1

Light Shaft Rendering

Jason L. Mitchell

Introduction

This article discusses a technique for rendering volumetric light shafts, which is an improvement over methods used by most games today. This technique applies slice-based volume rendering methods to the task of visualizing volumetric light shafts. Previously, Dobashi and Nishita have published a series of papers on applying slice-based volume rendering to this task [Dobashi02]. Here, we will present several improvements to Dobashi and Nishita's techniques, which are made possible by current graphics hardware with at least ps_2_0 shader support.

Light Shafts in a Scene

Typically, in real-time 3D graphics, we use simple fog models and particle systems to simulate the effect of light scattering back to our eye from particulate matter suspended in the air. These effects are important in giving us a sense of scale and otherwise enhancing the realism in our scenes, but we can do better on hardware with sophisticated shaders and high fill-rate. The effect that we will achieve in this chapter is the visualization of lit particulate matter or "participating media" in our scene. The particulate matter can be non-uniform in density and will be correctly shadowed by the scene as shown in Figure 8.1.1.

FIGURE 8.1.1 *Real-time volumetric light shafts.*

Previous Approaches

Most games render light shafts by drawing polygons representing the bounding region of the desired light shafts as shown in Figure 8.1.2(a). The polygons are blended with the scene and, when used well, are reasonably good at giving the sense of illuminated particulate matter in the air. The polygons are usually textured with a noise texture which may have slowly animating texture coordinates to give the sense of dust wafting gently through the space.

Some games further dice up this pyramidal light volume with additional polygons as shown in Figure 8.1.2(b) [Lepage04]. This gives an additional sense of parallax as the camera moves relative to the light shaft, since points *within* the volume are shaded. While drawing the bounding volume of the desired light shafts is usually cost effective, it is not always convincing. The illusion tends to break down as the user moves relative to the volume, particularly if the volume ever gets clipped by the camera's front clip plane.

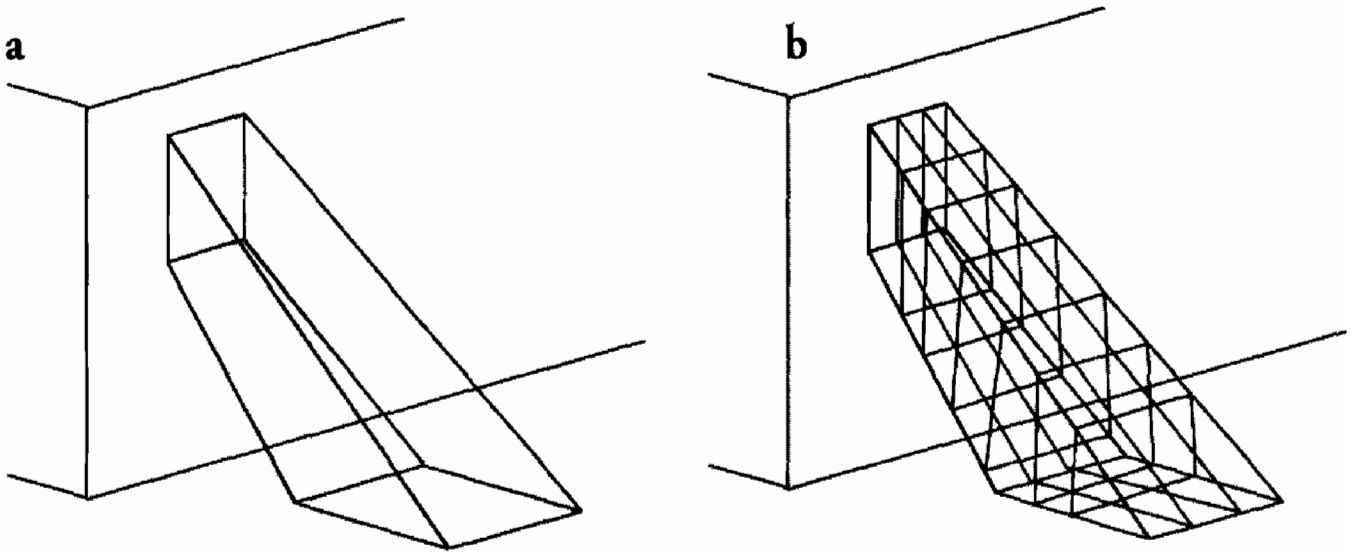

FIGURE 8.1.2 *Light-space sampling planes. (a) extruded window, (b) diced extruded window.*

Another class of approaches has been explored by [Mech01] and [James03]. These two articles discuss approaches in which the lengths of rays through a given volume are computed by adding and subtracting depths of front and back facing polygons which represent the bounding volume of the particulate matter. These are clever techniques which can integrate naturally with the geometry of 3D scenes, but they do not allow for light color variation, non-uniform density, or complex shadowing.

Volume Rendering Approach

The approach discussed in this chapter is based upon techniques used in the scientific visualization community, specifically *slice-based volume rendering*. Instead of rendering geometry which is fixed in world space or the space of the light casting the light shafts, we render geometry which is consistently oriented in view space in order to integrate along rays through light volume.

After the rest of the scene has been drawn, a series of planes are drawn perpendicular to the viewing direction, as shown in Figure 8.1.3. These planes are additively blended with the frame buffer in order to accumulate the light that the particulate matter in the scene scatters back along the rays to the viewpoint through each pixel.

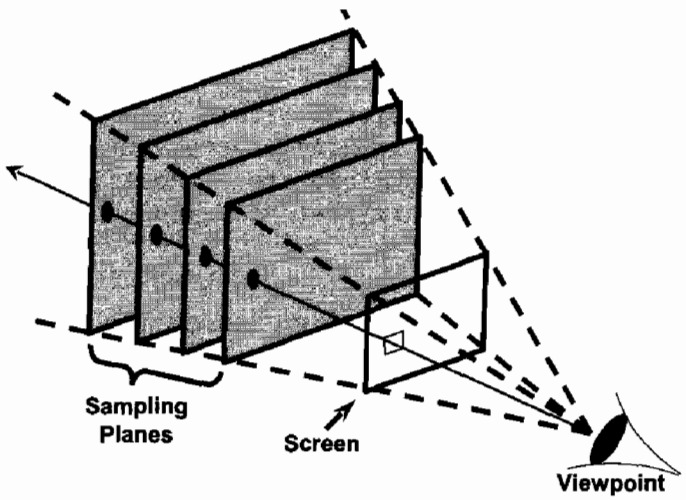

FIGURE 8.1.3 *Integration using sampling planes.*

Whenever you are shading in general, you are computing the amount of light scattered back to the eye through a given pixel, but you can get away with rendering just the simple polygonal surface of objects, which are opaque. In the case of semi-transparent volumetric data such as light shafts, you sample along rays *through* the volume in order to sum up all of the scattered light. This is the reason that we draw a set of sampling planes aligned in view space, to allow you to sample along all of the rays through the volumetric light shafts.

These sampling planes are shaded by projecting a number of textures onto them from the position of the light source. The intention is to approximate the integral along a ray from the eye through each pixel on the screen and, in turn, through each sampling plane. The end result should approximate the amount of light scattered back to the eye by the simulated particulate matter in the air (i.e., the light shaft).

Positioning the Sampling Planes

For our purposes, we model lights as projective (think of a flashlight or slide projector). As a result, each light has a position, direction, up vector, horizontal and vertical fields of view, as well as near and far planes. In short, each light has a frustum just like any usual viewer of a 3D scene. A given light source only casts light within its frustum. This is important, because it means that the only region in space that we need to sample is the region inside the frustum.

The vertex data representing the sampling planes must be transformed to the proper position in eye space each frame. This is done with a simple trilinear interpolation trick performed in the vertex shader. The sampling plane vertex buffer contains some number of quads (100 in the example application on the CD-ROM) evenly distributed in z and fully filling the unit cube. The data is layed out this way in the vertex buffer so that the following simple vertex shader code will transform it to fill the view-space-aligned bounding box of the light frustum.

ON THE CD

```
// Trilerp position within view-space-axis-aligned bounding
// volume of light's frustum
float4 pos = vMinBounds * vPosition +
             (vStretchedMaxBounds * (1.0f - vPosition));
pos.w = 1.0f;
```

The view-space-aligned bounds are loaded into the constant store and the shader trilinearly interpolates within this box to place each vertex. This means that the positioning of the planes is very lightweight, as the shader is able to correctly compute their positions with the above code. Figure 8.1.4 shows the light frustum with sampling planes positioned perpendicular to the view direction (from the viewer at left). The sampling planes fully fill the view-space-aligned bounding box of the light frustum.

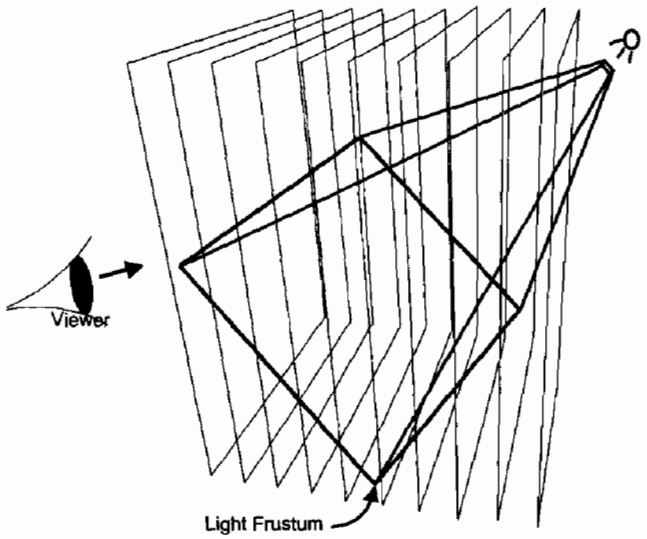

FIGURE 8.1.4 *Sampling planes.*

Shading the Sampling Planes

Now that we have positioned the sampling planes in order to allow us to uniformly sample along the rays from our eye through the light frustum, how exactly should we shade them? There are a number of terms in our lighting equation, each of which adds an additional level of realism and control:

- Light scattering
- Cookie / Gobo
- Shadow
- Noise / Nonuniformity

The first term is the most complex and is used to describe how much light would reach our eye if the particulate matter were of uniform density, while the remaining three terms essentially mask off the scattered light in different ways.

Light Scattering

A lot of work has gone into studying how light scatters in volumes of particles and how this affects what we see. It is well known that applying some sort of fog model to expansive outdoor scenes is fundamental to evoking a proper sense of scale. This is done in all flight simulators and in other 3D applications which have to render large outdoor scenes. Recently, these algorithms have been advanced even further, as many forms of scattering have been incorporated into usable models [Preetham99] [Dobashi02]. Various terms such as the density of the particles, the distances of the light and eye from a given particle, and angular falloff all contribute to the amount of light scattered to the viewer. Since some of these (such as angular falloff) can be baked into the cookie (see next section) and because we plan to render non-uniformly dense particles anyway, we ignore all of these terms except for the $1/\text{distance}^2$ light intensity falloff, which is the most important aspect of light scattering. You can think of the amount of light scattered to our eye from a given particle as simply a scalar multiplied by $1/\text{distance}^2$, with all of the other terms of our lighting equation potentially masking this off. To illustrate the effect of the $1/\text{distance}^2$ term, Figure 8.1.5 shows a spotlight emitting a cone of light with different kinds of falloff terms. Figure 8.1.5(a) shows a conic volume of light with no falloff at all. Figure 8.1.5(b) shows a falloff which is proportional to one over the distance from the light source. Figure 8.1.5(c) shows a fall off which is proportional to $1/\text{distance}^2$, which gives an effect that is reasonably close to what one observes in the real world. In our example application, we chose to add a small ambient term to the $1/\text{distance}^2$ term as shown in Figure 8.1.5(d).

Throughout the rest of this article, we will be masking off the intensity of this scattered light with three more terms: a light cookie, shadows, and noise.

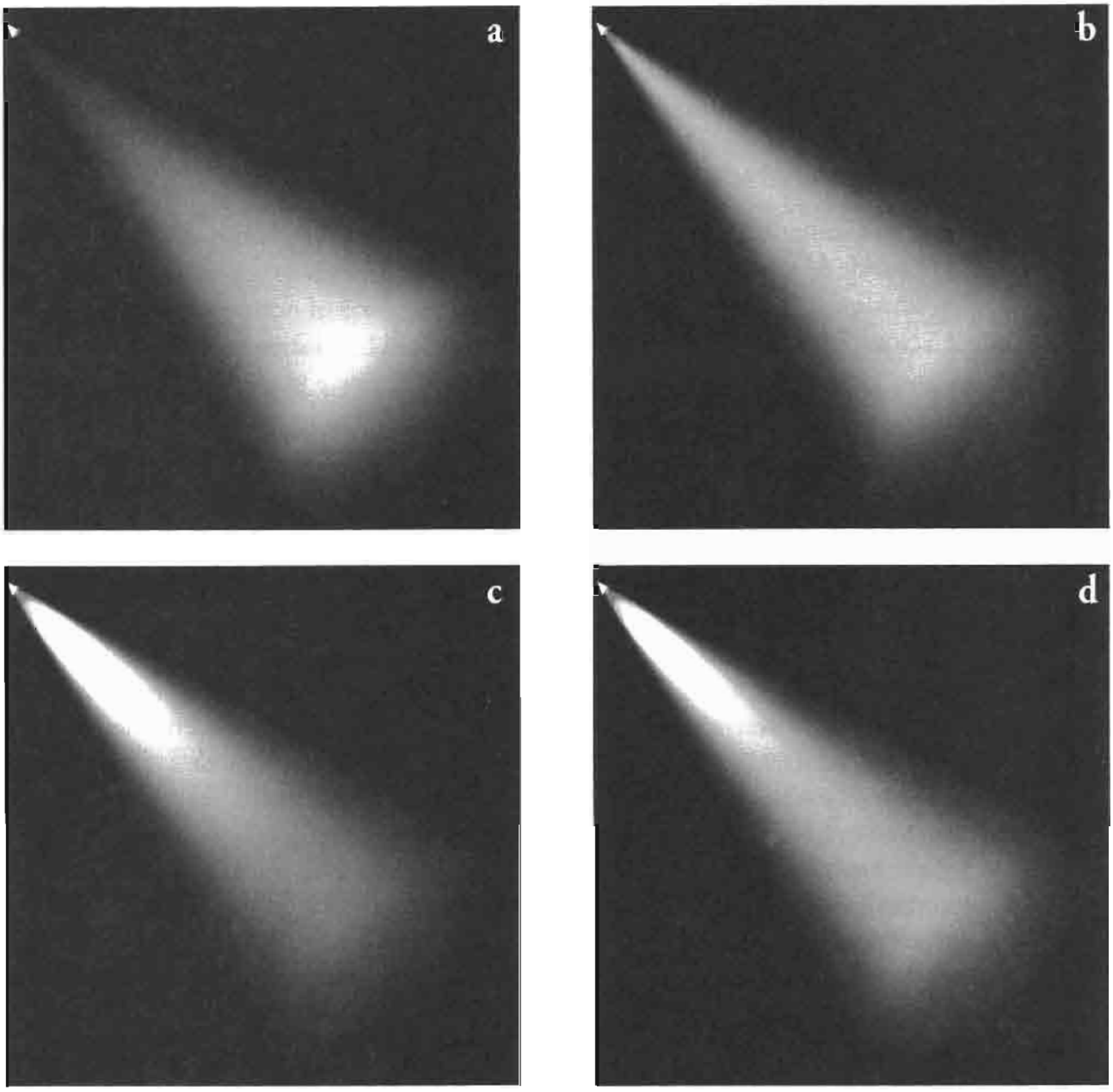

FIGURE 8.1.5 *Various options for light intensity. (a) constant intensity, (b) 1/distance fall-off, (c) 1/distance^2 fall-off, (d) 1/distance^2 + ambient fall-off.*

The Light Cookie

The second term in our lighting equation is commonly referred to as a *cookie* or a *gobo* in stage lighting. It is a simple cut-out, which is used to shape the light cast onto a scene including, in our case, particulate matter in the air. Different cookies can be used to shape the resulting light shafts in interesting ways, as is commonly done in theater and film. Examples of cookies are shown in Figure 8.1.6.

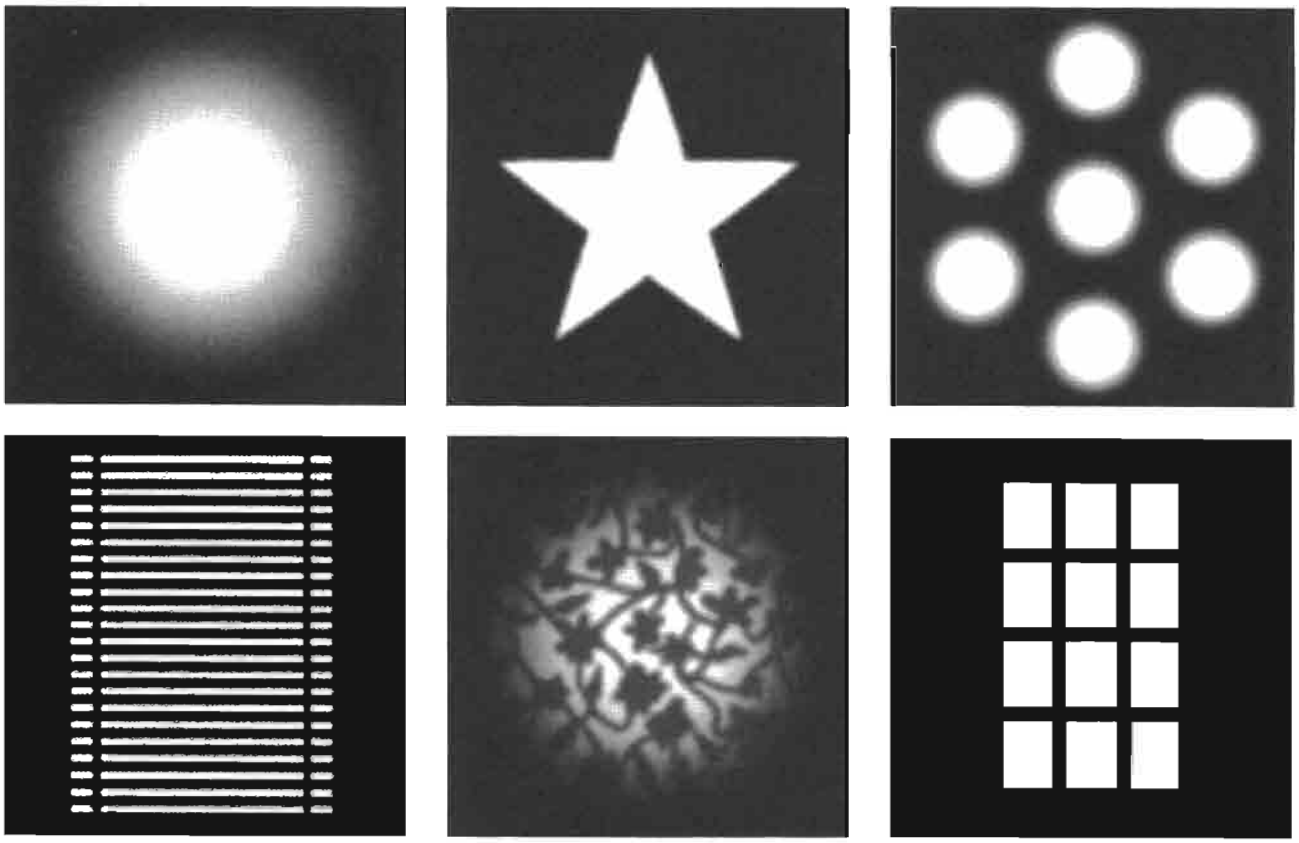

FIGURE 8.1.6 *A variety of cookie or gobo textures.*

Shadows

The next term used to mask off scattered light is the shadow term of our lighting equation. For any 3D point in the light frustum, we must be able to determine whether or not the point is shadowed by the rest of the scene. The most natural tool to use for this task is a *shadow map* [Williams78]. To create the shadow map, we must render the depths of our scene from the point of view of our projective light source into a separate texture, which we may update every frame. Figure 8.1.7 shows a typical shadow map for our test scene. Distance from the light source is stored as a scalar, hence black is near and white is far.

When rendering the light shafts, the shadow map is projected onto the sampling planes with projective texture mapping. The light-space depth is interpolated across the sampling planes and can be tested against the value from the shadow depth map at each pixel of each sampling plane. If the interpolated depth is greater than the value from the shadow map then the point must be shadowed by objects in the scene and the scattered light is multiplied by zero to mask it out. Figure 8.1.8 shows the result of including shadows in the lighting equation. The streaks through the light volume caused by the shadows give a very dramatic look, particularly when the light or the viewer (or both) are in motion.

FIGURE 8.1.7 *Shadow depth map.*

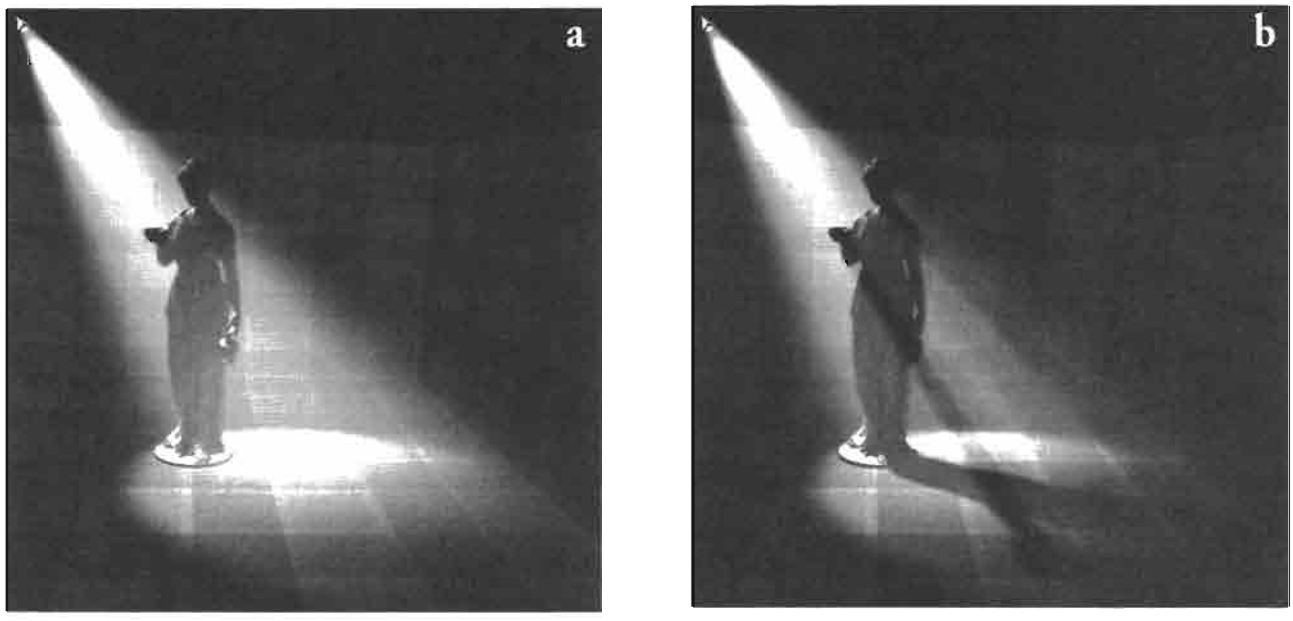

FIGURE 8.1.8 *Applying shadows using a shadow map. (a) no shadows, (b) with shadows.*

Noise

An easy way to make the light shafts appear even more interesting and volumetric is to project noise onto the sampling planes. This gives the impression of non-uniformity of the particulate matter in the air and just generally adds visual interest. In our example application, two repeating (tileable) 2D grayscale noise maps, shown in Figure 8.1.9, are projected onto the sampling planes and are scrolled slowly in different directions.

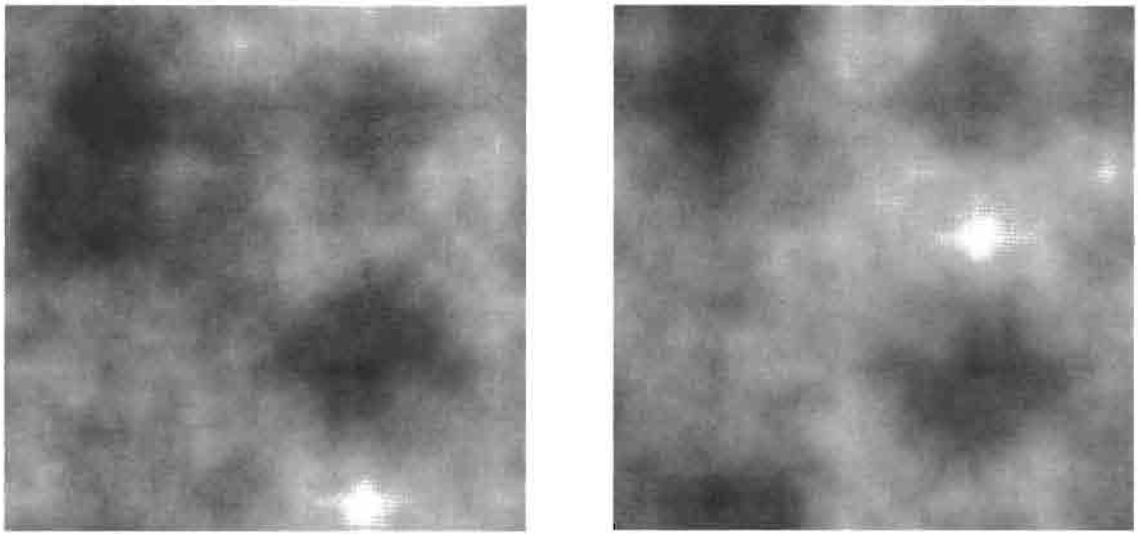

FIGURE 8.1.9 *Projective tiling noise textures.*

Compositing these two noise maps together gives a realistic dynamic appearance to the light shafts. Figure 8.1.10 shows the scene with and without the use of noise. Of course, the true impact of the noise is best perceived in motion.

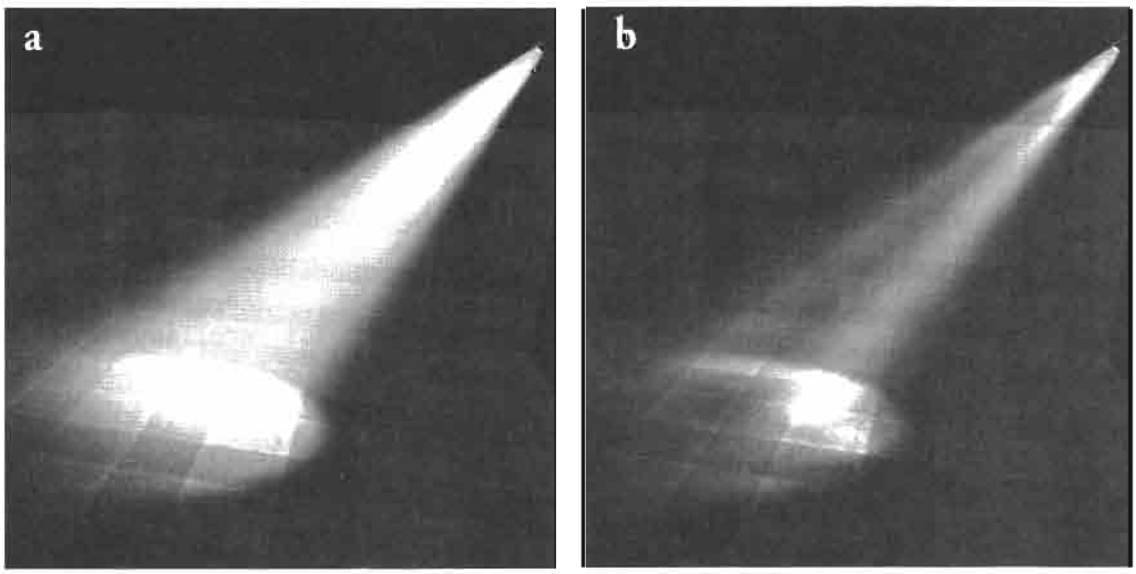

FIGURE 8.1.10 *Projecting noise into the light volume. (a) no noise, (b) with noise.*

Light Shaft Pixel Shader

Now that we have described the various terms in our lighting equation, we can have a look at the pixel shader which computes the final scattered light intensity for each sample in our light volume. The following DirectX® High Level Shading Language (HLSL) shader is used by our example application to do this math.

As you can see from the function prototype, there are three sets of projective texture coordinates interpolated across the sampling planes. These are used to sample the shadow map and the two noise maps. In addition to these projective texture coordinates, the light space position and the light space depth are interpolated in another texture coordinate. These will be used in the shader to compute distance attenuation and in the shadow mapping computation. The final interpolated parameter acts as a mask which is used to route our final intensity into the proper channel of our output buffer. This will be discussed in more detail later in the section on quality.

```
float4 ps_main(float4 tcProj : TEXCOORD0,
               float4 tcProjScroll1 : TEXCOORD1,
               float4 tcProjScroll2 : TEXCOORD2,
               float4 lsPos_depth : TEXCOORD3,
               float4 ChannelMask : COLOR0,

               // Uniforms to generate shader permutations
               uniform bool bScrollingNoise,
               uniform bool bShadowMapping,
               uniform bool  bCookie) : COLOR
{
    float compositeNoise = 0.015f;
    float shadow = 1.0f;
    float4 cookie = {1.0f, 1.0f, 1.0f, 1.0f};

    float shadowMapDepth;
    float4 output;

    if (bCookie)
    {
        // Sample the cookie
        cookie = tex2Dproj(CookieSampler, tcProj);
    }

    if (bScrollingNoise)
    {
        float4 noise1 = tex2Dproj(ScrollingNoiseSampler,
                                  tcProjScroll1);
        float4 noise2 = tex2Dproj(ScrollingNoiseSampler,
                                  tcProjScroll2);
       compositeNoise = noise1.r * noise2.g * 0.05f;
    }

    shadowMapDepth = tex2Dproj(ShadowMapSampler, tcProj);

    if (bShadowMapping)
    {
```

```
        if (lsPos_depth.w < shadowMapDepth)
            shadow = 1.0f; // The pixel is in light
        else
           shadow = 0.0f; // The pixel is occluded
    }

    // Compute attenuation 1/(s^2)
    float atten = 0.35f + 20000.0f /
                  dot(lsPos_depth.xyz, lsPos_depth.xyz);

    float scale = 9.0f / fFractionOfMaxShells;

    float intensity = compositeNoise * luminance(cookie.rgb) *
                      scale * atten * shadow;

    // Route intensity to correct channel
    return intensity * ChannelMask;
}
```

After the interpolated parameters, you see three parameters of type `uniform bool`, which are used for code specialization. These inputs are known at compile time (because they are uniform) and are used by the compiler to optimize out dead code. Since this pixel shader is used in the D3DX Effects framework, a given technique generates a specialized version of this pixel shader by passing Booleans to the shader as shown below.

```
technique clip_nonoise_shadow_cookie_technique
{
    pass P0
    {
        VertexShader = compile vs_1_1
                       vs_project_scrolling_noise_1();
    PixelShader  = compile ps_2_0 ps_main(false,   // noise
                                          true,    // shadow
                                          true);   // cookie
    ...
```

This technique uses the shadow map and the cookie, but no noise. A specialized version of the shader, which doesn't do any of the noise sampling or compositing, is generated by the compiler and used by the application at runtime if noise is not needed.

Looking back to the pixel shader itself, you can see the various blocks of the shader which are bracketed by `if` tests of these three uniform Boolean variables. What this means is that it is only necessary to maintain this one HLSL main function despite the fact that many different shaders are generated for use at runtime.

The first block of the shader, which is bracketed by a uniform Boolean, is the cookie sampling. By default, the cookie color is white, but this default color will be overridden if `bCookie` is true.

Subsequently the noise texture may be sampled. You'll note that the same texture is sampled twice with two different sets of projective texture coordinates. This is because we have chosen to store our two different repeating grayscale noise maps in the red and green channels of the same actual texture map for convenience. These two samples are then composited together, extracting out the two different scalar channels we are looking for.

After the noise is sampled and composited together, we optionally sample our shadow depth map and compare against the interpolated depth. If the shadow depth map sample is nearer the light, then our sample is in shadow and our shadow term is zero. If the shadow depth map sample is farther from the light, then our sample is in light and our shadow term is one.

Since we always compute distance attenuation, we compute the value of the variable `atten` in code, which is not bracketed by a uniform Boolean variable. Since we are drawing our sampling planes as large quads that are not tessellated, we cannot compute distance from the light in the vertex shader and interpolate it. Instead, we interpolate the 3D light space position and compute the $1/\text{distance}^2$ attenuation term in the shader by taking the reciprocal of the light space position dotted with itself. This is further tweaked with a few magic constants which are a function of the scale of the world used in the example application. The variable `scale` is used to scale the contribution of each sampling plane in the event that less than the maximum number of planes is drawn. As an optimization, the example application will scale down the number of sampling planes used if the user moves far away from the light volume, hence, this scale factor must be dynamic. Finally, the terms are all multiplied together, giving our three masking terms (cookie, shadow, and noise) an opportunity to mask out the light scattering term. The very last operation in the pixel shader multiplies the scalar intensity computed above with a 4D vector called `ChannelMask`. This effectively routes the scalar intensity to one of the four channels of our render target. The motivation for this will become clear as we discuss ways to improve the quality of this effect.

Quality

As discussed earlier, this technique relies upon sampling along rays through a volume which contains lighting information. In graphics, we are constantly sampling a variety of different signals and any time that we perform sampling, we must consider antialiasing. This is true when rasterizing polygons, which have hard edges, and sampling from texture maps, which may have high spatial frequency components. When rasterizing, we apply multisample full-scene anti-aliasing and when sampling from textures, we antialias the textures by performing some sort of filtering such as trilinear filtering of mipmaps.

Likewise, when we are reconstructing volume data, we must be careful not to take too few samples along each ray and miss high-frequency components in the signal, such as the shadow map edges. This means that we want to use as many sampling

planes as possible in order to increase our sampling frequency. We could also try and filter the components of our lighting equation, particularly the shadow map in order to possibly get away with fewer sampling planes [Reeves87]. In our example application, however, we have chosen to use inexpensive pick-nearest filtering of the projective shadow texture. If we use too few sampling planes, we will not reconstruct the shadow edges well enough and will get stair step artifacts as shown in Figures 8.1.11(a) and 8.1.11(b). These two images show our scene with 25 and 50 sampling planes respectively, in order to illustrate the perils of undersampling the volume. In Figure 8.1.11(c), which was drawn with 100 sampling planes, the stair step artifacts are essentially gone.

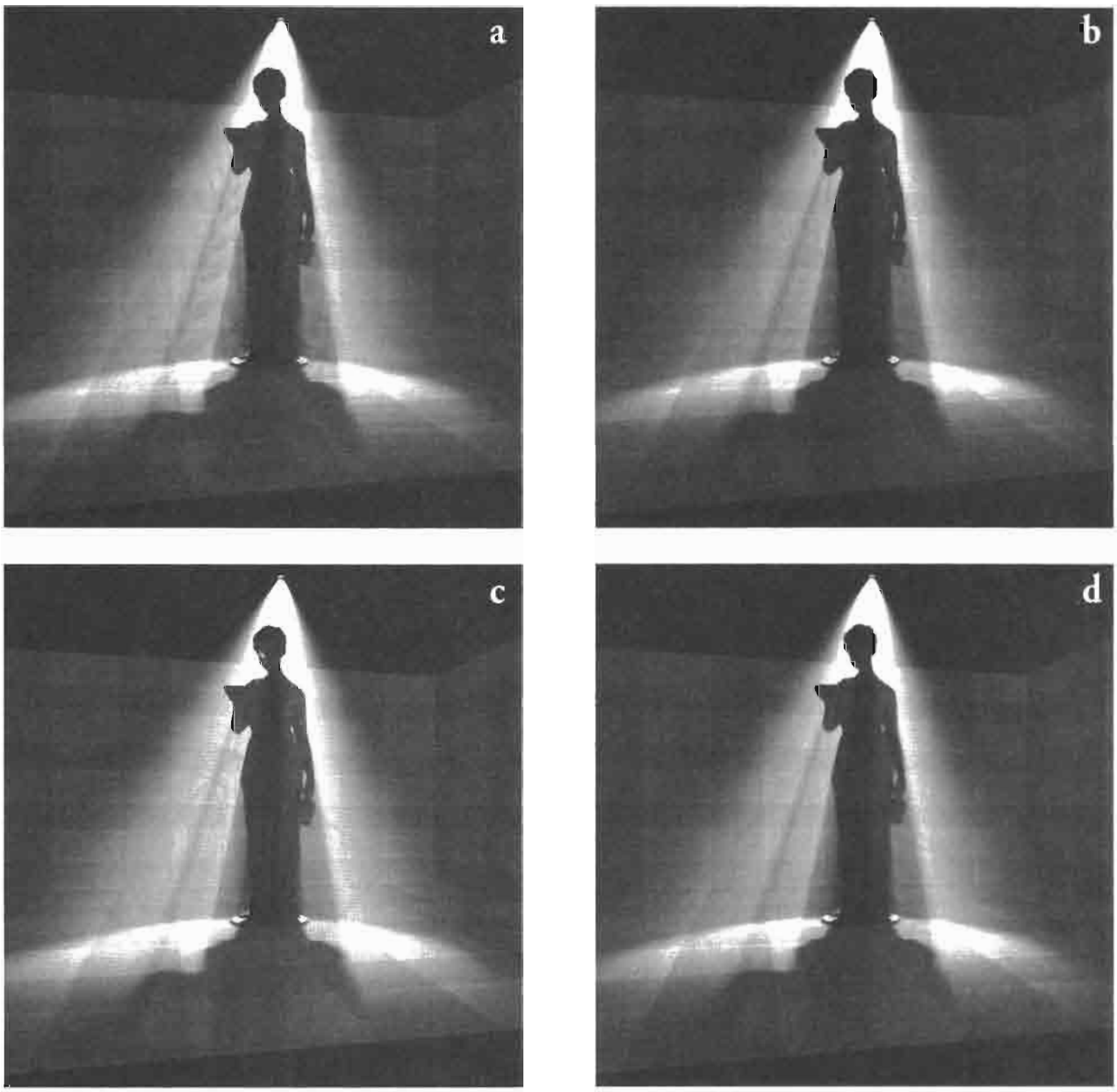

FIGURE 8.1.11 *Varying numbers of sampling planes. (a) 25 planes, (b) 50 planes, (c) 100 planes, (d) 100 planes with blurring.*

As mentioned above, in order to antialias the sampling of the shadows, we want to draw as many sampling planes as possible. On current graphics architectures, we can only perform alpha blending to surfaces which have at most 8 bits of precision per channel. So, if we are drawing a lot of sampling planes, each plane can only add a bit or two of data to the frame buffer. This can lead to quantization or *banding* in the resulting image.

If you can live with a monochrome light color, there is a simple trick that can be employed to dramatically increase precision. In our example application, we have chosen to render into an offscreen RGBA image rather than render directly to the back buffer. We perform additive blending and render each sampling plane into only one of red, green, blue, or alpha by using the `ChannelMask` parameter in our pixel shader shown above. This allows us to accumulate the light from one quarter of the planes in each of our four 8-bit channels, dramatically increasing the number of bits that can be contributed by each sampling plane. When compositing this offscreen RGBA buffer to the final image in the back buffer, we sample it multiple times to provide a small amount of blurring as shown in Figure 8.1.11(d). This softens the look of the light shafts and further reduces aliasing. Of course, this is less needed on hardware which supports blending to surfaces of more than 8 bits per channel.

Efficiency

Since this technique is so fill-rate bound, we want to reduce the number of pixels filled in order to increase performance. As a result, the most important optimization one can make to this rendering technique is the aggressive use of user clip planes to cut down on fill rate. As mentioned earlier, we model each light as a projective light which has its own frustum. While the sampling planes bound this frustum, they also cover a lot of area outside of the frustum as shown in Figure 8.1.12(a).

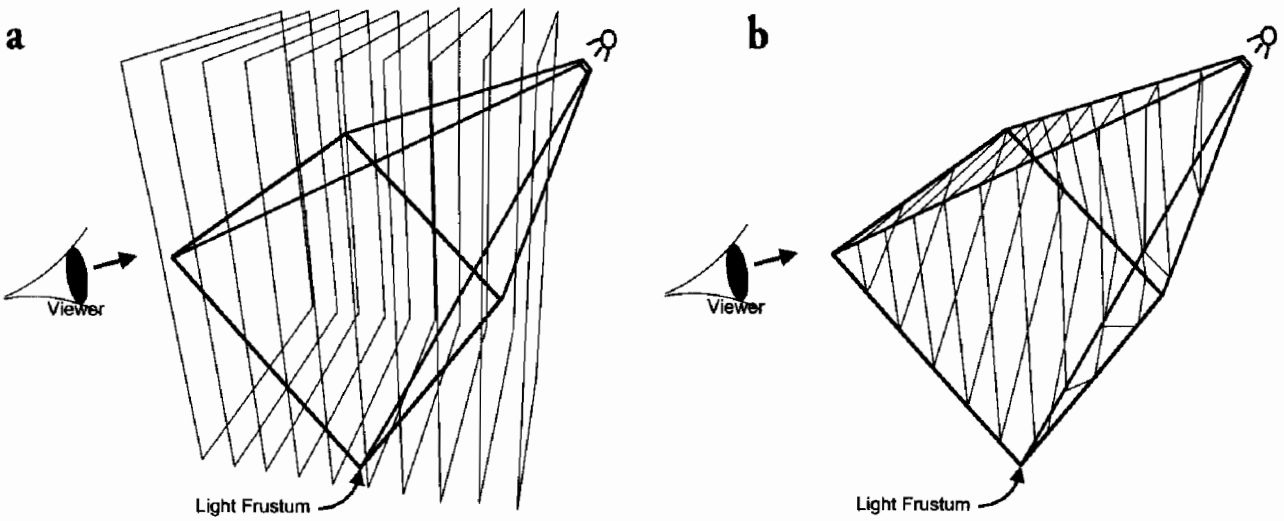

FIGURE 8.1.12 *Clipping to light frustum. (a) unclipped sampling planes, (b) clipped sampling planes.*

Since only regions inside of the light frustum will receive any light, we can safely clip away the parts of our sampling planes which lie outside of the light frustum as shown in Figure 8.1.12(b). For typical viewer and light configurations, this results in a massive performance boost which is essentially the difference between this technique being considered real-time and not real-time on current hardware. In fact, hardware implementations which support user clip planes through raster rather than proper geometric techniques suffer such a massive performance loss as to be non real-time.

In certain scenarios, it may be desirable to further restrict the light volume. For a light shining in a window as in Figure 8.1.2, it may make more sense to use the plane of the window instead of the light's front plane to further clip the sampling planes. Likewise, the floor plane of the room in Figure 8.1.2 may be a better choice than the light's far plane. In short, it is desirable to aggressively minimize the number of pixels filled by whatever means necessary.

Future Enhancements

What we have described thus far is a robust means for rendering volumetric light shafts at interactive rates. One obvious area for improvement is better shadow map filtering [Reeves87]. While this will result in a more expensive pixel shader, it may allow you to get away with using fewer sampling planes. Another area to be explored is a technique called *interleaved sampling*, which employs screen-space dithering techniques to hide aliasing and banding [Keller01]. Interleaved sampling should be reasonably straightforward to integrate into the existing technique. It may also prove useful to move to a ray-casting approach, where different proxy geometry is drawn and where simpler *stopping criterion* shaders can be used to skip over the computation of pixels that are known to be in empty or shadowed regions of the light volume [Krüger03].

Example Application

The sample `ShaderX3_LightShafts` on the companion CD-ROM illustrates the process of drawing volumetric light shafts with the technique outlined in this article.

References

[Dobashi02] Yoshinori Dobashi, Tsuyoshi Yamamoto and Tomoyuki Nishita, "Interactive Rendering of Atmospheric Scattering Effects Using Graphics Hardware," *Graphics Hardware* 2002.

[James03] Greg James, "Rendering Objects as Thick Volumes," *ShaderX²*, Wordware 2003.

[Keller01] Alexander Keller and Wolfgang Heidrich, "Interleaved Sampling," Eurographics Workshop on Rendering Techniques 2001.

[Krüger03] Jens Krüger and Rüdiger Westermann, "Acceleration Techniques for GPU-based Volume Rendering," *IEEE Visualization 2003.*

[Lepage04] Dany Lepage, Personal Communication, 2004.

[Mech01] Radomír Mech, "Hardware-accelerated Real-time Rendering of Gaseous Phenomena," *Journal of Graphics Tools*, 6(3):1–16, 2001.

[Nishita01] Tomoyuki Nishita and Yoshinori Dobashi, "Modeling and Rendering of Various Natural Phenomena Consisting of Particles," Proc.Computer Graphics International 2001.

[Preetham99] Arcot Preetham, Peter Shirley and Brian Smits, "A Practical Analytic Model for Daylight," SIGGRAPH 1999, pp. 91–99, 1999.

[Reeves87] William T.Reeves, David H.Salesin, and Robert L.Cook, "Rendering Antialiased Shadows with Depth Maps," *Computer Graphics* (SIGGRAPH '87 Proceedings), 21(4):283–291, July 1987.

[Williams78] Lance Williams, "Casting Curved Shadows on Curved Surfaces," pp.270–274 SIGGRAPH 1978.