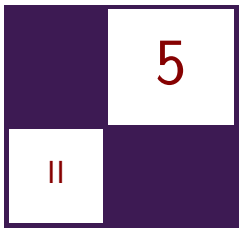

Sparse Procedural Volume Rendering

Doug McNabb

5.1 Introduction

The capabilities and visual quality of real-time rendered volumetric effects disproportionately lag those of film. Many other real-time rendering categories have seen recent dramatic improvements. Lighting, shadowing, and postprocessing have come a long way in just the past few years. Now, volumetric rendering is ripe for a transformation. We now have enough compute to build practical implementations that approximate film-style effects in real time. This chapter presents one such approach.

5.2 Overview of Current Techniques

There are many different kinds of volumetric effects, and games render them with several different techniques. We cover a few of them here.

Many games render volumetric effects with 2D billboard sprites. Sprites can produce a wide range of effects, from smoke and fire to water splashes, froth, and foam. They have been around for years, and talented artists are constantly getting better at using them. But, the sprite techniques have limits and are beginning to show their age. The growing excitement for virtual reality's stereoscopic rendering is particularly difficult because the billboard trick is more apparent when viewed in stereo, challenging the illusion. We need a better approximation. The techniques presented here help improve the illusion. (See the example in Figure 5.1.)

There have been several recent advancements in rendering light scattering in homogeneous media, enabling effects like skies, uniform smoke, and fog. These techniques leverage the volume's uniformity to simplify the light-scattering approximations. They're now fast enough to approximate multiple scattering in

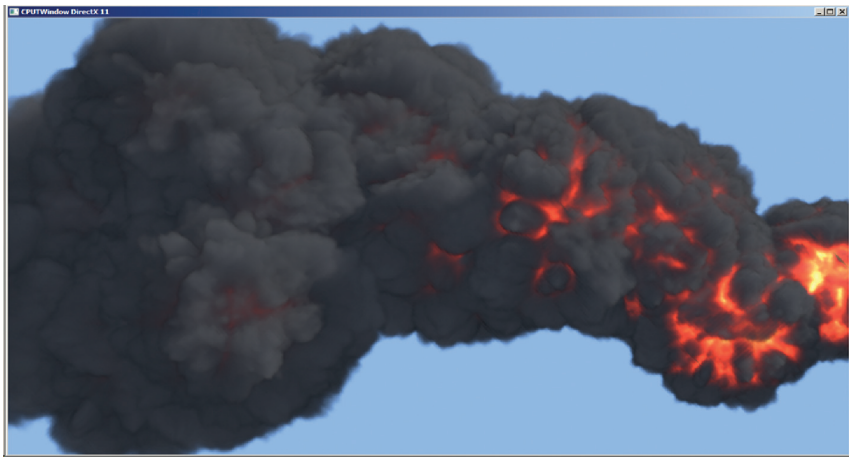

Figure 5.1. Sparse procedural volume rendering example.

real time with amazing visual quality [Yusov 14]. Light scattering in heterogeneous participating media is the more-general problem, and correspondingly is more expensive. Our technique approximates single scattering in heterogeneous media and can look very good. It is worth noting that our scattering model is simpler than the typical homogeneous counterparts to accommodate the added complexity from heterogeneous media.

Fluid simulation is another mechanism for generating volumetric effects. The results are often stunning, particularly where accuracy and realism are required. But, the costs can be high in both performance and memory. Developers typically use these simulations to fill a volume with "stuff" (e.g., smoke, fire, water, etc.), and then render that volume by marching rays originating from the eye's point of view. They periodically (e.g., every frame) update a 3D voxel array of properties. Each voxel has properties like pressure, mass, velocity, color, temperature, etc. Our technique fills the volume differently, avoiding most of the traditional simulation's computation and memory costs. We can use less memory than typical fluid simulations by directly populating the volume from a small set of data. We can further reduce the memory requirements by filling the volume on demand, processing only the parts of the volume that are covered by volume primitives. This volume-primitive approach is also attractive to some artists as it gives good control over sculpting the final effect.

5.3 Overview

Our goal for rendering the volume is to approximate efficiently how much light propagates through the volume and reaches the eye. We perform a three-step

process to produce our results:

1. Fill a volume with procedural volume primitives.
2. Propagate lighting through the volume.
3. Ray-march the volume from the eye's point of view.

Before we compute how much light propagates through a volume, we need to know the volume's contents; we need to fill the volume with interesting stuff. Volume primitives are an inexpensive, expressive option for describing a volume's contents [Wrennige and Zafar 11]. Different volume primitive types are characterized by their different mechanisms for describing and controlling the contents. Procedural volume primitives describe the contents with algorithms controlled by a set of parameters (e.g., size, position, radius, etc.). We can populate a volume with multiple primitives, sculpting more-complex results. There are many possible volume primitive types. Our system implements a single "displaced sphere" procedural volume primitive. We often refer to them interchangeably as *particles* and *displaced spheres*.

Rendering a single volume primitive is interesting, but a system that can render many and varied overlapping volume primitives is much more useful. We need to render many volume primitives within a unified volume; they need to correctly shadow and occlude each other. Supporting translucent volume primitives is particularly useful. We satisfy these requirements by decoupling the volume "filling" step from the light propagation step. We fill a metavoxel with all relevant volume primitives before we propagate lighting by ray-marching through the volume. We simplify light propagation by supporting only a single directional light (e.g., the sun), allowing us to orient the volume to align with the light's direction. This enables us to traverse trivially the volume one voxel at a time along the light's direction. Each light propagation step illuminates the current voxel with the current light intensity. At each step, the intensity is attenuated to account for absorption and scattering. Note that we propagate only the light intensity. This process can be extended (at additional cost) to accommodate colored light by independently propagating each of the red, green, and blue wavelength intensities.

We capture the volume's lit color and density (or opacity) at each voxel. Note that our model doesn't include light scattered from the neighboring volume. This could presumably be added at additional cost in time and complexity. Our model does account for shadows cast by the rest of the scene onto the volume, and for casting the volume's shadow onto the rest of the scene.

When the eye sees the lit volume, the amount of light it sees at each voxel is reduced by any other voxels between the lit volume and the eye. This is similar to propagating light through the volume with a couple of important differences. The eye view is a perspective view, in contrast to the directional light's orthographic view. And, each voxel along the eye ray can both occlude more-distant voxels and contribute light (if the voxel is emissive, or lit by the light).

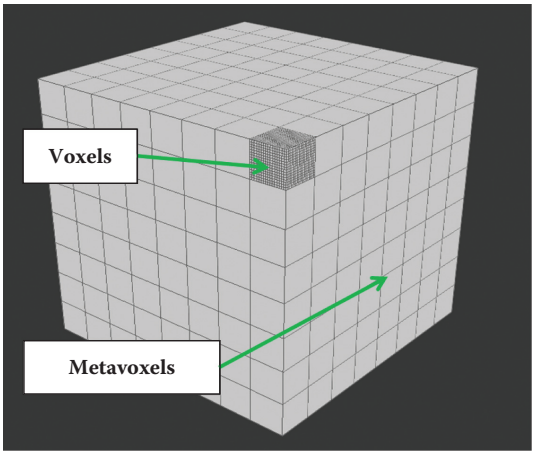

Figure 5.2. A large volume composed of metavoxels, which are composed of voxels.

The volume may also occlude the background; the amount of light from the background that reaches the eye can be absorbed and scattered by the volume. Our approach separates these two eye-view contributions. We determine the lit volume's contribution with a pixel shader and attenuate the background's contribution with alpha blending.

5.4 Metavoxels

The key point of our approach is that we can gain efficiency by avoiding unoccupied parts of the volume. Each of our tasks can be made significantly less expensive: we can fill fewer voxels, propagate light through fewer voxels, and ray-march fewer voxels. We accomplish this by logically subdividing the volume into a uniform grid of smaller volumes. Each of these smaller volumes is in turn a collection of voxels, which we call a *metavoxel*. (See Figure 5.2.)

The metavoxel enables us to efficiently fill and light the volume. Most importantly, it allows us to avoid working on empty metavoxels. It also allows processing multiple metavoxels in parallel (filling can be parallel; lighting has some dependencies). It allows us to switch back and forth between filling metavoxels and ray-marching them, choosing our working set size to balance performance against memory size and bandwidth. Using a small set improves locality. Reusing the same memory over many metavoxels can reduce the total memory required and may reduce bandwidth (depending on the hardware). It also improves ray-marching efficiency, as many rays encounter the same voxels.

Figure 5.3 shows a few variations of a simple scene and the related metavoxels. The first pane shows a few stand-in spheres, a camera, and a light. The second

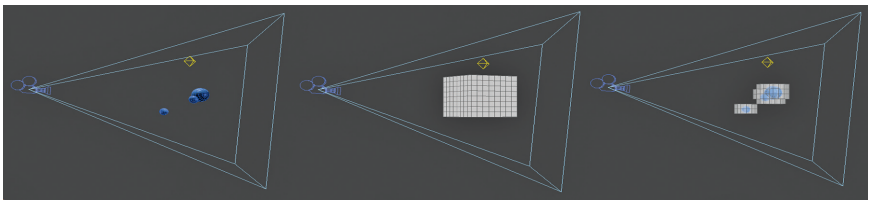

Figure 5.3. A simple scene (left), with all metavoxels (middle) and with only interesting/occupied metavoxels (right).

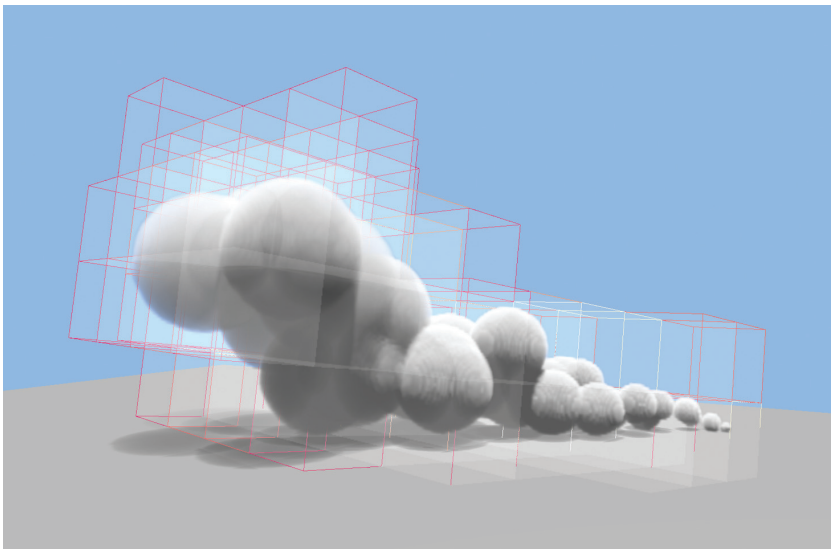

Figure 5.4. Multiple spheres and the metavoxels they cover.

pane shows a complete volume containing the spheres. The third pane shows the scene with only those metavoxels covered by one or more spheres. This simplified example shows a total volume of $512(8^3)$ metavoxels. It requires processing only 64 of them, culling 7/8 of the volume.

Figure 5.4 shows a stream of simple spheres and a visualization of the metavoxels they cover. Note how the metavoxels are tilted toward the light. Orienting the volume this way allows for independently propagating light along each voxel column. The lighting for any individual voxel depends only on the voxel above it in the column (i.e., the next voxel closer to the light) and is unrelated to voxels in neighboring columns.

Computers get more efficient every year. But memory bandwidth isn't progressing as rapidly as compute efficiency. Operating on cache-friendly metavoxels

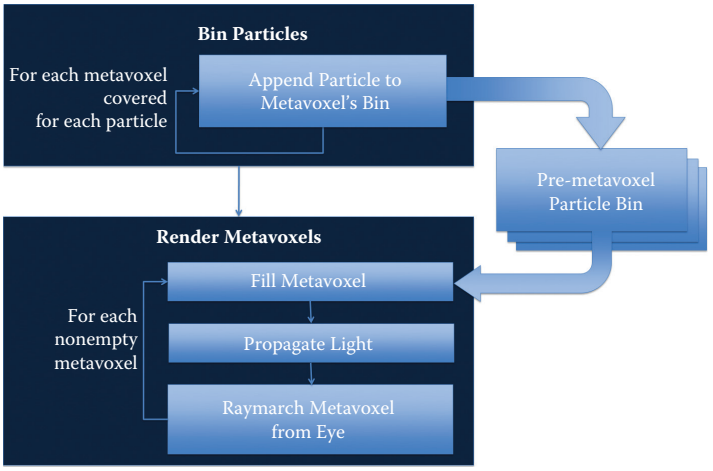

Figure 5.5. High-level algorithm.

may be more useful in the coming years as compute efficiency will almost certainly continue to outpace bandwidth efficiency. Ray-marching multiple metavoxels one at a time can be more efficient than ray-marching a larger volume. The metavoxel localizes the sample points to a relatively small volume, potentially improving cache hit rates and minimizing expensive off-chip bandwidth.

We fill a metavoxel by testing its voxels against the set of particles that cover the metavoxel. For each of the voxels covered by a particle, we compute the particle's color and density at the covered location. Limiting this test to the metavoxel's set of voxels is more efficient than filling a much larger volume; choosing a metavoxel size such that it fits in the cache(s) can reduce expensive off-chip bandwidth. Processing a single voxel multiple times, e.g., once for each particle, can also be more efficient if the voxel's intermediate values are in the cache. Populating the metavoxel with one particle type at a time allows us to maintain separate shaders, which each process different particle types. Note that we currently populate the volume with only a single particle type (displaced sphere). But, composing an effect from multiple particle types is a desirable feature and may be simplified through sharing intermediate results versus a system that requires that a single shader support every particle type.

5.5 Algorithm

Our goal is to render the visible, nonempty metavoxels. Figure 5.5 shows that we loop over each of these interesting metavoxels, filling them with particles (i.e., our displaced sphere volume primitive), and then ray-marching them from the

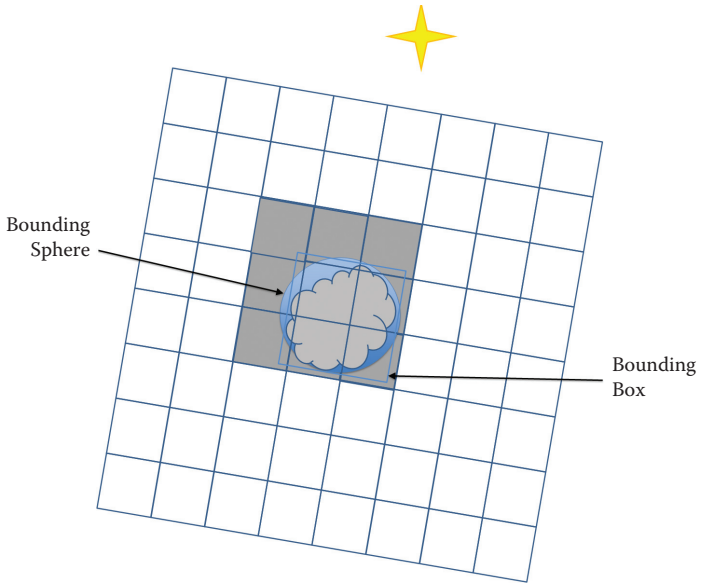

Figure 5.6. Visualization of binning rules.

eye. It's worth noting that "visible" here means visible either from the eye's view or the light's view. We consider the light's view when culling because even if a metavoxel lies outside the eye view, it may still lie between the light and the eye's view such that the metavoxels that are within the eye's view may receive its shadows. We need to propagate lighting through all parts of the volume that contribute to the final scene.

5.5.1 Binning

We determine the interesting metavoxels using a binning process. Binning adds a small amount of extra work but it reduces the overall workload. We can quickly generate a list for each metavoxel containing the indices for the particles that cover the metavoxel, and only those particles. It also allows us to completely avoid metavoxels that aren't covered by any particles.

Each bin holds a list of particle indices. We populate the bin with an index for every particle that covers the metavoxel. We maintain an array of bins—one bin for every metavoxel. (For example, were we to subdivide our total volume into $32 \times 32 \times 32$ metavoxels, then we would have a $32 \times 32 \times 32$ array of bins.) A typical sparsely populated volume will involve a small fraction of these, though the algorithm does not inherently impose a limit.

We bin a particle by looping over the metavoxels covered by the particle's bounding box. (See Figure 5.6.) We refine the approximation and improve overall

```
// Determine the particle's extents
min = particleCenter - particleRadius
max = particleCenter + particleRadius

// Loop over each metavoxel within the extents
// Append the particle to those bins for the
// metavoxels also covered by the bounding sphere
for Z in min.Z to max.Z
   for Y in min.Y to max.Y
      for X in min.X to max.X
         if particleBoundingSphere covers metavoxel[Z,Y,X]
            append particle to metavoxelBin[Z,Y,X]}
```

Listing 5.1. Binning pseudocode.

efficiency by testing each of these metavoxels against the particle's bounding sphere. If the particle's bounding sphere covers the metavoxels, then we append the particle to the metavoxel's bin.

Listing 5.1 shows simple pseudocode for binning a particle.

5.5.2 Filling Metavoxels

Our goal is to ray-march the metavoxels from the eye's point of view. Before we can do that, we need a metavoxel through which to march rays. We populate a metavoxel by testing each of its voxels against each volume-primitive particle. We say the voxel is covered by the particle if and only if the voxel is inside the volume primitive.

We reduce the number of tests by testing each metavoxel only against the particles that cover it; many more particles may participate in the system, but they may cover only other metavoxels. There are many more potential optimizations for reducing the total number of tests (e.g., progressive/hierarchical traversal). Some of these strategies can be explored within this framework, but some of them encourage fundamental changes. We look forward to future improvements.

Our task for filling the metavoxel has two goals:

1. a final density value for every voxel,
2. a final color value for every voxel.

We use a simple model for combining particle densities and colors:

$$\text{density}_{\text{final}} = \sum_{1}^{n} \text{density}_n,$$

$$\text{color}_{\text{final}} = \max(\text{color}_0 \ldots \text{color}_n).$$

The final density is given by a simple sum of the densities for every particle that covers the voxel. Color is more complex. We could blend colors together

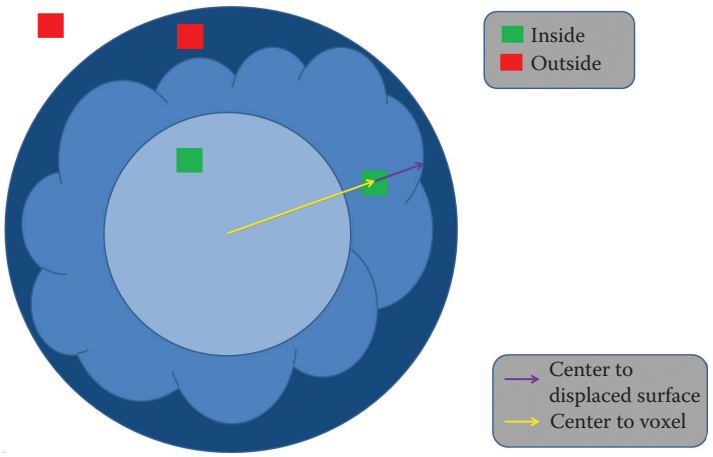

Figure 5.7. Determining coverage.

proportionally to particle density (i.e., a dense particle affects the final color more than a less-dense particle). In practice, simply accepting the maximum between two colors produces plausible results and is computationally inexpensive. This won't work for every effect, but it efficiently produces good results for some.

Different color components may be required for different effects. For example, fire is emissive with color ranging from white through yellow and orange to red, then black as the intensity drops. Smoke is often constant color and not emissive. The diffuse color is modulated by light and shadow, while the emissive color is not.

We compute the density by performing a coverage test. Figure 5.7 shows our approach. We determine the particle's density at each voxel's position. If a voxel is inside the displaced sphere, then we continue and compute the particle's color and density. Voxels outside the displaced sphere are unmodified. Note that the displacement has a limited range; there are two potentially interesting radii—inner and outer. If the voxel is inside the inner radius, then we can be sure it's inside the displaced sphere. If the voxel is outside the outer radius, then we can be sure that it's outside the displaced sphere. Coverage for voxels that lie between these two points is defined by the displacement amount.

We radially displace the sphere. The position of each point on the displaced sphere's surface is given by the length of the vector from the sphere's center to the surface. If the vector from the sphere's center to the voxel is shorter than this displacement, then the voxel is inside the sphere; otherwise it's outside.

Note a couple of optimizations. First, the dot product inexpensively computes length^2: $\mathbf{A} \cdot \mathbf{A} = \text{length}^2(\mathbf{A})$. Using distance^2 allows us to avoid the potentially expensive square-root operations. The second optimization comes from storing

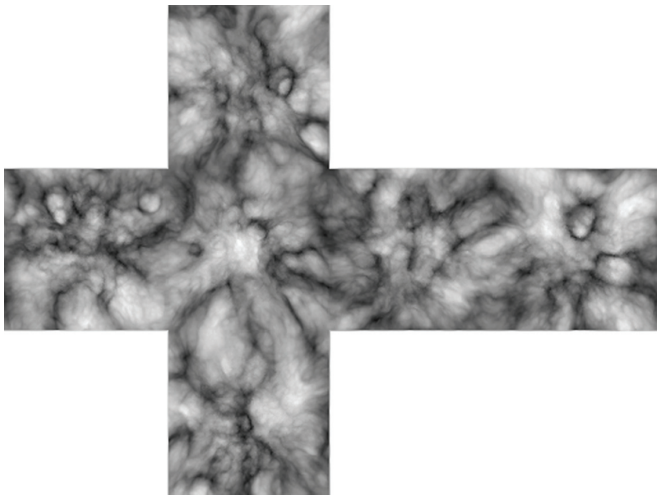

Figure 5.8. Example cube map: 3D noise sampled at sphere surface, projected to cube map faces.

our displacement values in a cube map. The cube map, like the displacement is defined over the sphere's surface. Given a voxel at position (X, Y, Z) and the sphere's center at $(0, 0, 0)$, the displacement is given by cubeMap$[X, Y, Z]$.

We don't currently support dynamically computed noise. We suspect that a dynamic solutions would benefit from using a cube map for intermediate storage as an optimization; the volume is 3D while the cube map is 2D (cube map locations are given by three coordinates, but they project to a flat, 2D surface as seen in Figure 5.8). The number of expensive dynamic-noise calculations can be reduced this way.

We determine each voxel's lit color by determining how much light reaches it and multiplying by the unlit color. We propagate the lighting through the volume to determine how much light reaches each voxel. (See Figure 5.9.)

There are many possible ways to compute the color: constant, radial gradient, polynomial, texture gradient, cube map, noise, etc. We leave this choice to the reader. We note a couple of useful approximations: Figure 5.10 shows the results of using the displacement map as an ambient occlusion approximation and using the radial distance as a color ramp (from very bright red-ish at the center to dark gray further out). The ambient occlusion approximation can help a lot to provide form to the shadowed side.

Many of the displaced sphere's properties can be animated over time: position, orientation, scale, opacity, color, etc. This is a similar paradigm to 2D billboards, only with 3D volume primitives.

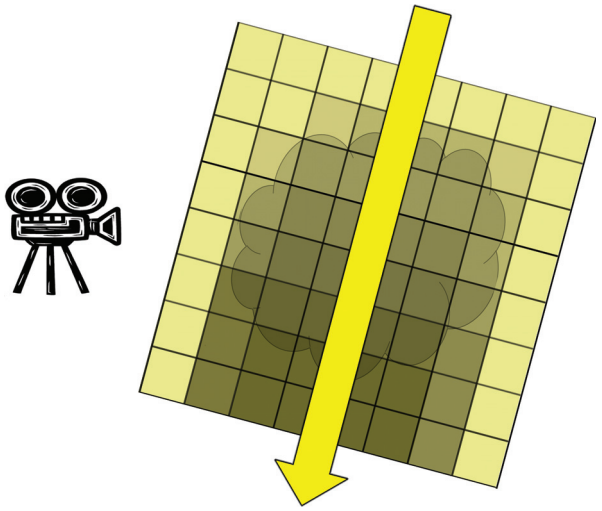

Figure 5.9. Propagating light through a metavoxel's voxels.

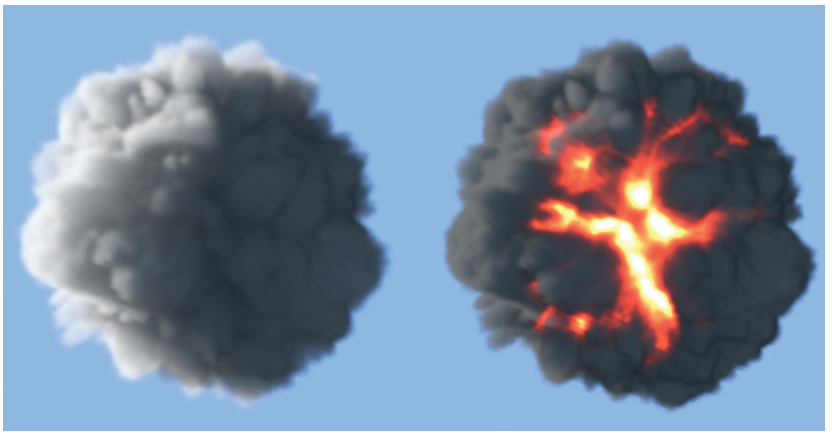

Figure 5.10. Procedural colors.

5.5.3 Light Propagation

We propagate light through the metavoxel with a simple loop. We use the rasterizer and pixel shader to perform the work. We draw one pixel for each of the metavoxel's voxel columns—i.e., a two-triangle quad covering one pixel for each of our voxel columns (e.g., for a $32 \times 32 \times 32$ metavoxel, we draw a 32×32 pixel square). Our pixel/fragment shader loops over each voxel in the corresponding voxel column.

```
// 100% light propagates to start
propagatedLight = 1

// Loop over all voxels in the column
for Z in 0 to METAVOXEL_HEIGHT
    // Light this voxel
    color[Z] *= propagatedLight

    // Attenuate the light leaving this voxel
    propagatedLight /= (1 + density[Z])
```

Listing 5.2. Light propagation pseudocode.

Listing 5.2 shows pseudocode for propagating lighting through the metavoxel. At each step, we light the current voxel and attenuate the light for subsequent voxels.

5.5.4 Eye-View Ray March

We march along the eye rays, through the metavoxel, accumulating color from the metavoxel's lit voxels and attenuating according to the voxel's density.

We implement the eye-view ray march by drawing a cube (i.e., 6 quads = 12 triangles) with the rasterizer from the eye's point of view. The pixel shader executes once for each pixel covered by the cube. Listing 5.3 gives pseudocode for the pixel shader. It loops along a ray from the eye through the pixel, sampling the

```
// The ray starts at the eye and goes through the
// near plane at the current pixel
ray = pixelPosition - eyePosition

// Compute the start and end points where the ray
// enters and exits this metavoxel
start = intersectFar( ray, metavoxel )
end = intersectNear( ray, metavoxel )

// Clamp the ray to the eye position
end = max( eyePosition, end )

// Start assuming volume is empty
// == black, and 100% transmittance
resultColor = 0
resultTransmittance = 1

// step along the ray, accumulating and attenuating
for step in start to end
     color = volume[step].rgb
     density = volume[step].a
     blendFactor = 1/(1 + density)
     resultColor = lerp( color, resultColor, blendFactor )
     resultTransmittance *= blendFactor
```

Listing 5.3. Ray march pseudocode.

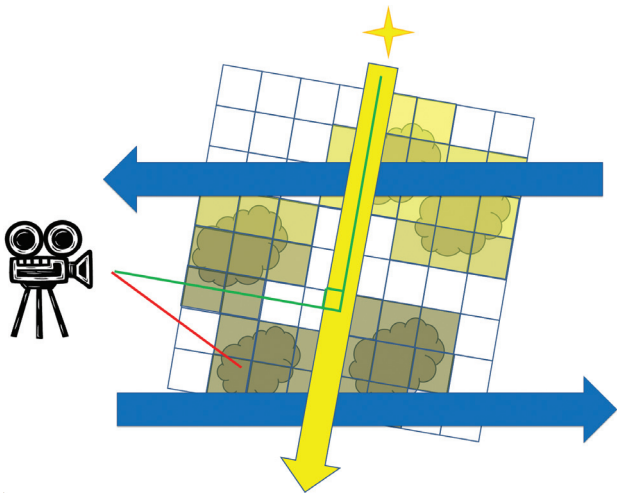

Figure 5.11. Metavoxel sort order.

volume as a 3D texture at each step. It accumulates lighting from the sample's color and attenuates it by the sample's alpha (i.e., density). The end result is a color and an alpha we can use with alpha blending to composite with our back buffer.

Note that we draw this box with front-face culling. If the eye is inside the box, then it sees only back faces. If we were to use back-face culling, then the pixels wouldn't draw and no ray marching would occur. We also don't want to draw without culling because that would potentially cause our pixels to unnecessarily draw twice.

5.5.5 Metavoxel Sort Order

Lastly, we need to render the metavoxels in the correct order for the alpha blending to be correct. We render the metavoxels one at a time, propagating light and ray-marching each one. The results blend to a shared render target. Because the metavoxels can contain semitransparent voxels, order matters.

Figure 5.11 demonstrates why we need to process our metavoxels from the top to bottom (with respect to the light) and back to front (with respect to the eye). Light propagation dictates the top-to-bottom order because an individual metavoxel's final colors depend on how much light propagates through any metavoxels nearer the light. Similarly, we need to blend each metavoxel's eye-view ray march results with those of previously rendered metavoxels.

There's a twist, however. Rendering from top to bottom and back to front can produce incorrect results for those metavoxels below the perpendicular (the green

line from the camera). The eye is looking down on those metavoxels. So, the eye can see through some previously rendered metavoxels. In this case, we need to render the more-recent metavoxel behind the previously rendered metavoxel. The solution is to process all of the metavoxels above the perpendicular before processing those below. We also switch sort order and render those metavoxels below the line sorted front to back.

The different sort orders require different alpha-blending modes. We render back to front with *over blending*. We render front to back with *under blending* [Ikits et al. 04].

It is possible to render all metavoxels sorted front to back with under blending. That requires maintaining at least one column of metavoxels. Light propagation requires processing from top to bottom. Sorting front to back can require rendering a metavoxel before those above it have been processed. In that case, we would still propagate the lighting through the entire column before ray-marching them. Consistently sorting front to back like this could potentially allow us to "early out," avoiding future work populating and ray-marching fully occluded voxels.

5.6 Conclusion

Computers are now fast enough for games to include true volumetric effects. One way is to fill a sparse volume with volume primitives and ray-march it from the eye. Efficiently processing a large volume can be achieved by breaking it into smaller metavoxels in which we process only the occupied metavoxels that contribute to the final image. Filling the metavoxels with volume primitives allows us to efficiently populate the volume with visually interesting contents. Finally, sampling the metavoxels from a pixel shader as 3D textures delivers an efficient ray-marching technique.

Bibliography

[Ikits et al. 04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. "Volume Rendering Techniques." In *GPU Gems*, edited by Randima Fernando, Chapter 39. Reading, MA: Addison-Wesley Professional, 2004.

[Wrennige and Zafar 11] Magnus Wrennige and Nafees Bin Zafar "Production Volume Rendering Fundamentals." SIGGRAPH Course, Vancouver, Canada, August 7–11, 2011.

[Yusov 14] Egor Yusov. "High Performance Outdoor Light Scattering using Epipolar Sampling" In *GPU Pro 5: Advanced Rendering Techniques*, edited by Wolfgang Engel, pp. 101–126. Boca Raton, FL: CRC Press, 2014.