Volumetric Particle Shadows

Simon Green

July 2012

Abstract

This paper describes an easy to implement, high performance method for adding volumetric shadowing to particle systems. It only requires a single 2D shadow texture and the ability to sort the particles along a given axis.

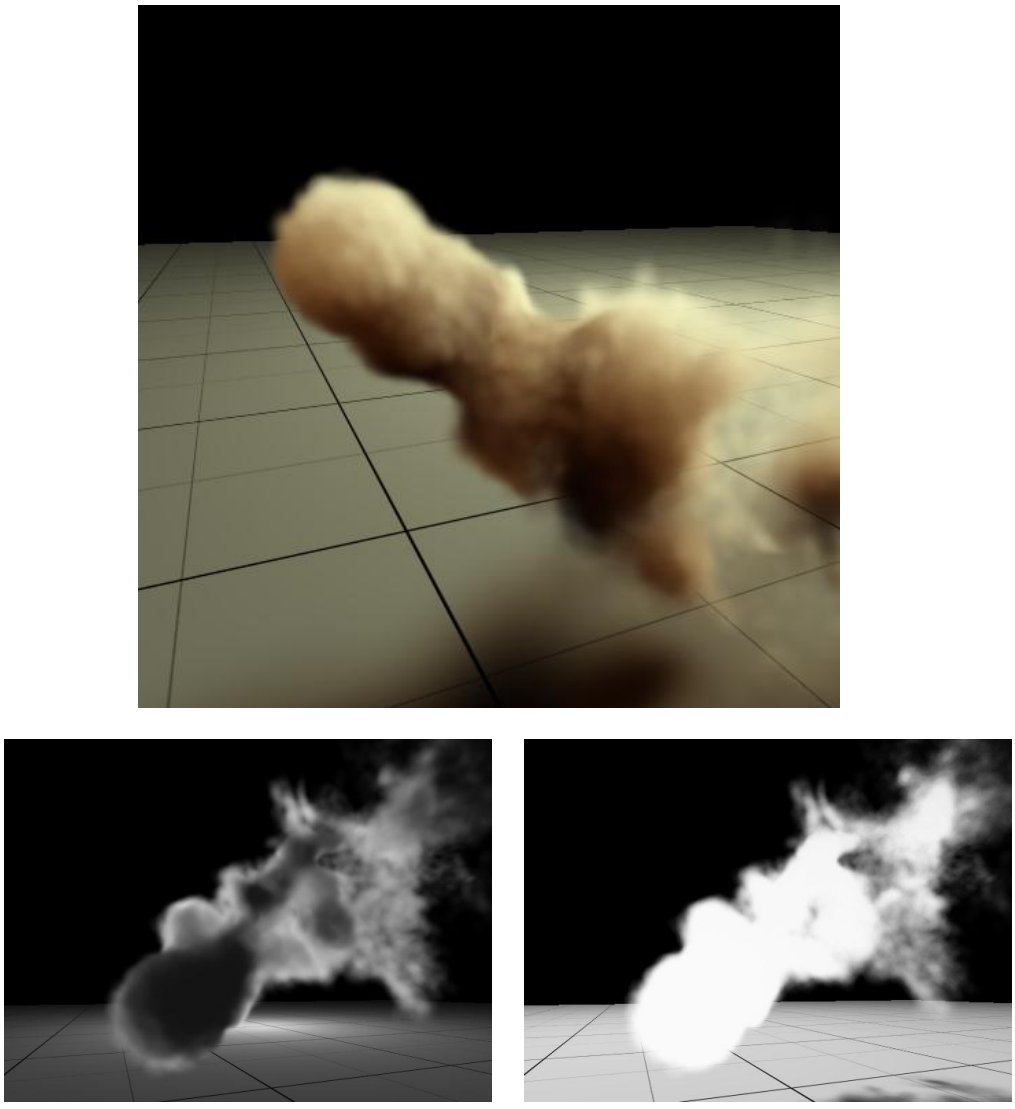

Figure 1. Particle system with volumetric shadows (a, b), and without (c)

Motivation

Particle systems are an important tool for visual effects in games and film. They are used for a wide variety of effects including smoke, explosions and fire. With the introduction of GPU-accelerated particle systems such as those provided by the NVIDIA PhysX library, both the quality of particle simulations and the quantity of particles able to be simulated have increased enormously.

Particle systems used in today's games typically use a relatively small number of large particles with simple shading. The particles are usually rendered as point sprites with an artist-painted texture (or sequence of texture images), sometimes with the addition of a normal map to get some integration with the scene lighting.

As games transition from these simple particle systems consisting of just a few hundred scripted particles to more complex physically-simulated particle systems with tens of thousands of particles (more similar to the particle systems used in offline rendering), shading becomes much more important.

Probably the most important aspect of shading dense particle systems is self-shadowing. Shadows give important cues to the density and shape of a cloud of particles. In production rendering, volumetric shadowing of this kind is usually achieved by the use of volume rendering techniques or "deep" shadow maps [1]. There have been several published attempts to implement deep shadow maps on graphics hardware, but none are yet practical for use in real-time games.

Volume Rendering Approaches

Before describing the half-angle shadowing technique advocated by this paper we will briefly discuss some alternative methods.

The most obvious way to achieve shadowing for a particle system is to use volume rendering techniques. This requires converting the particles into a discrete volume representation by rasterizing them into a 3D volume (this is sometimes know as voxelization). With DirectX 10 class graphics hardware this is possible by using the geometry shader to expand the particle points into a number of quads, which can then be routed to the appropriate slices of a 3D texture using the render target array index.

Once we have a volume representation of the particles, there are several ways to render it with shadowing. The brute force method is to march through the volume from front to back along each view ray and for each sample march towards the light to calculate the shadowing. This works but is very expensive due to the total number of samples required.

A more practical method is to first render from the point of view of the light and store the lighting information to a separate 3D shadow texture. Then when we render from the point of view of the camera we can look up in the shadow volume to get the amount of light reaching each point in the volume.

These techniques can get very good results, but the voxelization step is slow and requires a lot of storage. Additionally, these techniques require the particles to be confined to the bounds of a fixed volume, whereas particles in games are usually free to travel anywhere in the scene. One solution to this problem is to use a view-aligned 3D texture (for example, in

normalized device coordinate space) that fills the screen and follows the camera around, but this uses an impractical amount of video memory for use on current hardware.

Opacity shadow maps [2] essentially approximate the 3D shadow volume using a set of 2D textures. This can give good very good results for hair [3], but still requires a relatively large amount of storage and there can be banding artifacts visible between the layers unless many layers are used (it is possible to blur the textures to reduce this problem somewhat).

Future hardware should allow the use of more complex sparse data structures such as octrees, which would avoid the massive storage requirements of fully occupied 3D volumes.

Half-Angle Slice Rendering

The technique presented in this paper is a simple extension of the volume shading technique presented by Joe Kniss in GPU Gems 1 [4].

The key idea is to calculate a vector which is half way between the view direction and light direction. We then render the volume as a series of slices perpendicular to this half-angle vector.

This allows us to render the same slices from both the camera's and the light's point of view, since they will be roughly facing towards both. This means that we can accumulate the shadowing from the light at the same time as we are blending the slices to form the final image. The main advantage of this technique is that it only requires a single 2D shadow buffer.

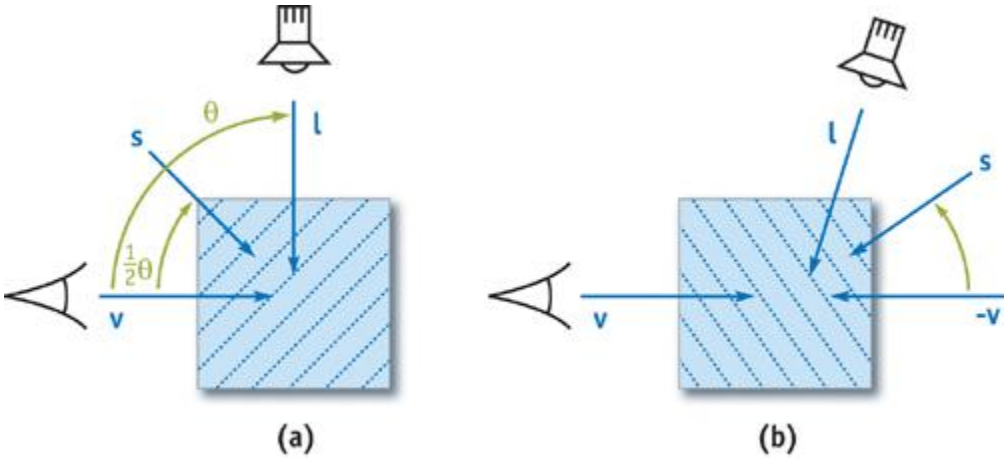

Figure 2 – calculating the half-angle vector based on the light and view directions (after [4])

The direction of rendering depends on the orientation of the light relative to the viewer. We want to always render from front-to-back from the light's point of view so that we can accumulate the shadowing correctly.

When the viewer is pointing in the approximately the same direction as the light, as shown on the left (the dot product between the two vectors is positive), we render the slices from front-to-back from the eye's point of view.

When the camera is pointing towards the light, as shown on the right (the dot product between the light and the view vectors is negative), we negate the view vector, and render the slices from back to front from the eye's point of view.

Each slice is first rendered from the eye's point of view, looking up shadowing information from the light buffer texture from the previous pass. The slice is then rendered from the light's point of view, rendering to the light buffer, to calculate the amount of light arriving at the next slice.

Details of the algorithm can be found in [4] and in the sample code that accompanies this document, available on the NVIDIA developer website.

Half-Angle Axis Sorting

In our implementation, instead of rendering polygon slices and sampling from a 3D texture as is done in the original algorithm, we simply sort the particles along the half-angle axis, and then render them in batches from front to back (or back to front, depending on the situation). For example, given 32000 particles and 32 slices, we would sort the particles along the axis and then render them in 32 batches of 1000 particles in sorted order.

We calculate the sorting order by projecting the particle position onto the sorting axis using a simple dot product. The final sorting order is still approximately correct since for a given view ray the particles will be sorted from front-to-back (or back to front) relative to the camera.

The particles are rendered as point sprites using alpha blending.

Rendering in front to back order requires using a different blending function - the "under" operator (one_minus_dest_alpha, one) rather than the normal "over" blending function (src_alpha, one_minus_src_alpha) used for back-to-front rendering [7]. This blend function requires there to be a destination alpha channel in the render target to store the accumulated opacity. It is also necessary to pre-multiply the color by alpha in the pixel shader, e.g.:

```
return float4(color.rgb*color.a, color.a);
```

Pseudo Code

```
// calculate half angle vector
If (dot(viewVector, lightVector) > 1.0) {
      halfVector = normalize(viewVector + lightVector)
      flipped = false
} else {
      halfVector = normalize(-viewVector + lightVector)
      flipped = true
```

```
}

// calculate sorting order
for(i=0; i<numParticles; i++) {
      sort_keys[i] = dot(halfVector, particle[i].pos)
      sort_index[i] = i;
}
// sort particles
sort(sort_keys);

// draw slices
clear(light_buffer)
clear(image_buffer)
batchSize = numParticles / numSlices;

for(int i=0; i<numSlices; i++) {
      drawSlice(i, image_buffer)
      drawSlice(i, light_buffer)
}

drawSlice(int i)
{
      // draw in sorted order
      drawPoints(i*batchSize, batchSize); // (start, count)
}
```

Particle Rendering

We render the particles as point sprites with a simple pixel shader that calculates a radial fall-off to make the particles appear as circular splats which are more transparent at the edges. It is also possible to add some diffuse shading based on the light direction, but typically this is not a significant effect in semi-transparent volumes like smoke.

More complex techniques such as generating individual smoke puff particles by ray marching through a noise function are possible, but typically the appearance of the kind of particle systems we are interested in comes more from the aggregate structure and shading of a large number of small particles rather than the look of the individual particles themselves.

Scene Shadowing

The final accumulated particle light buffer can also be used to perform shadowing on the scene. The light buffer texture can be accessed in the scene material shaders just like a regular 2D projective shadow map texture.

Note that if solid objects in the scene intersect the particle volume this will not give correct results, since the light buffer at this point contains the amount of light reaching the final slice only. One possible solution (assuming the objects are small) would be to sort them along the half-angle axis as well, and render them at the same time as the particle slices. Alternatively, it may be possible to use a simple linear approximation to attenuate the shadowing as the objects move closer to the light.

Performance Considerations

Since this technique uses a lot of frame buffer blending it is fill-rate bound in many cases.

However, the light buffer texture can be relatively low-resolution in most cases (e.g. 256 x 256 pixels or less) since volume shadowing effects are typically quite diffuse in nature.

Another common performance optimization is to render the final particle image at a lower resolution than the screen (typically half or quarter resolution) to an off-screen render target, and then composite it back on top of the scene using a bilinear texture lookup. This can cause artifacts where particles are occluded by the full-resolution background, but this can be fixed by using the technique of [6].

The performance of this technique is also strongly affected by the number of slices used. Although the number of slices doesn't affect the total number of particles rendered, it does affect the number of draw calls and, more importantly, render target changes. In our tests 32 to 128 slices is a good tradeoff between visual quality and performance.

The sample application renders 65 thousand particles with simple noise-based procedural animation and sorting implemented in CUDA. With a 1280 x 1024 resolution window (using 2x down-sampling) and a 256 x 256 light buffer it achieves more than 100 frames per second on a GeForce 9800 GTX+.

Particle Opacity Culling

When rendering from front-to-back, it is possible to save fill rate by culling particles that are obscured by previous particles. This can be done by checking the accumulated opacity at the previous slice using a texture lookup in the vertex shader, and culling the particle by moving it outside of the view frustum if the alpha value is over some threshold. This is the same "opacity threshold" technique used by NVIDIA's Gelato renderer [8].

Typically we can just check the opacity at a single point in the center of the point sprite. This does not give perfect results since other parts of the particle may still be visible, and can cause some popping artifacts. It is also possible to check the four corners of the point sprite and only cull the particle if they are all occluded, but this is probably not worth the additional expense.

Unfortunately this optimization cannot be used all the time with the half-angle slice technique since we are not always rendering from front to back.

GPU Simulation and Sorting

For a large number of particles the depth sorting can be come a bottleneck. For CPU implementations we recommend using a fast radix sort [5]. For systems that are simulated on the GPU it makes sense to also do the sorting on the GPU since this avoids reading back the particle positions to the host CPU. There are a number of highly optimized sorting functions that have been implemented in CUDA [12] which give good results.

GPU based simulation enables much more sophisticated procedural simulation techniques such as those based on noise functions [9], which are not feasible in real-time on the CPU.

For static particle scenes (for example, clouds), it is actually possible to pre-compute the particle orderings and select the correct order based on the view direction [10]. This avoids the need to sort at runtime completely.

Extensions

As described in [4], one extension of this technique is to blur the light buffer after each pass to simulate the scattering of light through the volume. This produces softer, more cloud like results and can also help avoid shadow aliasing when small particles are rendered. It also has the side effect of blurring the final projected shadow on the scene as well.

It is also relatively easy to modify the technique to support colored lights and different attenuation amounts for each color channel (see Figure 3). This can give good results at the expense of more storage and memory bandwidth for the extra channels.

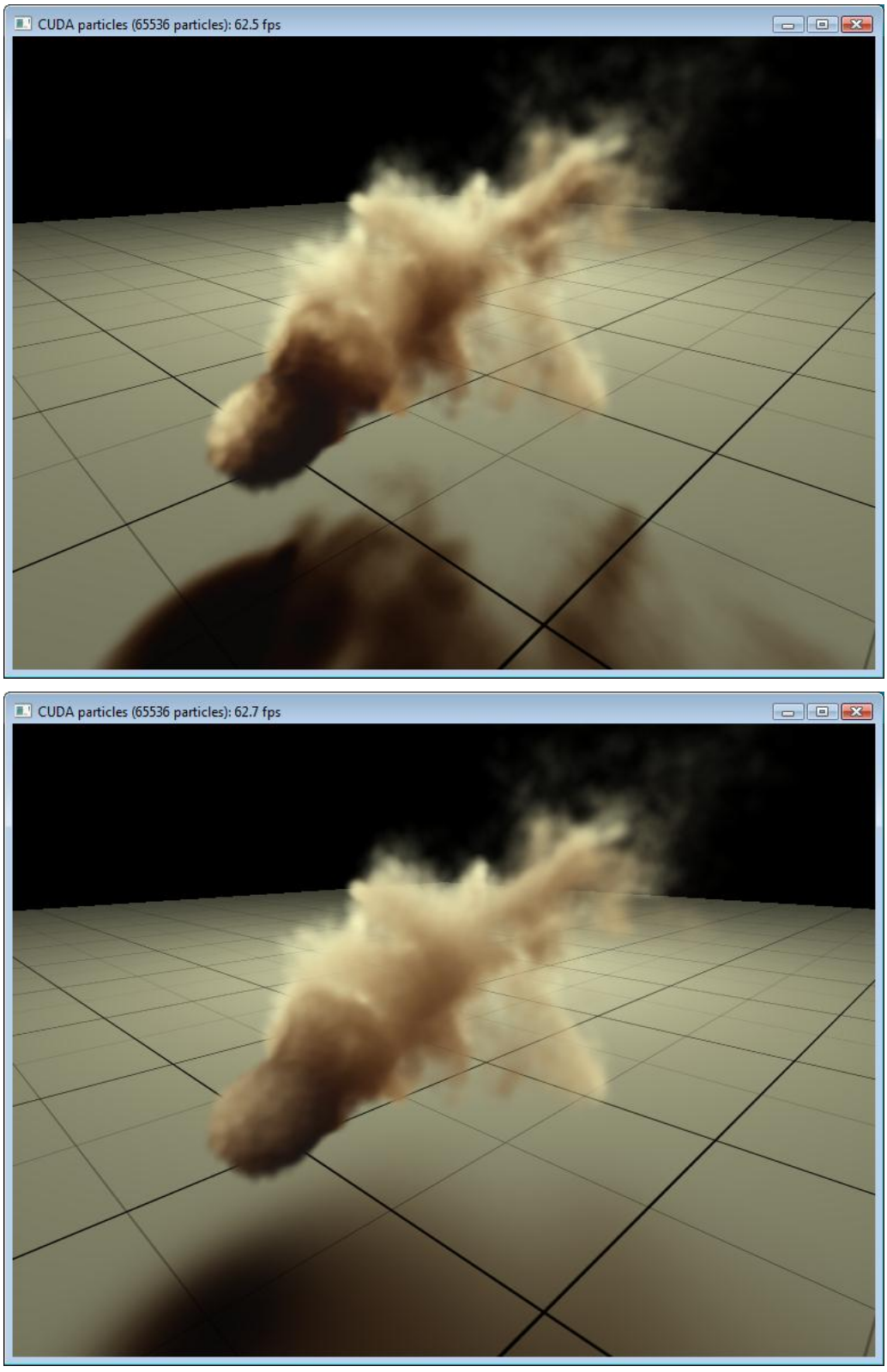

Figure 3 – Smoke without blur (a) and with blur (b) to simulate scattering

Issues

Since the particles are only sorted by their center points, there can be some popping visible when particles move in front of each other as the viewpoint changes. In practice this is not a huge problem since the particles are individually quite transparent.

Avoiding this completely would require per-pixel sorting (possibly using depth peeling or an k-buffer type approach [11] that stores all fragments for later sorting).

When the viewpoint changes slightly such that the sorting direction is reversed, there can sometimes be sudden changes in the shading visible. This is most noticeable on the slice nearest the light since this always has no shadowing. These artifacts can be reduced by increasing the number of slices.

Since the attenuation of light through the volume is exponential, it may make sense to modify the distribution of samples along the axis so that is non-linear, with more samples near the light where the majority of the attenuation occurs.

Using opacity correction [4], which compensates for the changes in final opacity caused by the varying distance between slices, can also help.

When blending a large number of particles together, the precision limitations of 8-bit frame buffers can sometimes be apparent as banding in the image. This can be avoided by rendering to an off-screen buffer with a 10-bit or 16-bit floating point format, at some cost in performance.

Sample Code

The included sample code implements a simple smoke simulation using CUDA for the simulation and depth sorting, and OpenGL for the rendering. It uses the half-angle technique to render volumetric shadows through the smoke volume. The simulation is procedural and not intended to be physically accurate - it is based on a simple noise function stored in a 3D texture.

Camera navigation is achieved using the mouse and keyboard. The left mouse button rotates the viewpoint. Dragging with the middle mouse button pressed (or the left button plus the shift key) translates the camera. Dragging with both the left and middle buttons pressed (or the left button plus the ctrl key) zooms the view.

The 'W' 'A' 'S' and 'D' keys move the camera in the appropriate direction.

Most of the other options are listed on the menu, which can be displayed by pressing the right mouse button.

Pressing the 'h' key displays the sliders, which can be used to adjust various parameters of the simulation and rendering.

Conclusion

Volumetric shadows can add a lot of realism to particle system effects. The presented technique is simple to add to an existing particle system renderer, and when combined with complex particle simulations calculated on the GPU can achieve impressive visual effects in real-time.

References

1. Deep Shadow Maps, Tom Lokovic and Eric Veach, *SIGGRAPH 2000 Proceedings* (August 2000)
2. Opacity Shadow Maps, Tae-yong Kim, Ulrich Neumann, Rendering Techniques (2001)
3. Real-time Rendering and Animation of Realistic Hair in 'Nalu'. Nguyen, H. and William Donnelly. GPU *Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (2005). 361-380
4. Volume Rendering Techniques, Milan Ikits, Joe Kniss, Aaron Lefohn, Charles Hansen. Chapter 39, section 39.5.1, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (2004).
5. Radix Sort Revisited, Pierre Terdiman (2000) http://www.codercorner.com/RadixSortRevisited.htm
6. High-Speed, Off-Screen Particles, Iain Cantlay. *GPU Gems 3* (2007)
7. Compositing Digital Images, T. Porter & T. Duff, Computer Graphics Volume 18, Number 3 July 1984 pp 253-259.
8. GPU-accelerated high-quality hidden surface removal, Daniel Wexler, Larry Gritz, Eric Enderton, Jonathan Rice, SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware (2005).
9. Curl-noise for procedural fluid flow, Robert Bridson, Jim Houriham, Marcus Nordenstam, ACM Transactions on Graphics (TOG) (2007).
10. Volumetric Sort, Iñigo Quilez (2006) http://rgba.scenesp.org/iq/computer/articles/volumesort/volumesort.htm
11. Multi-Fragment Effects on the GPU using the *k*-Buffer, L. Bavoil, S.P. Callahan, A. Lefohn, J.L.D. Comba, C.T. Silva, Symposium on Interactive 3D Graphics, Proceedings of the 2007 symposium on Interactive 3D graphics and games, pp. 97-104, 2007
12. *Designing Efficient Sorting Algorithms for Manycore GPUs*, Nadathur Satish, Mark Harris and Michael Garland. To appear in the proceedings of IEEE International Parallel & Distributed Processing Symposium 2009.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com