5.2

Per-Pixel Lit, Light Scattering Smoke

Aurelio Reis, **Firaxis Games**

Introduction

We define smoke as a dense collection of particulate matter suspended in air. Steam or even clouds are similar, and we include them in our definition of smoke. Smoke particles are small yet large enough to be influenced by Mie scattering [Nave05]. Accordingly, they exhibit strong forward scattering; that is, all light frequencies scatter roughly in the same direction. Light entering a cloud of smoke reflects, refracts, and is absorbed.

Reeves [Reeves83] was the first to introduce pixel-sized particles to computer graphics; however, they prove too expensive for real-time use. Most past and current games thus model smoke as a few modestly sized, screen-aligned billboards [Moller02]. Although modern processors are ever more powerful, allowing for many more on-screen billboards [Latta04], pixel-sized particles continue to be impractical for today's real-time games. The emphasis is thus on making these multipixel-sized billboards look as much like smoke as possible. For example, using texture maps and texture animations on these billboards is standard practice. We continue this emphasis and introduce per-pixel lighting and light-scattering for smoke particles.

Lighting a screen-aligned particle per pixel requires per-pixel normals. How should one orient the normals of a screen-aligned particle? Erez [Erez05] suggests a number of possible solutions, for example, copying a particle's normals from the surface that emits it or using a particle's velocity vector as a normal. While this solution works modestly well, it leads to artifacts: particles using their velocity vectors as a normal and moving away from a light source remain dark, although intuitively the particle's back should be brightly illuminated.

To allow for normal-mapped lighting, we also require a tangent-basis matrix that transforms a light vector into texture space. ATI's *ToyShop* demo [Tatarchuk06a] includes a particle simulation for running water and raindrops. Since the raindrop particles are screen-aligned, Tatarchuk uses the view matrix as the tangent basis. Because the normal is now screen oriented, the particle always appears to be facing the user. Since smoke

billboards simulate a voluminous spherical mass receiving light from every direction, we employ the same trick.

Another aspect of smoke is its light-scattering property. A standard diffuse illumination model covers light reflection, but in- and out-scattering, as well as absorption, are equally important for smoke. Harris [Harris03] describes a technique in which imposters are used to render complex cloud groups: each cloud particle within a group subdivides the scattering equations. This technique allows for an efficient real-time algorithm for single and multiple scattering but relies upon intermittent update rates to minimize the performance burden. This solution offers impressively accurate results yet is only feasible for slowly evolving environments, such as clouds. It also assumes that sunlight is the major light influencing the particles. Harris's method thus does not easily generalize for simulating smoke.

Although per-pixel lighting has become something of a standard rendering technique, lit smoke still remains outside the realm of most rendering engines. Using graphics hardware that supports Shader Model 3, we show how to light particle systems in a single pass, thus allowing for any blend mode to composite these particles into a scene. We examine how to calculate single scattering for a smoke particle, that is, as if it was a volumetric medium. We then offer a fast, yet faithful, approximation of this calculation. In the appendix, we also mention a technique to procedurally generate smoke textures to help alleviate the next-generation art bottleneck.

This article only examines the lighting of smoke. For guidance on how to avoid clipping and popping artifacts when using billboarded particles, see "Spherical Billboards for Rendering Volumetric Data" [Umenhoffer06] in this book.

Making Better Smoke

Equation 5.2.1 describes subsurface scattering of skin [Struck04] but works equally well to approximate light-scattering of smoke. The light vector $\mathbf{L}$, the view vector $\mathbf{E}$, and a translucency map determine how much light is scattered through and around a smoke particle.

$$\text{translucency} = \min(0, \mathbf{E} \cdot \mathbf{L}) * \text{translucencyMap} \tag{5.2.1}$$

Standard per-pixel lighting then adds the influence of direct light reflection. The result of this simple technique is shown in Figure 5.2.1.

Our technique treats each puff as a cloud mass; large masses of smoke particles thus do not influence their neighbors' brightness. A solution to that problem is outside the scope of this article.

Direct Illumination of Smoke

We calculate all of our lighting contributions in a single rendering pass. Multiple light attributes are set as shader parameters (i.e., uniform constants), and the pixel shader iterates over all lights to calculate the lighting function for each. We find that four lights

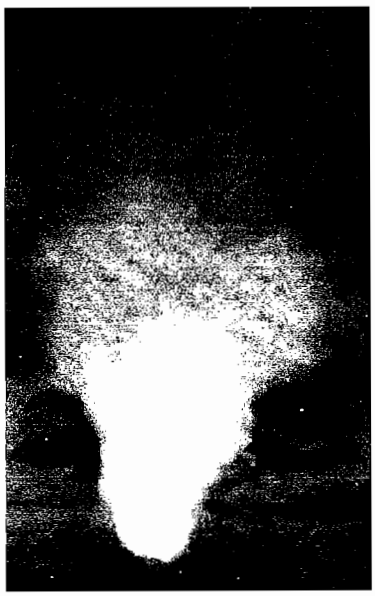

FIGURE 5.2.1 Applying standard diffuse and wrap-around lighting to smoke particles lit from below.

suffice to provide realistic illumination. The remainder of this article thus assumes four light sources. Per-pixel normals are specified through use of a 2D normal map encoded in the DXT-NM format for highest visual quality at the smallest texture size.

The pixel shader evaluates Lambert's diffuse lighting model for each of the four active lights in the scene:

$$\text{diffuse} = N \cdot L \tag{5.2.2}$$

Light attributes (e.g., light direction or position) are passed in as world-space coordinates. Per-pixel normals, however, use tangent space, so we transform these to world space using the inverse-transpose of the tangent matrix. The vertex shader calculates this matrix and then passes it to the pixel shader using texture coordinate interpolators. The vertex shader simply uses the view matrix as the tangent matrix since smoke billboards are always screen-aligned [Tatarchuk06a].

Light Scattering of Smoke

We evaluate Equation 5.2.1 for each light source in the pixel shader. If a light is in front of a smoke particle, then it does not add a scattering contribution. To test for this case we evaluate the dot product of the view vector and the light vector (Figures 5.2.2 and 5.2.3).

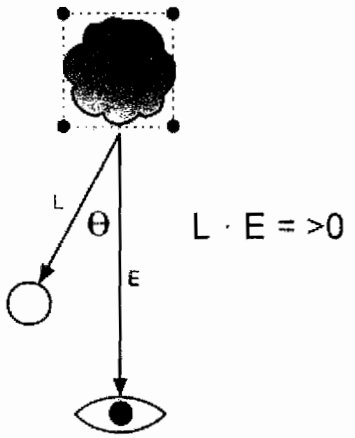

FIGURE 5.2.2 If the cosine of the angle between the light vector and the eye vector is negative, the light is behind the particle.

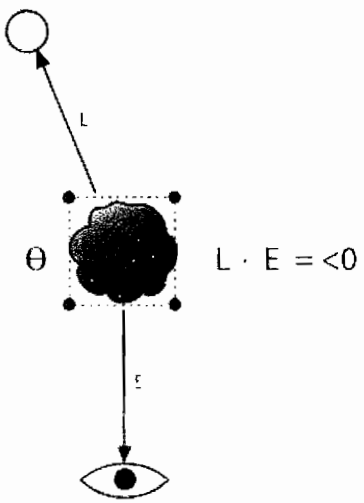

FIGURE 5.2.3 Otherwise the light is in front of the smoke particle.

Then we determine the contribution of the subsurface scattering. For the backlit term we take the dot product between the surface normal (which we sample per pixel in tangent space) and the eye vector. Using an exponential function or remapping the results to a color ramp allows for special effects such as bright light halos around smoke particles.

To hint at the smoke makeup and overall darkness we use a standard diffuse map where the color channels store smoke brightness and the alpha channel stores translucency. To simulate thickness within the smoke volume we sample our translucency map and modulate it by the previous results.

We compute a wrap-around contribution to represent light diffusing through smoke. Equation 5.2.3 generates a light term scale and bias.

$$\text{scale} = 1.0 / (1.0 + w)$$

$$\text{bias} = w / (1.0 + w) \tag{5.2.3}$$

where w represents the wrap-around coefficient, which can range from 0.0 to 1.0 (a constant value of 0.6 is used in the demo program). The scale and bias is then applied to the lighting term given in Equation 5.2.2, which results in Equation 5.2.4.

$$\text{diffuse} = (N \cdot L) * \text{scale} + \text{bias}. \tag{5.2.4}$$

Even though the wrap-around term works best with the diffuse lighting term, it can also be used with the backlighting term given in Equation 5.2.1 (although subjectively the results aren't as effective).

More Realistic Scattering

To improve the light-scattering component of our lighting equation, we project a ray from the light source and trace it per pixel through a height field using a pixel shader. As the ray intersects with the voxels of the height field, a light absorption factor accumulates (Figure 5.2.4). We use this absorption factor to determine how dark a pixel should be owing to smoke thickness as represented by the height field. The main problem with this ray-casting approach is the large number of steps required to integrate along the height field to avoid stepping artifacts. The high number of texture samples required incurs high rendering overhead.

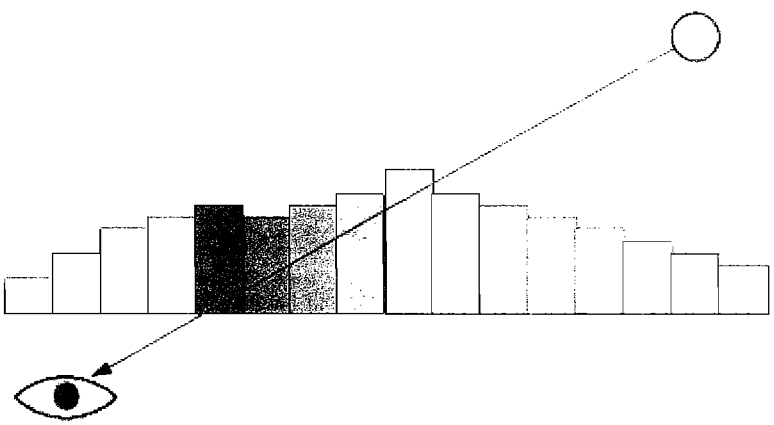

FIGURE 5.2.4 As the eye ray marches through the pseudo-volume it collects the light absorption factor when it intersects with any voxel.

McGuire [McGuire05] employs a similar per-pixel ray-tracing technique to solve parallax bump mapping. Tatarchuk [Tatarchuk06b] improves upon per-pixel ray marching by removing the need for a high and fixed number of ray steps. Using a dynamic sampling rate, it is possible to determine the minimum number of steps for any camera perspective based on the angle between the normal and the view direction:

$$n = \text{nMin} + \text{N} \cdot \mathbf{V}(\text{nMax} - \text{nMin}), \tag{5.2.5}$$

where n is the number of steps to use, nMin is the minimum number possible, nMax is the maximum allowed, N is the normal, and $\mathbf{V}$ is the normalized view vector. It is also possible to adjust the sampling rate based on the current mip-level. This feature is only present on Shader Model 3 hardware, however, since it requires dynamic looping. Tatarchuk [Tatarchuk06b] presents a comparison of three possible hardware implementations of such a technique and describes the tradeoffs and advantages.

This technique generates excellent results and more accurately represents the amount of light potentially scattered through a smoke medium. Despite its high cost, it performs reasonably well on modern graphics hardware and scales gracefully with the number of samples used, using either Shader Model 2 or 3. A major drawback of this algorithm is that it represents only the scattering for an individual particle and not a group of particles. Spawning a large number of particles thus does not occlude adjacent particles' lighting contributions. We are, however, able to dampen the lighting based on the number of particles generated via a dampening parameter passed to the light scattering pixel shader.

Conclusion

In this article we show how to implement realistic-looking smoke. Taking advantage of the relationship between a light, the viewer, and the smoke particle, we create an illusion of light scattering. Adding per-pixel lighting to smoke particles via pixel shaders and using tricks to simulate subsurface scattering, we are able to increase the realism of smoke particles. For a sample implementation of the above technique, examine the accompanying source code and demo application in the article folder on the CD-ROM. We hope you find this technique useful in your own smoke implementations.

References

[Ebert94] Ebert, David, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, 1994.

[Erez05] Erez, Eyal. "Interactive 3D Lighting in Sprites Rendering." GDC, 2005. Available online at *http://www.gamasutra.com-gdc2005/features/20050308/erez_01.shtml.*

[Harris03] Harris, Mark J. "Real-time Cloud Simulation and Rendering." Ph.D. dissertation, University of North Carolina, Technical Report #TR03-040, 2003.

[Hicks03] Hicks, O'dell. "Screen-Aligned Particles with Minimal VertexBuffer Locking." *ShaderX²*, edited by Wolfgang Engel. Wordware Publishing, 2003: pp. 107–112.

[Latta04] Latta, Lutz. "Building a Million Particle System." GDC, 2004. Available online at *http://www.2ld.de/gdc2004*.

[McGuire05] McGuire, Morgan and Max McGuire. "Steep Parallax Mapping." I3D 2005 Poster, 2005. Available at *http://www.cs.brown.edu/research/graphics/games/SteepParallax/index.html*.

[McGuire06] McGuire, Morgan and Andi Fein. "Real-time Rendering of Cartoon Smoke and Clouds." NPAR, 2006. Available at *http://cs.brown.edu/research/graphics/games/CartoonSmoke*.

[Moller02] Akenine-Moller, Tomas. *Real-time Rendering*, 2nd ed. AK Peters, Ltd, 2002: Section 8.3 "Billboarding," pp. 318.

[Nave05] Nave, C. R. "Mie Scattering." Georgia State University, 2005. Available at *http://hyperphysics.phy-astr.gsu.edu/hbase/atmos/blusky.html*.

[Reeves83] Reeves, William T. "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects." *Computer Graphics*, *17*(3), (1983): pp. 359–376, 1983.

[Reichert] Texture Maker, Reichert Software Engineering. Available at *http://www.texturemaker.com*.

[Struck04] Struck, Florian, Christian-A. Bohn, Sebastian Schmidt, and Volker Helzle, "Realistic Shading of Human Skin in Real-time." *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, 2004: pp. 93–97.

[Tatarchuk06a] Tatarchuk, Natalya. "Artist-Directable Real-time Rain Rendering in City Environments." GDC, ATI Research, Inc., 2006.

[Tatarchuk06b] Tatarchuk, Natalya. "Practical Parallax Occlusion Mapping for Highly Detailed Surface Rendering." GDC, ATI Research, Inc., 2006.

[Umenhoffer06] Umenhoffer, Tamás, László Szirmay-Kalos, and Gábor Szijártó. "Spherical Billboards for Rendering Volumetric Data." *ShaderX²*, edited by Wolfgang Engel. Charles River Media, 2006.

Appendix

Generating Smoke Textures

There are a variety of ways to generate good-looking smoke textures. The main rule is to reduce prelighting of a texture while still providing interesting detail. The easiest technique is to start with a photograph of smoke and create an alpha mask to delineate the smoke area. This approach yields good results as long as contrast and brightness of the photograph are adjusted so that any built-in prelighting does not dominate the computed lighting. A high-contrast, monochrome version of the same image is useful as a rough approximation of a normal map.

Another solution is to use procedural texturing techniques. [Ebert94] is an excellent reference for procedural techniques including cellular texturing and multifractal noise. Fractal Brownian motion and turbulence functions (with results weighted by the distance to the center of the image) can be used to great effect. Traditional art packages such as Photoshop and their plug-ins, such as, for example, Photoshop's built-in Clouds or Difference Clouds, are also useful. New procedural art packages (e.g., Texture Maker [Reichert]) generate procedural textures in a WYSIWYG environment using premade noise functions.

Finally, McGuire [McGuire06] employs an interesting technique for his smoke texture generation. He utilizes an artist-specified base 3D shape that is cloned and randomly dispersed throughout a volume. Separate textures then capture the color, normal, and depth information. This technique produces excellent results, although care must be taken if the base model is overly complex. We believe this technique is the most versatile and yields the best results, as it offers artistic freedom and quick turnaround.

Special Effects

In our implementation, particle systems are represented through *special effects*. A special effect is a dedicated system that plays back any number of special effect steps, which may include particle emitters, sounds, lights, screen space effects such as flashes, or even geometric debris. For the purpose of this article we only focus on emitters. Our special effects are exposed through a special effect file format (spfx) that uses XML as a base structure. Examples can be found in the "media/spfx" directory of the accompanying demo program.

The ParticleEmitter step type is a straightforward, yet general particle system implementation. In practical use, however, we urge tailoring a general system to your specific needs. For example, if the particles in your system do not need collision detection, then the existing implementation should be improved to work exclusively on the graphics card, thus reducing the number of vertex/index buffer locks, which are expensive since they consume bandwidth. Hicks [Hicks03] describes how to implement just such a system. Our implementation instead maintains general flexibility. Extending it to support full collision detection is a minor task.