Production Volume Rendering

Systems

SIGGRAPH 2011 Course Notes

Course Organizers

Magnus Wrenninge[1]
Sony Pictures Imageworks

Nafees Bin Zafar[2]
DreamWorks Animation

Presenters

Antoine Bouthors
Weta Digital

Jerry Tessendorf
Clemson University

Victor Grant
Rhythm and Hues

Andrew Clinton
SideFX Software

Ollie Harding
Double Negative

Gavin Graham
Double Negative

Updated: 10 aug 2011

1 magnus.wrenninge@gmail.com
2 nafees@nafees.net

Course Description

Computer generated volumetric elements such as clouds, fire, and whitewater, are becoming commonplace in movie production. The goal of this course is to familiarize attendees with the technology behind these effects. The presenters in this course have experience with and have authored proprietary volumetrics systems.

We present the specific tools developed at Double Negative, Weta Digital, Sony Imageworks, Rhythm & Hues, and Side Effects Software. The production system presentations will delve into development history, how the tools are used by artists, and the strengths and weaknesses of the software. Specific focus will be given to the approaches taken in tackling efficient data structures, shading architecture, volume modeling techniques, light scattering, and motion blurring.

Level of difficulty: Intermediate

Intended Audience

This course is intended for artists looking for a deeper understanding of the technology, developers interested in creating volumetrics systems, and researchers looking to understand how volume rendering is used in the visual effects industry.

Prerequisites

Some background in computer graphics, and undergraduate linear algebra.

On the web

http://magnuswrenninge.com/productionvolumerendering

About the presenters

Nafees Bin Zafar is a Senior Production Engineer in the Effects R&D group at DreamWorks Animation where he works on simulation and rendering problems. Previously he was a Senior Software Engineer at Digital Domain for nine years where he authored distributed systems, image processing, volume rendering, and fluid dynamics software. He received a BS in computer science from the College of Charleston. In 2007 he received a Scientific and Engineering Academy Award for his work on fluid simulation tools.

Magnus Wrenninge is a Senior Technical Director at Sony Pictures Imageworks. He started his career in computer graphics as an R&D engineer at Digital Domain where he worked on fluid simulation and terrain rendering software. He is the original author of Imageworks' proprietary volumetrics system Svea and the open source Field3D library, and is also involved with fluid simulation R&D. He has worked as an Effects TD on films such as *Spiderman 3*, *Alice In Wonderland* and *Green Lantern*, and is currently Effects Animation Lead on *Oz: The Great and Powerful*. He holds an M.Sc. in Media Technology from Linköping University.

Antoine Bouthors joined Weta Digital as a software developer in 2008 after earning a Ph.D on real-time clouds rendering from Université de Grenoble, France. He has been working on volumetric tools and techniques, starting with a cloud modeling and rendering system for Avatar. Some of his work in this area include support for Weta's deep image compositing pipeline, global illumination techniques, and real-time lighting.

Dr. Jerry Tessendorf is a Principal Graphics Scientist at Rhythm and Hues Studios. He has worked on fluids and volumetrics software at Arete, Cinesite Hollywood, and Rhythm and Hues. He works a lot on volume rendering, customizing simulations, and crafting volume manipulation methods based on simulations, fluid dynamics concepts, noise, procedural methods, quantum gravity concepts, and hackery. He has a Ph.D in theoretical physics from Brown University. Dr. Tessendorf received a Technical Achievement Academy Award in 2007 for his work on fluid dynamics at R&H.

Andrew Clinton is a software developer at Side Effects Software. For the past four years he has been responsible for the R&D of the Mantra renderer. He has worked on improvements to the volumetric rendering engine, a micropolygon approach to volume rendering, a physically based renderer, and a port of the renderer to the Cell processor.

Ollie Harding joined Double Negative's R&D department in 2008 after completing an MEng in Information and Computer Engineering at the University of Cambridge, UK. Having worked on various fluid visualisation and processing tools, he took over as lead developer of DNeg's volumetric renderer DNB in 2009. Ollie's software has been used on many of DNeg's recent productions, including Harry Potter and Inception, and is currently in use on John Carter of Mars and Captain America: The First Avenger.

Gavin Graham started working at Double Negative in 2000 as an effects TD, initially doing all manner of shot based effects work while also assisting R&D in battle testing in-house tools such as DNA the particle renderer, then later DNB the voxel renderer and Squirt, the fluid solver. He holds a Computer Science degree from Trinity College Dublin and an MSc in Computer Animation from Bournemouth University. He has over the last few years been CG Supervisor or FX Lead on the likes of Harry Potter 6, 2012, The Sorcerer's Apprentice and Captain America.

Presentation schedule

2:00pm	**Introduction** *(Bin Zafar)*
2:05pm	**Weta Digital** *(Bouthors)*
2:35pm	**Rhythm & Hues** *(Tessendorf/Grant)*
3:05pm	**SideFX Software** *(Clinton)*
3:35pm	Break
3:55pm	**Sony Imageworks** *(Wrenninge)*
4:25pm	**Double Negative** *(Harding/Graham)*

Production Volume Rendering at Weta Digital

Antoine Bouthors

August 10, 2011

Chapter 1

Introduction

These course notes offer an overview of the tools and techniques used at Weta Digital for volumetric authoring and rendering. Weta Digital's volumetric toolkit is a continually evolving body of works, with contribution from many people and departments (FX of course, but also Shaders, R&D, Lighting, etc.).

The Weta pipeline is centered around a few core components. In modeling, animation and lighting, Autodesk's Maya is the main tool on which we have developed a number of special-purpose plugins. In shading and rendering, we rely on Pixar's PhotoRealistic RenderMan (PRMan), here again augmented by a set of procedural plugins, shader pugins (shadeops), RIB filters, and a large library of shaders. On the FX side, we use a variety of tools depending on the task at hand, generally outputting their results in files (caches) that are fed into the regular rendering pipeline. We assume that the basics of volume rendering and lighting (covered in the first part of this course) are known by the reader.

Since PRMan has been missing native volumetric rendering support up until version 15, we have had to author our own volumetric rendering pipeline. However, rather than writing a separate, standalone volumetric renderer, the decision was made to implement volumetric rendering inside the PRMan pipeline via a set of shaders and plugins. While this method has several disadvantages in terms of practicality and ease of development, it allows us to reproduce the same look on volumetric lighting as the one developed for regular geometry. In addition, some of the development cost is offset by the fact that we do not have to re-implement a shading pipeline and the associated shaders, as we would have needed to with a standalone renderer.

In practice, we did develop a full volumetric renderer (if not several), the difference being that it is implemented as a set of PRMan shaders and plugins, as opposed to a clean, standalone application. Chapters 2–4 describe the various stages (modeling, rendering, compositing) of our volumetric pipeline while chapter 5 presents a collection of cases in which it was used.

The work described in these notes involved the hard work of many talented people from the R&D, FX and Lighting departments. In particular, Toshiya Hachisuka is to be thanked for the GPU deep shadow map implementation, Chris Edwards for the fire and explosion shaders, Jason Lazaroff for the weapons rigs, Mark Davies for the godrays and smoke trails rendering code, Florian Hu for the wavelet turbulence implementation and Peter Hillman for Weta Digital's deep image compositing pipeline and file format.

1.1 Workflow

The traditional workflow of FX shots is to run heavy, mind-boggling simulations using simulation packages before lighting and rendering. In these cases, using the best simulation tool for the job is what matters. In addition to existing commercial packages, we develop our own tools in areas where we find existing solutions missing or lacking the necessary quality or performance (see section 2.1).

However, there are numerous occasions where volumetric elements are needed but simulation is not strictly necessary. In these situations, existing commercial simulation tools are actually more of a hinderance than a help. They generally provide few modeling primitives, and the interface, oriented around simulation design, forces the user to create complex graphs or scripts just to get simple shapes out of them.

This difficulty prompted us to develop, in addition to our simulation tools, a volumetric modeler specifically designed to address these cases. Its aim is to be intuitive and simple to use, requiring no background knowledge in simulation. As a result, this modeler is accessible to and mostly used by lighting TDs as opposed to FX TDs (see section 2.2).

This allows Weta Digital to insert volumetric elements such as clouds, fog, godrays, torch beams, etc into shots without over-burdening the FX department, reducing time spent for both computers and people. In some cases, the volumetric elements may be first procedurally modeled by the lighting TD before being handed over to FX for simulation, then passed back to Lighting for rendering.

Once volumetric elements are authored, either by simulation methods or by procedural modeling, they are lit and rendered by Ligthing TDs in the same manner as the usual geometry renders, often using the very same light rig (see chapter 3).

In cases where elements recur over and over in a show (such as muzzle flashes for weapons) the FX department sets up rigs that are incorporated into the Animation rigs and driven by Animation and Lighting TDs. For example, weapons are assets that include bullets, tracers, sparks, muzzle flashes and smoke. All these elements are driven by the trigger input directed by the Animation TD. The rig also exposes lighting parameters for the Lighting TD to control the final look (e.g., color ramp, size, randomness of the flash, overall density of the smoke, etc).

Figure 1.1 summarizes this overall workflow.

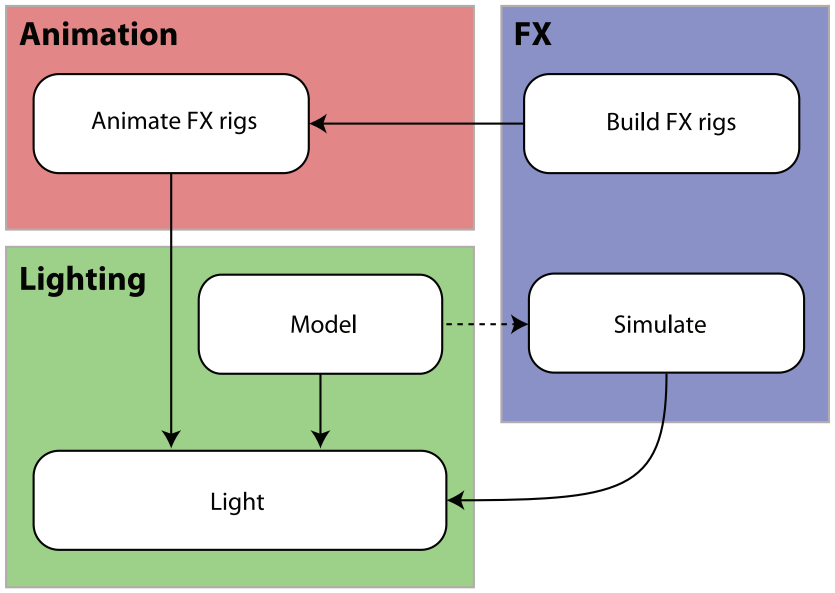


Figure 1.1: Volumetric workflow across departments (the traditional pipeline workflow is not depicted here). Simulation is traditionally handled by FX TDs, while most of the non-simulation volumetric modeling is done by lighters. Some of the FX elements are incorporated into the animation pipeline and driven by Animation and Lighting TDs.

Chapter 2

Modeling

2.1 FX and simulation

Volumes created through simulation are generally stored as files ("simulation caches") to be fed to the render job. Depending on the job, different tools will be used to perform these simulations. Describing each tool and its uses in detail falls outside the scope of this course and would be better suited to a simulation-focused session. Therefore we only outline here the general uses for these tools.

Maya's *Fluids Effects* and *nParticles* are generally the tools of choice for fire and smoke simulations, often enhanced by our implementation of the Wavelet Turbulence method [KTJG08]. For water simulations, we rely on Exotic Matter's *Naiad* software, as well as our implementation of the FFT wave generation for deep water waves [T+99]. In addition to these, we have a collection of in-house simulation packages, including *Synapse*, which lets us combine several simulations together and much more.

2.2 Procedural volumetric modeling

In addition to simulation software, we developed a volumetric modeling package designed to handle the creation of assets for which simulation is not needed. Targeted to lighting TDs, this software (dubbed *wmClouds*) is designed to be intuitive, simple to use, and requires no previous knowledge of simulation from the user. Its key claims are:

- Procedural implicit primitives are simple to use and sufficient for the vast majority of tasks ;
- The user interface should be as easy as possible to make it accessible to anyone - no graph management or script writing should be necessary ;
- The modeler should display a real-time preview of the volume with its accurate lighting, so that the user does not spend their time waiting for a render to see what they are doing.

The modeler is implemented as a Maya plugin and hooks into the rest of our RIB-generation and rendering pipeline, effectively providing a volumetric primitive in addition to the geometric primitives offered by Maya.

The modeler relies on a simple representation using spherical implicit primitives (metaballs) combined together with procedural noise functions to generate the final volume, very much as described in [Ebe97] (see figure 2.1). To keep things simple, the implicit primitives are not combined through an implicit modeling tree, as is common with complex implicit modeling, but kept in a flat hierarchy. Each primitive provides control to various parameters such as density, falloff ramp, etc.

The falloff ramp value at point $\mathbf{p}$ (in metaball space) is computed as $ramp(d)$ with

$$d = \sqrt[s]{\mathbf{p}_x^s + \mathbf{p}_y^s + \mathbf{p}_z^s}$$

with s a parameter controlling the "squareness" of the metaball and $ramp()$ a user-defined falloff ramp. When $s = 2$ (the default value), d represents the euclidean distance from $\mathbf{p}$ to the metaball center. When $s \to \infty$, d approaches the Chebyshev distance and the metaball turns into a "metasquare". Any value in between makes the primitive a cube with edges more or less rounded.

The modeler also provides controls for the noise field, with typical parameters such as noise type (fBm, absolute, etc), number of octaves, gain, lacunarity, 4^{th} dimension, and so on. In addition to the modeling parameters, wmClouds exposes lighting parameters such as extinction, scattering and emission coefficients so that all the necessary controls are grouped in one simple and unique interface (see figure 2.1).

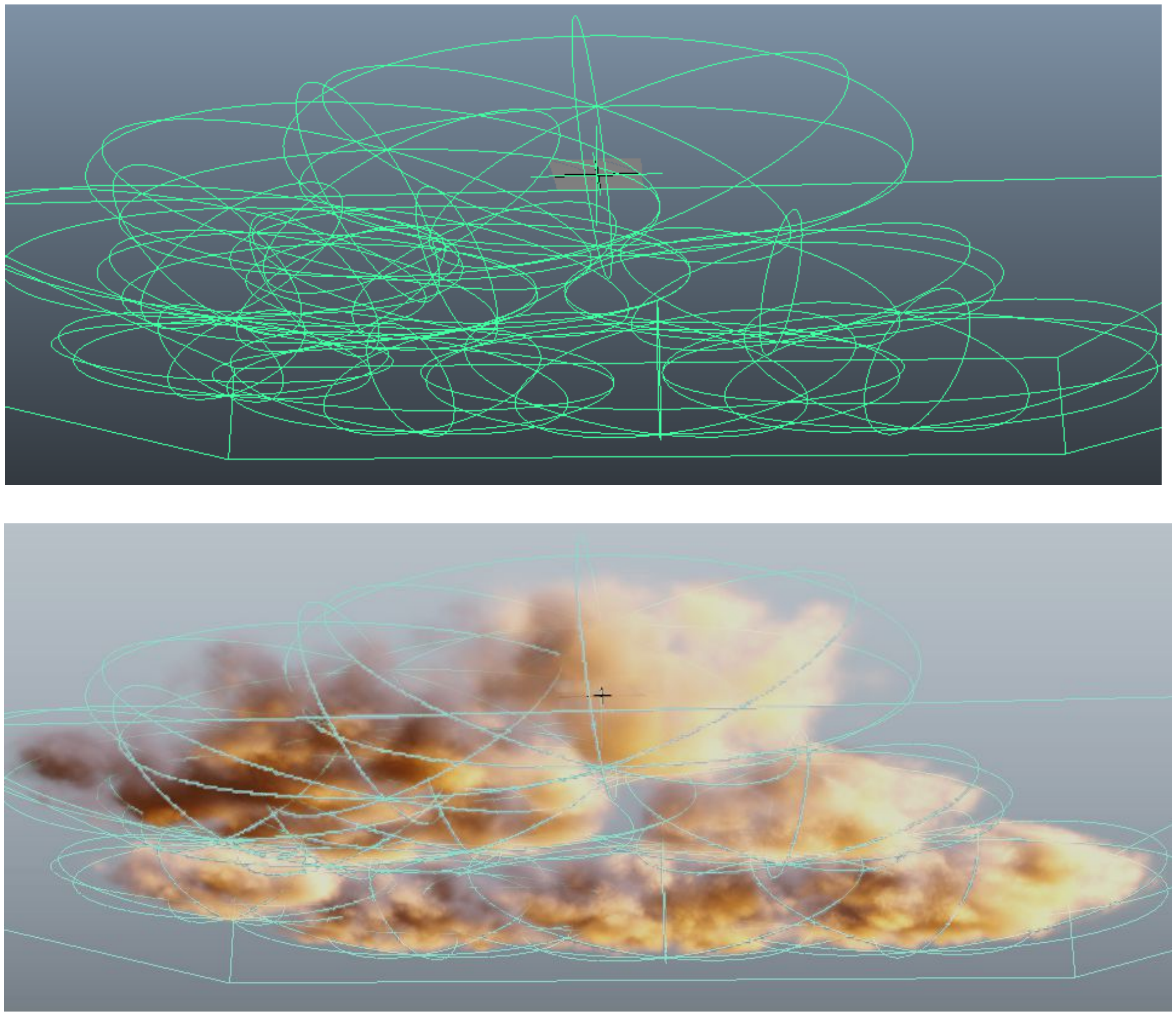


Figure 2.1: Primitives used for the modeling of a bank of clouds, without (left) and with (right) real-time lighting preview.

Chapter 3

Rendering

3.1 Real-time rendering and lighting

3.1.1 Overview

The goal of the real-time preview is to provide immediate feedback to the user without hampering the modeling process. Therefore it needs to retain interactivity. We toyed around with various ways of doing this, including progressive rendering, automatic quality adjustment and a user-defined quality knob.

We found that the user-defined solution was the most sensible one. The other options meant that the look can change in non-predictable ways while the user is adjusting parameters and modeling, which is not desirable: it is important that when a parameter is changed, the view reflects the change of that parameter and only that, so that the user gets the best idea of what they are doing. Moreover, progressive rendering does not allow the user to easily compare two options, since the view is changing all the time.

The quality knob controls various parameters of the real-time preview at once, including raymarching step size and render resolution.

Rendering is implemented in GLSL shaders. The shaders are attached to a proxy geometry (the bounding boxes of the models, much like in offline rendering - see 3.2). All the necessary data from the modeler (metaball parameters and ramps, noise parameters, scattering parameters, lighting configuration, etc) are passed on to the GLSL shaders in the form of uniform variables, arrays and textures.

First, deep shadow map generation passes are performed if necessary (the maps are not regenerated if the lighting or shape haven't changed since the previous frame), then the render pass is performed in an offscreen buffer, at the resolution specified by the quality knob. That buffer is then composited over the viewport. All lighting and rendering computations are performed in floating-point domain, and we apply in GLSL the same tone-mapping process

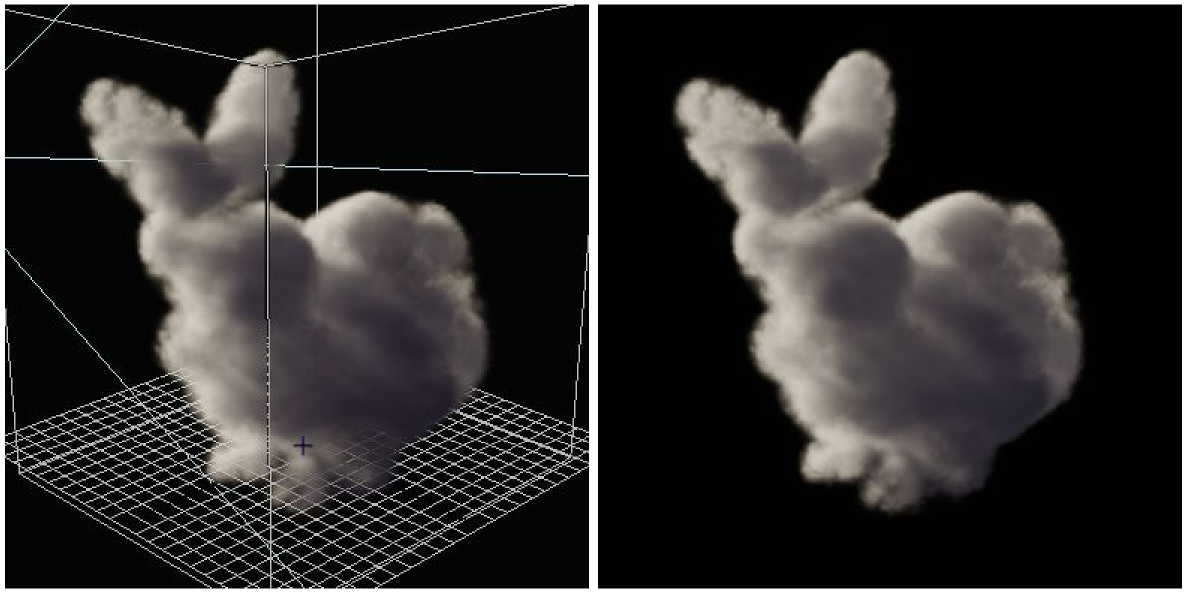


Figure 3.1: Realtime lighting preview (left) and final render (right)

than the one used in our offline pipeline.

The GLSL shaders implement a similar ray-marching algorithm as the offline rendering shaders (section 3.2): for each pixel, a ray is cast on the primitives to determine the start and end points of the volume. The shader then marches between these points, samples the volume, performs lighting computations and integrates the result. Rather than sampling the volume on the CPU and pass the result to the GPU, we only pass the procedural parameters of the model and perform ray-tracing and procedural evaluation on GPU. This way, we make the best use of GPU power and sample the volume only where the raymarcher needs it. Similarly, light shaders are executed on the GPU at the sampling locations. When enabled, our approximate multiple scattering method (see section 3.3) is also performed on GPU

The depth buffer of the Maya viewport is copied into a texture and passed on to the GLSL shaders to be used as a holdout so the preview shows how the volumes integrate with the rest of the scene. This is especially useful since models are often used to highlight other elements such as characters and thus need to be precisely modeled and placed around them. Figure 3.1 shows the result of our realtime rendering compared with the offline rendering used for production.

3.1.2 Volumetric shadow mapping on GPU

During the shadow map passes, the same ray-marching algorithm is performed, but instead of rendering an image we render a deep shadow map. The key issue here is finding a GPU-friendly format in which to write that deep shadow map that is both accurate and efficient.

Because of the dimension of the problem, a 3D texture is the natural choice for storing the deep shadow map. A 3D texture can be written into from a GLSL shader through the Multiple Render Target extension (MRT), mapping each render buffer to a slice of the 3D texture. Reading from it is only a `texture()` call away.

A simple and naive solution would be to use a uniform grid mapping between the 3D texture and the bounding box of the volume. Such a mapping guarantees constant read and write times without any indirection, making it the fastest possible solution. However, this solution wastes space on empty voxels, which is especially inefficient on sparse models. Conversely, the resolution in detailed areas is not better than anywhere else in the volume. Plus, because of the limited number of render targets, there is only so much resolution to throw at the texture to try and make it look better. A less naive approach would be to slice the volume in a non-uniform manner [HKSB06], but this is only slightly more efficient than the uniform approach.

It is possible to store a "real" deep shadow map on the GPU that is more robust against sparse volumes and closer to the CPU representation, using a linked list of deep samples for each pixel [KN01]. Unfortunately, this involves a more complex representation and results in non-constant read time involving indirections.

As a result, we devised our own representation, which lies somewhat half-way between [HKSB06] and [KN01]. For each (x, y) pixel, the z column represents a uniform sampling of the volume along the view ray. The difference here is that the domain spanned by that z column is different for each pixel, matching the portion of the volume that is not empty. We store the depths of the start and end points of this domain in the first two coordinates of the first texel. We use the rest of the z texels to store the deep shadow information between these two points with a uniform distribution as shown on figure 3.2.

This approach lets us sample the volume more densely than [KN01] in non-empty areas while keeping the read time constant and using only one indirection level. To compute the start and end point of the domain, we use the first entry and last exit point of the ray through the modeling primitives. Instead of storing RGB opacity at each sampled point in the deep shadow map, we store only one floating-point value representing what we call the "normalized optical depth" along the volume. This value represents the optical depth along the volume, without the extinction coefficient portion, for a volume with a canonical global density multiplier. We apply the extinction coefficient and actual global density multiplier at reading time. This has two advantages. First, it gives us three times more z resolution by storing 4 deep samples in each texel. Second, we do not have to recompute the deep shadow map if the extinction coefficient or the global density change. The constraint is that the extinction coefficient must be constant throughout the model.

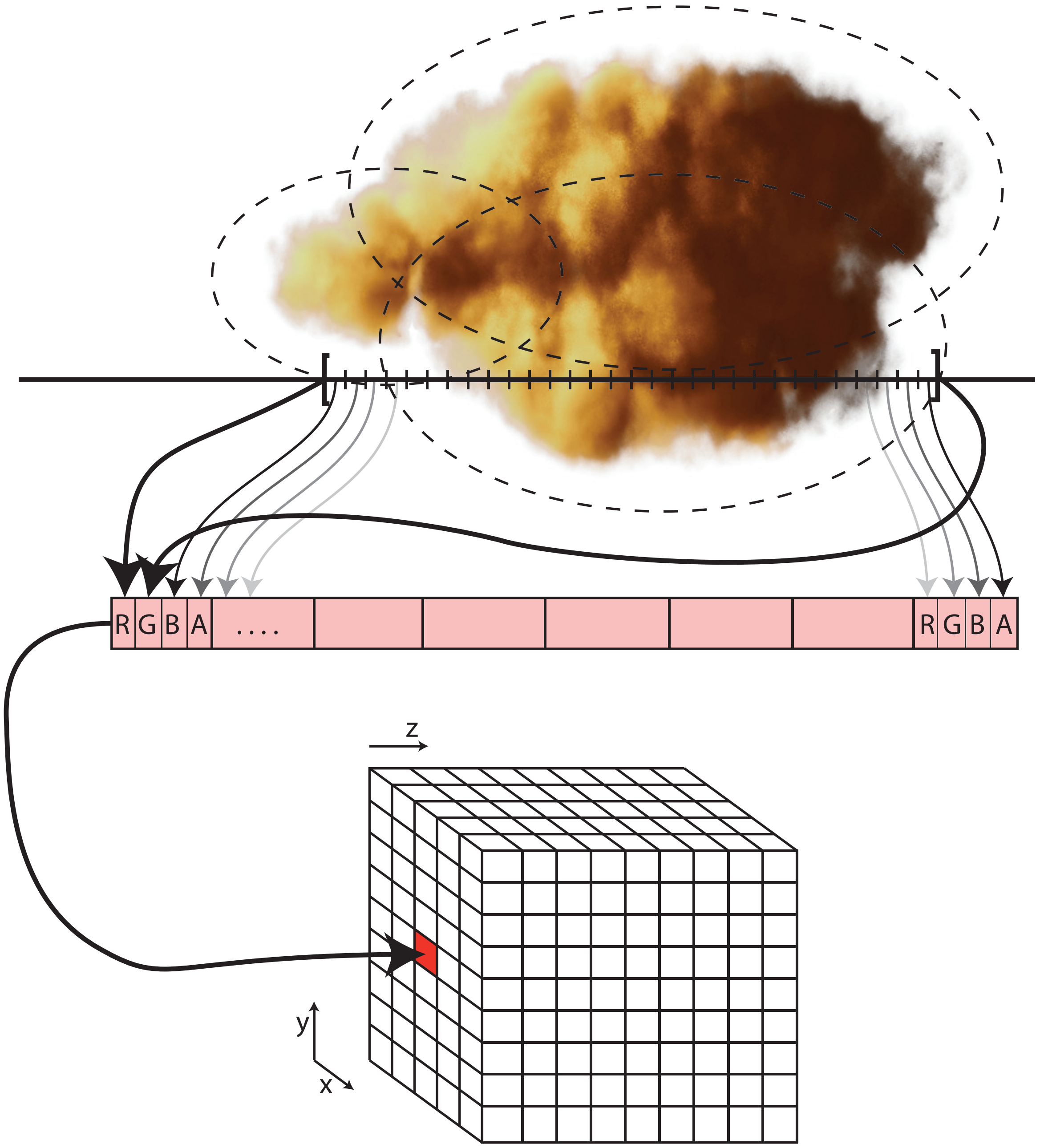


Figure 3.2: Volumetric shadow mapping on GPU. For each (x, y) coordinate in the texture, the z column stores uniformly-spaced deep samples. The first two floats in the first texel store the depth of the first and last sample. No marching is done outside the metaball boundaries.

3.2 Offline rendering

Offline rendering is performed roughly in the same way as our real-time rendering approach (see 3.1), the main difference being that it is implemented in a PRMan pipeline. GLSL shaders are replaced by Renderman Shading Language (RSL) shaders. Instead of using GPU 3D textures, we use PRMan's native deep shadow map format [LV00]. The RSL shaders are PRMan atmosphere shaders bound to a proxy geometry in the same manner as the GLSL shaders.

For rendering simulation caches, we wrote a PRMan shadeop that loads the volume data and provides the necessary ray tracing and sampling information to the RSL shader. The shader performs a 3D digital differential analyzer (DDA) marching algorithm using the fluid cache's resolution as input to ensure the ideal marching pattern.

When rendering volumes coming from our procedural modeling tool, the procedural model parameters are serialized on disk from the modeler and loaded by a PRMan shadeop that provides the tracing and sampling operations to the raymarching shader. As in the GLSL case, the shader first determines the marching domain by tracing a ray against the primitives, then marches along that domain.

Since PRMan does not know about the volume primitive when using atmosphere shading, its native deep opacity output only corresponds to the proxy geometry, which is not the result we want. In addition, we want to output deep color as well as deep opacity, which PRMan's deep driver would not support until version 16. To overcome these limitations, we implemented our own volumetric deep data output as a shadeop, using the PRMan *DTex* API. We output both deep color and deep opacity, which are then merged into a single deep RGBA ODZ file (see section 4.1).

At each raymarching step, we perform the lighting computation by querying the light shaders and applying the single scattering equation. Depending on the object being rendered, we may apply special-purpose shading operations, such as blackbody radiation [NFJ02] to determine the emission coefficient of fire and explosions, or multiple scattering for dense, high-albedo clouds. The next section describes our multiple scattering methods.

3.3 Global illumination

Overview Multiple scattering in volumes is a notoriously computationally expensive task and is generally achieved via various approximation techniques such as the diffusion approximation [Sta95] or the multiple forward scattering approximation [PAS03]. We use a selection of various algorithms to achieve the appearance of multiple scattering, depending on the type of participating media being rendered and the desired look.

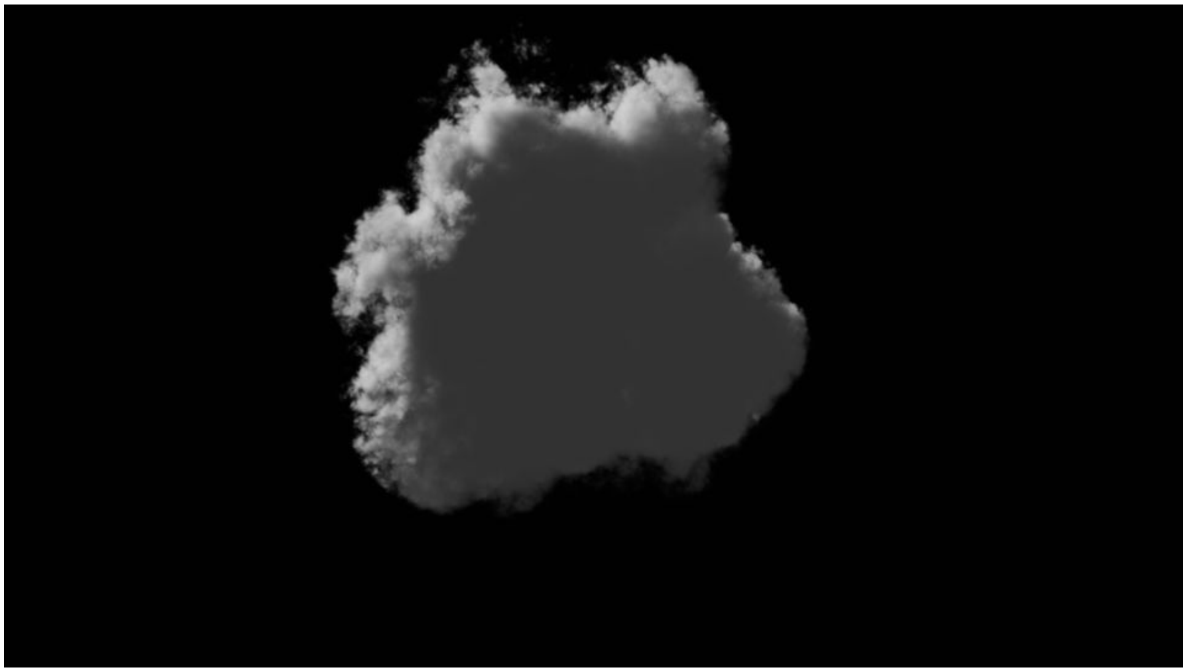
Fast approximate multiple scattering One of the methods we use to approximate multiple scattering is a simplified, production-robust adaptation of the one described in [BNM+08, Bou08]. The core idea behind it is that each order of scattering can be computed separately or in discrete groups rather than to be considered as either single scattering or multiple scattering. Moreover the multiple scattering studies done in [Bou08] show that higher orders of scattering display a behavior similar to that of single scattering.

Armed with these two premises, we can easily compute the higher orders of scattering by simply applying the single scattering equation with different parameters. Specifically, the mean free path should be lengthened to account for longer scattering paths, and the phase function should be made more isotropic. When using a deep shadow map technique, we can also add blurring of the deep shadow map depending on the scattering order to account for the spatial spreading of light.

By using this simple approach, one can go a long way towards plausible multiple scattering with very little computation, as shown on figure 3.3. Moreover, this technique is directly applicable to the GPU, which lets us use it in our real-time lighting preview (see section 3.1) at little cost. As an example, Figure 3.1 uses this technique in both the real-time version and the offline version.

Note that this technique falls somewhat in the the "multiple forward scattering approximation" category [PAS03] in that its implementation shares a lot of similarities. However, it does not get there with the assumption that multiple scattering is mostly forward for anisotropic media, which breaks for high-albedo media as shown in [Bou08]. Instead, this approach applies to media that are isotropic as well as anisotropic, and high-albedo as well as low-albedo. All it requires is for single scattering parameters mimicking the higher-order scattering parameter to be chosen so that the approximation matches the reference best. These parameters can be found through experimentation and curve-fitting as done in [Bou08, DLR+09], or even controlled independently or procedurally if necessary for ultimate control.

Volume color bleeding Since version 15, PRMan has provided functionality to compute volumetric multiple scattering through their `ptfilter` utility. The technique is a point-based global illumination technique similar to that used for surfaces [Chr08]. The main advantage of this approach is that it works with any kind of illumination such as point-cloud based illumination, whereas our multiple scattering approximation works best with deep shadow map-based illumination. The disadvantages are that it requires additional pipeline steps (baking the irradiance in a point cloud, running ptfilter, then reading the filtered result in the beauty render), it is much more computationally expensive, and it is not trivial to port to the GPU.



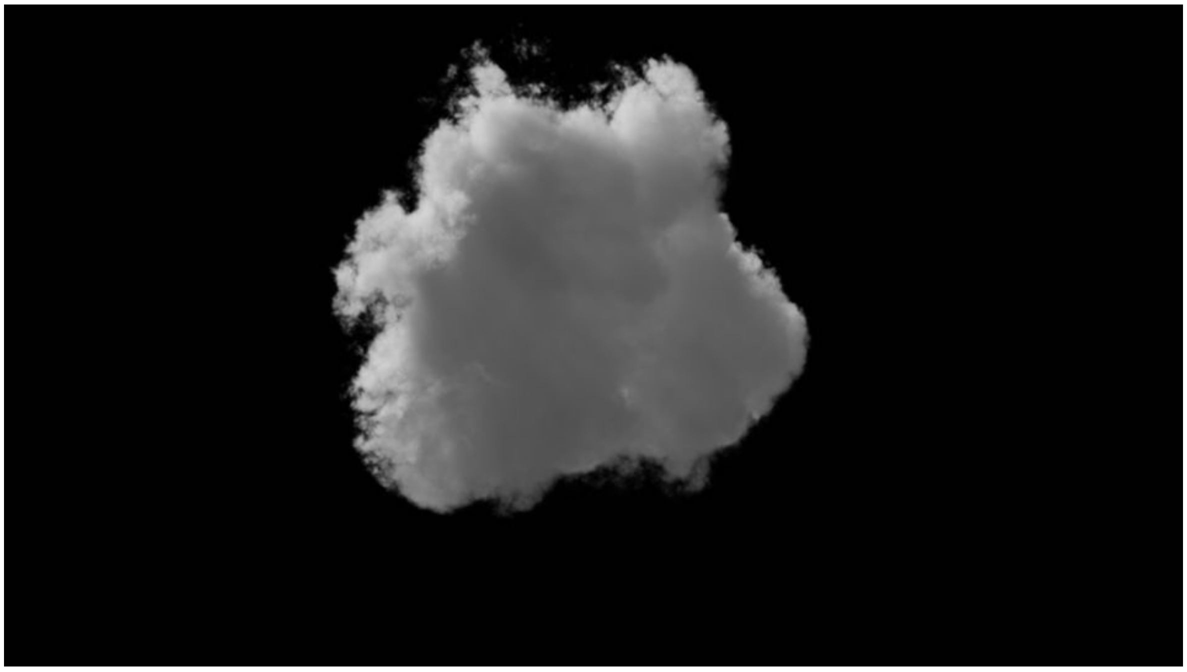


Figure 3.3: A cloud rendered with single scattering only (top) and with all orders of scattering (bottom) using our multiple scattering approximation technique. Note the soft translucency appearing in the core area, and the softening of the lighting on the edges. Render times for both images are similar.

Chapter 4

Compositing

4.1 Deep image format

In addition to Pixar's deep shadow map [LV00] format (DSHD), we also have implemented our own deep image file format, dubbed *ODZ*. This format is built on the OpenEXR file format, and supports features such as arbitrary channels, multiple views, better compression, volumetric samples, etc. This allows us to store stereo deep RGBA images in one single file, which would otherwise take 4 separate, bigger DSHD files.

This format is used widely in the pipeline. To ease the transition with tools that only support DSHD files, we wrote conversion utilities to easily convert files between the two formats.

4.2 Holdouts

Since we are making heavy use of deep image compositing, all our geometric renders output deep opacity images by default. Holding out a geometry pass from a volume pass is therefore as simple as picking the deep opacity sequence from the appropriate pass and using it as a holdout.

Dealing with holdout is done through a shadeop that can read either DSHD or ODZ files and handles all the holdout details for the shader. Because PRMan's traditional strategy is to shade at time 0 and to apply motion blur post-shading, this would result in holdouts being streaked by the volume's motion blur, which is highly undesirable. Moreover, the motion blur applied by PRMan is that of the proxy geometry and not of the volume, which is incorrect especially in cases where the camera is inside the volume.

To overcome these issues we use PRMan's "visible point volume" shading, which runs the shaders on the pixel samples rather than on the vertices. This allows us to render with accurate volumetric motion blur and pixel-precise holdouts (see figure 4.1).

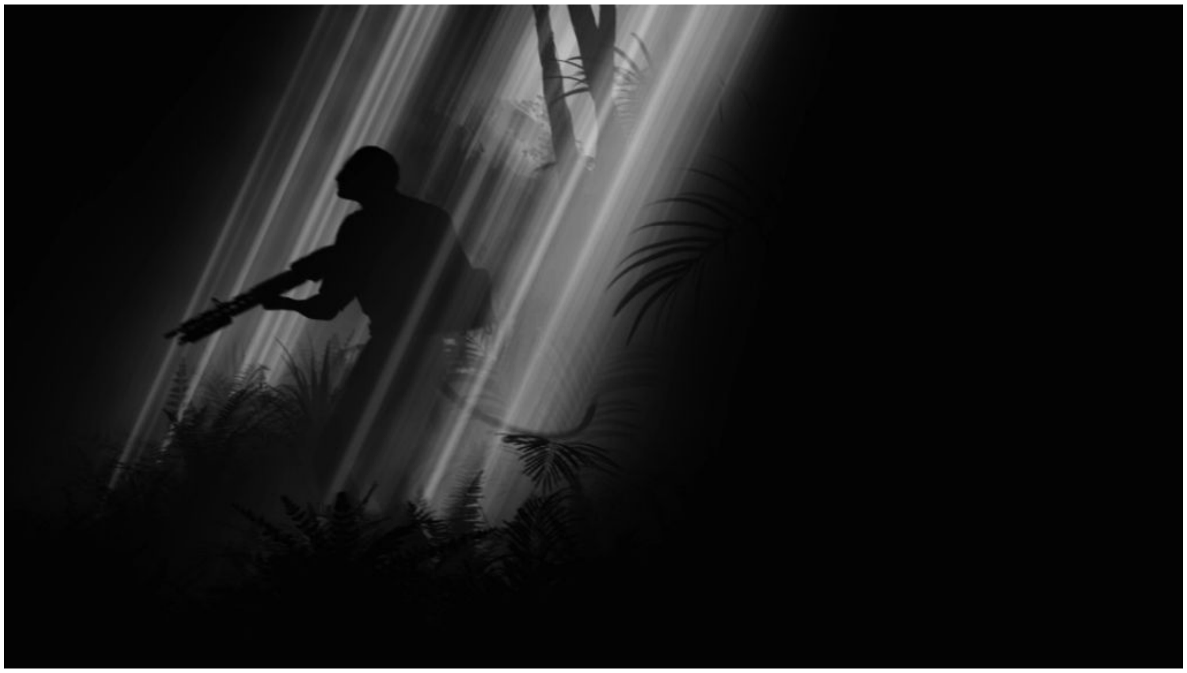


Figure 4.1: The godrays of figure 5.4, bottom, rendered with holdouts. If any bit of geometry changes, the rays have to be re-rendered.

Even though these features allow us to easily render volumes with holdouts and composite them properly with the typical compositing operations, using holdouts has two main disadvantages. First, this workflow is not the friendliest for our deep compositing pipeline, where compositors are used to bring in renders from any passe without having to bother about which pass should be held out by which other. Second and probably most important, using holdouts means that if any of the holdout passes change, the volume passes has to be re-rendered. This is especially annoying when working on complex shots where multiple passes are worked on by several TDs at the same time, and simply increase the iteration time for no good reason.

4.3 Deep image output

Because of the hinderances and limitations of working with holdouts, we implemented deep image output from volumetric renders. This is done through yet another shadeop called by the raymarching shader. It gives us the ability to output stereo deep RGBA volumetric ODZ images that are free of holdouts. These images are used in our deep compositing pipeline in virtually the exact same way as geometry renders are.

This volumetric deep output pipeline was developed during the production of Avatar and is now used on all our shows, to the point that rendering with holdouts is the exception rather than the rule.

Chapter 5

Case studies

5.1 Avatar: clouds

One of the many challenges that Avatar posed was the sheer number of CG shots, a good part of which were to take place in aerial environments such as the floating mountains of Pandora. These floating mountains were to be populated with clouds. While this task would be traditionally handled by FX TDs, these are a scarce and prized resource and had already much to do with the major effects sequences. Moreover, there was little simulation to do for a vast majority of these clouds. The job consisted mostly in modeling and lighting tasks.

This prompted us to develop the procedural modeler described in section 2.2. All the clouds in Weta Digital's shots were created with this modeler, most often by lighting TDs and sometimes by FX TDs and even compositors. The modeler was successful in getting these shots done without gobbling FX resources.

The desired look was wispy, light clouds, for which high detail was required in the shape while single scattering was sufficient for the shading. The procedural noise generated by the modeler ensured the high detail, and our volumetric deep shadow map approach was sufficient for the single scattering look.

The majority of shots used holdouts rendering. We developed our volumetric deep data output during that time and started using it towards the end of production. Figure 5.1 shows some examples of clouds-laden shots. Figure 5.1, top, was one of the first shots to use deep volumetric compositing instead of holdouts.

5.2 Avatar: fire and explosions

Fire, smoke and explosions were simulated using Maya Fluids, enhanced using our implementation of the wavelet turbulence [KTJG08] technique, and rendered as described in section 3.2.

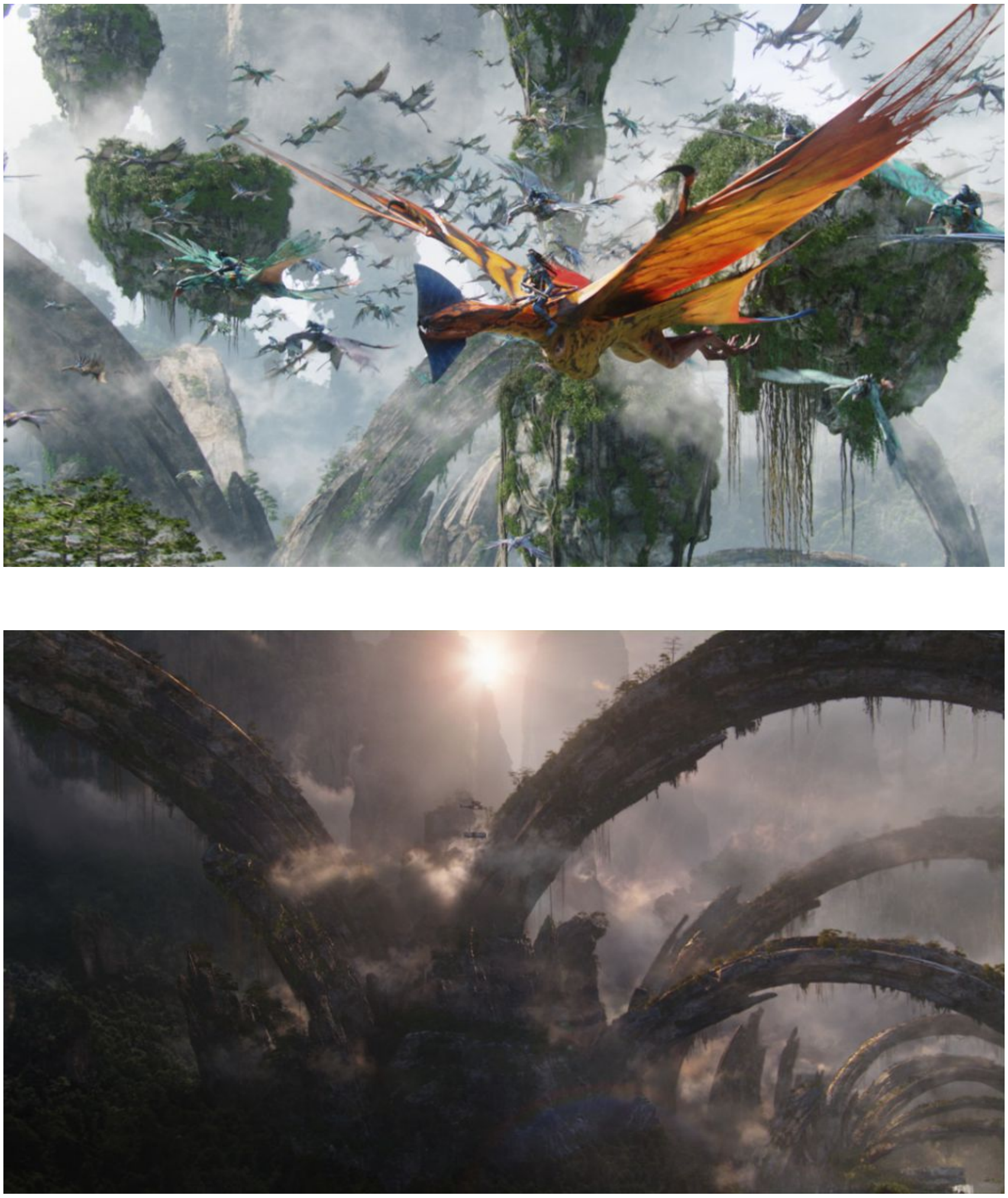
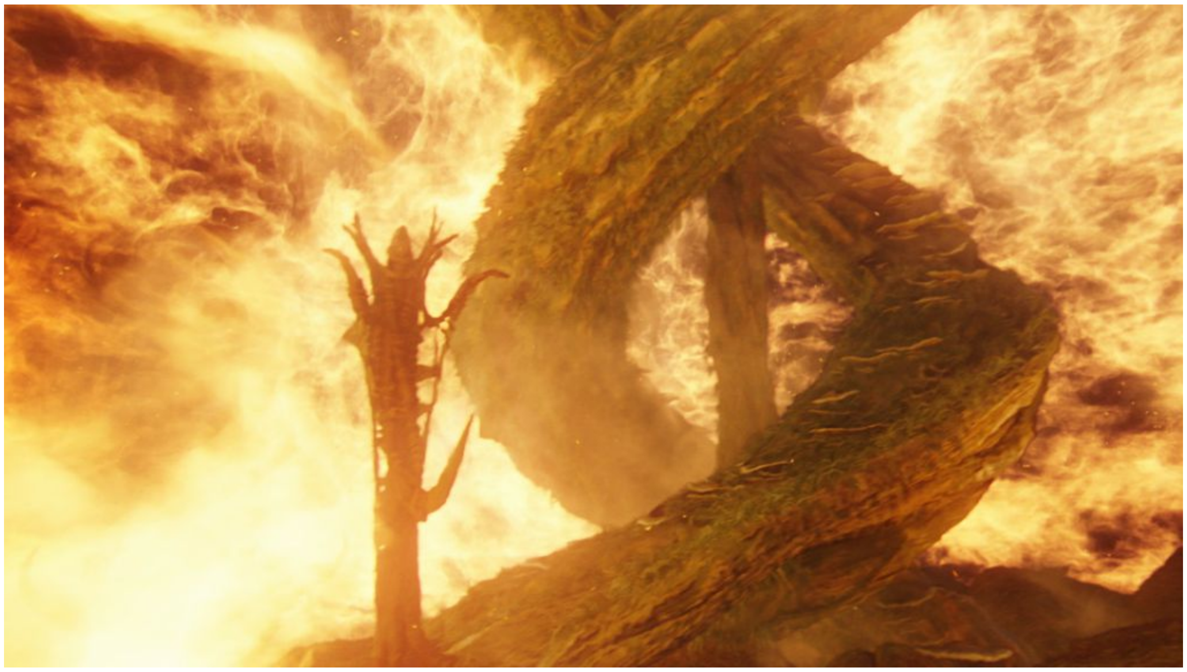


Figure 5.1: Clouds on Avatar were procedurally modeled.



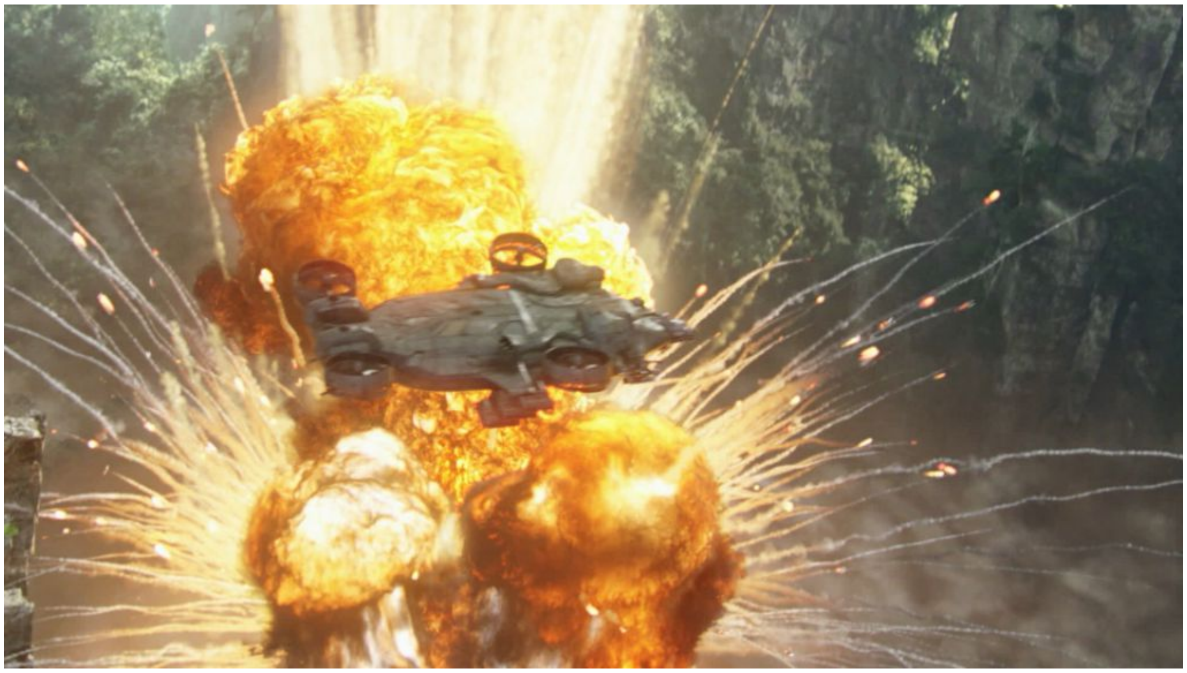


Figure 5.2: Fire and explosions on Avatar made heavy use of wavelet turbulence and blackbody radiation.

For fire and explosions we used blackbody radiation [NFJ02] rather than relying on artist-controlled color ramps, to ensure a uniform look across the show. Figure 5.2 shows a couple examples of these renders.

5.3 Avatar: muzzle flashes

As outlined in section 1.1, the muzzle flashes were volumetric elements attached to the weapon assets, for which the FX department built rigs. This spared the FX TDs the labor of manually modeling or simulating every flash for every trigger of every weapon of every battle shot.

For smaller weapons, muzzle flashes were procedurally built using our procedural modeler, controlled by the rig rather than manually by the artist (see Section 2.2). For larger weapons (such as the amp suits), the muzzle flashes were pre-simulated fluid caches dynamically loaded by the rig. Figure 5.3 shows the results generated by these rigs.

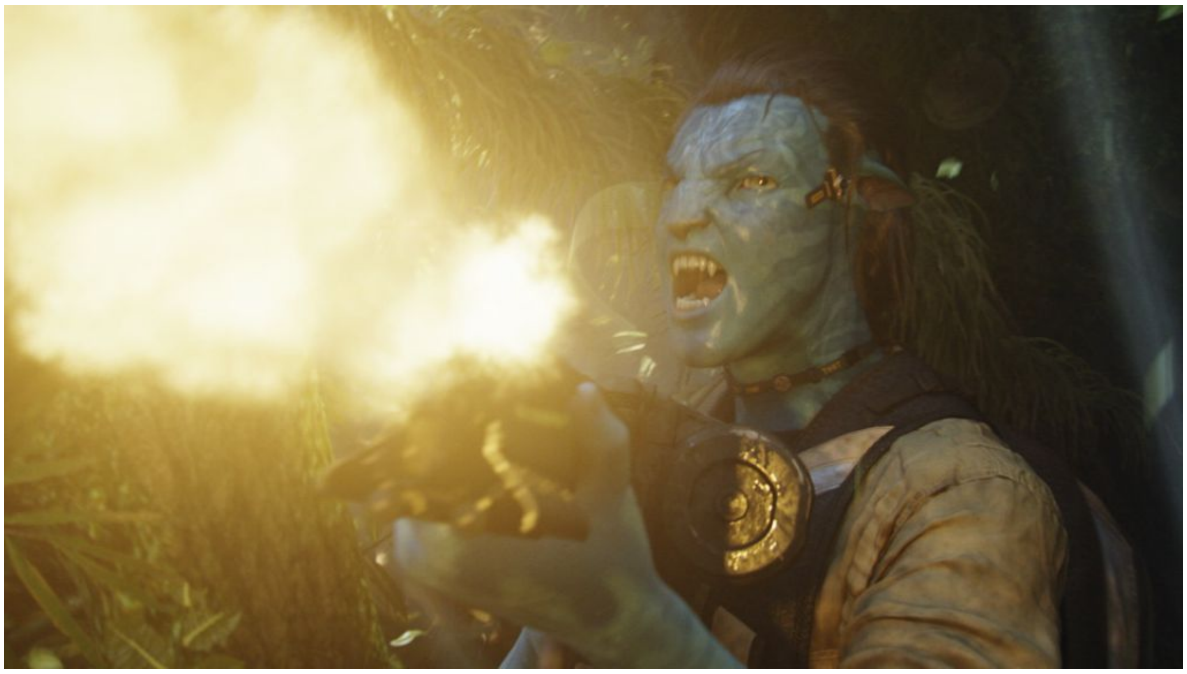
5.4 Avatar: godrays

Godrays were strongly present in all the Pandora jungle sequences on Avatar. The look of godrays and beams is very strongly directed by lighting and as a result is yet another task that is more naturally fit for lighting TDs than FX TDs, even though it involves volumetric elements. On Avatar, these rays were achieved by placing very simple procedural volumetric elements, such as a roughly constant density field modulated by noise, and proper lighting.

Lighting consisted primarily in using the shot's lights and using the environment to shadow the volume accordingly. When a more "beamy" look was needed, lighters could add gobos to the light just as they would for a standard beauty render. We also experimented with breaking the light more by adding beams procedurally directly into the volume shader. Figure 5.4 shows some of the shots using godrays. Figure 4.1 shows a godray pass alone rendered with holdouts.

5.5 The A-Team: clouds

Although Weta Digital's main work on *The A-Team* was on the "docks" sequence while Rythm & Hues was in charge of the "clouds" sequence (see [Dun10]), we were also tasked to deliver two shots of the clouds sequence early in the production to be used for the movie trailer. Since te whole sequence was handled by Rythm & Hues in the final movie, these two Weta shots can only be seen in the trailer. Despite the small coverage of Weta on this clouds sequence, we thought it would be interesting for the reader to see how two visual effects companies handled the very same shots on the very same movie. For more detail on Rythm & Hues' approach see the *Resolution Independent Volumes* section in this course, as well as [HIOT10].



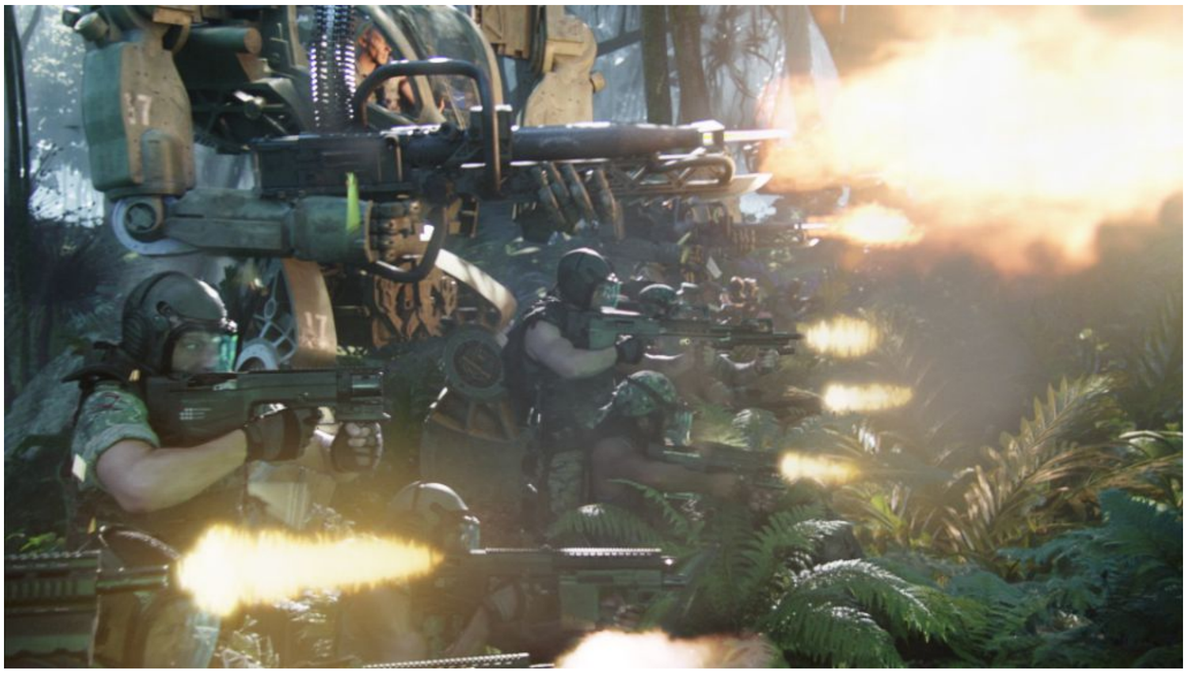
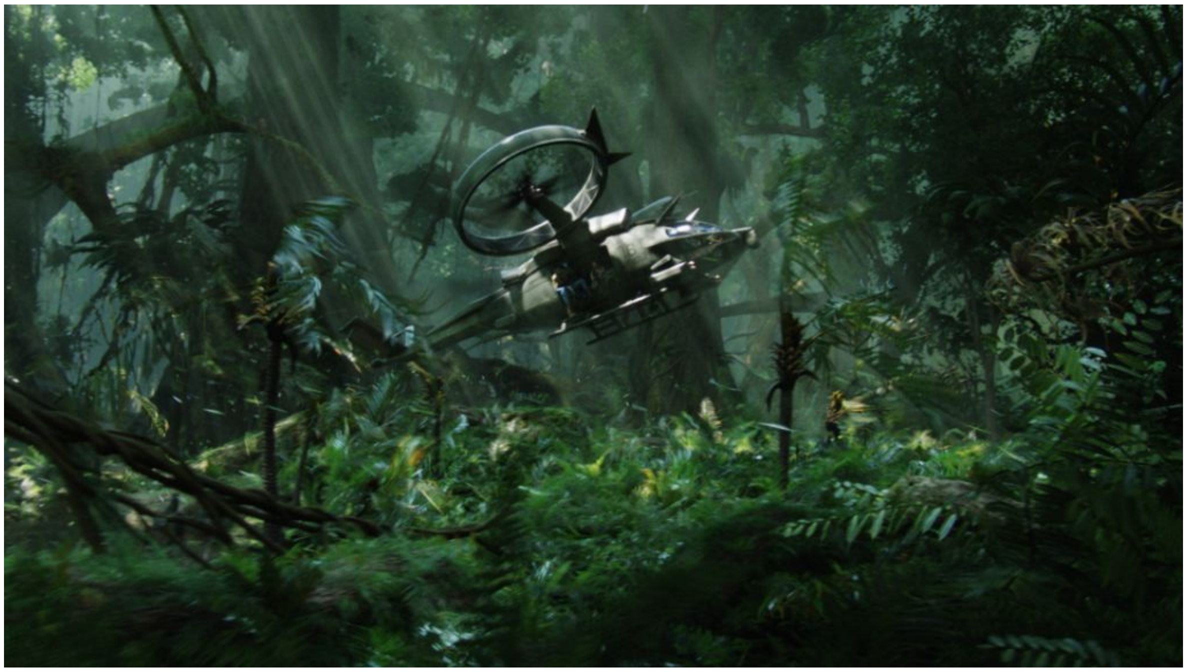


Figure 5.3: Muzzle flashes in Avatar. The amp-suit flashes were generated via fluid simulation, while the others were generated procedurally.



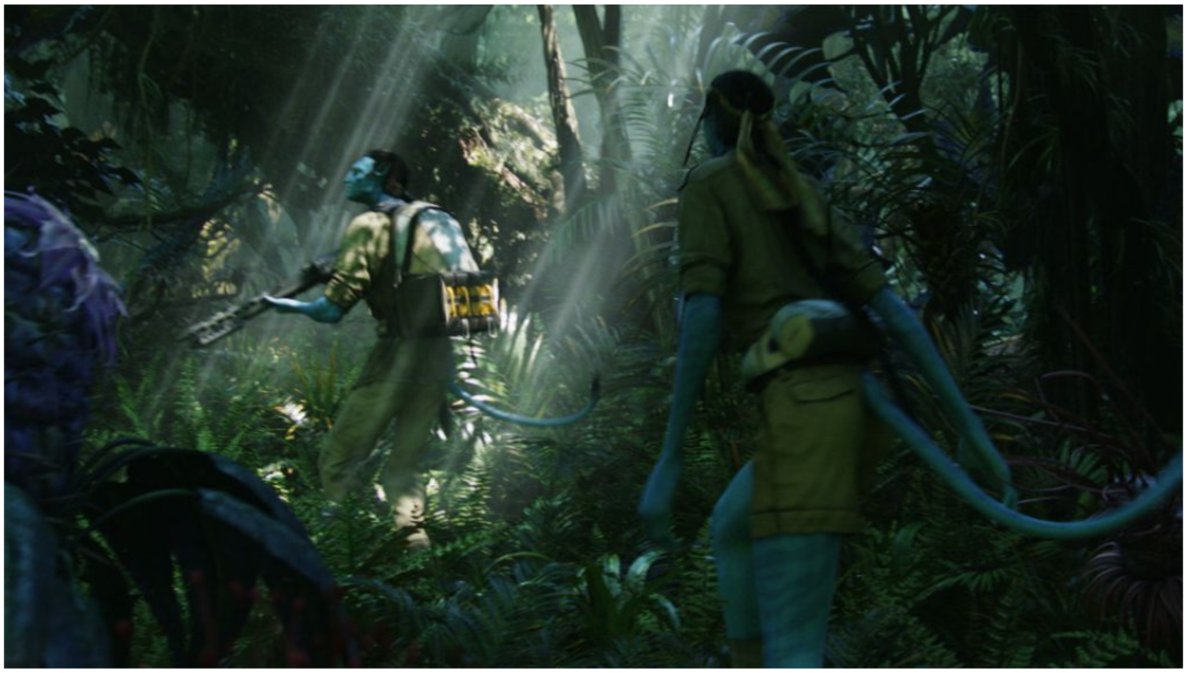


Figure 5.4: Godrays in *Avatar*. Figure 4.1 shows the rays pass alone for the second image.

Please note that the context in which these were handled was fairly different for each company: Rythm & Hues' involvement started early and encompassed the whole sequence, while Weta's involvement on the clouds sequence lasted only a few weeks and consisted of only two shots.

The cloud-heavy sky was fully modeled using our volumetric modeler (see section 2.2), with the exception of very distant clouds which were matte painted. On the rendering side, the single scattering approach that was used on Avatar was clearly not good enough for such dense clouds. We therefore developed and used the multiple scattering approximation described in section 3.3. Figure 5.5 shows the results in the final trailer shots.

Bibliography

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008. 12

[Bou08] Antoine Bouthors. *Realistic rendering of clouds in real-time*. Phd thesis, Université Joseph Fourier, june 2008. 12

[Chr08] Per H. Christensen. Point-Based Approximate Color Bleeding. Technical report, Pixar, 2008. 12

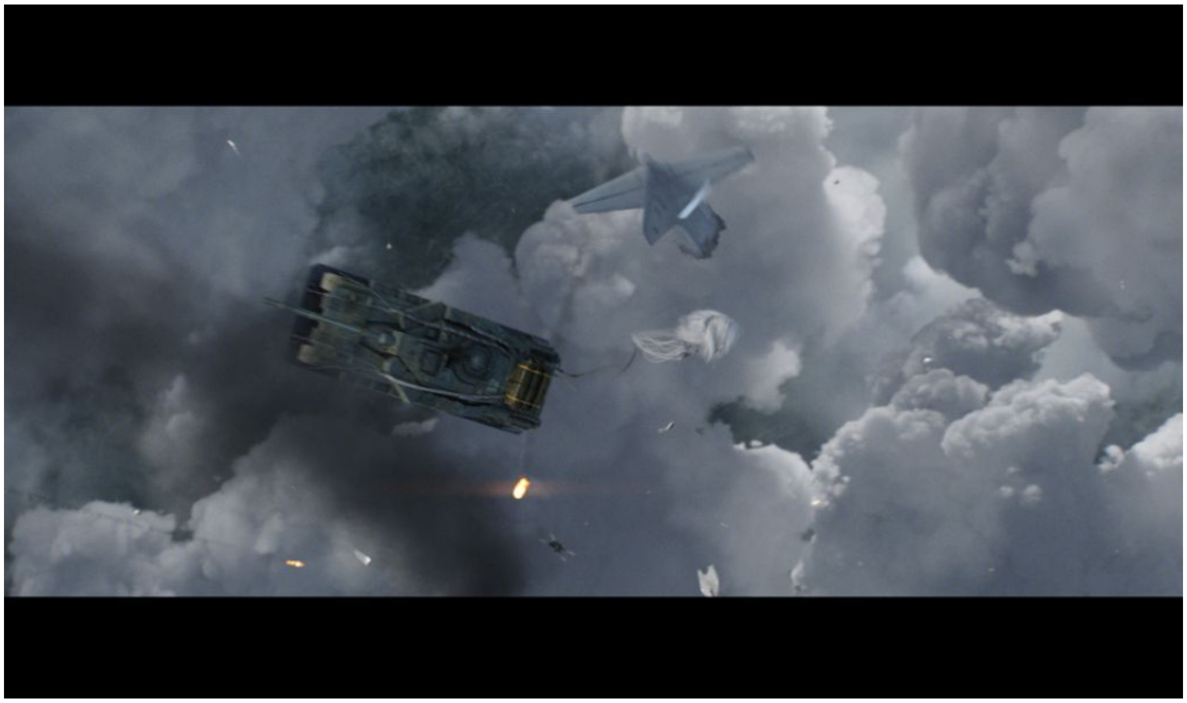
[DLR+09] Craig Donner, Jason Lawrence, Ravi Ramamoorthi, Toshiya Hachisuka, Henrik Wann Jensen, and Shree Nayar. An empirical bssrdf model. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM. 12

[Dun10] Jody Duncan. The A-Team: Plan A. *Cinefex*, 123, 2010. 19

[Ebe97] David S. Ebert. Volumetric modeling with implicit functions (a cloud is born). In *SIGGRAPH'97 Sketches*, page 245, 1997. 5

[HIOT10] Sho Hasegawa, Jason Iversen, Hideki Okano, and Jerry Tessendorf. I love it when a cloud comes together. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, pages 13:1–13:1, New York, NY, USA, 2010. ACM. 19

[HKSB06] Markus Hadwiger, Andrea Kratz, Christian Sigg, and Katja Bühler. GPU-accelerated deep shadow maps for direct volume rendering. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 49–52, New York, NY, USA, 2006. ACM. 9



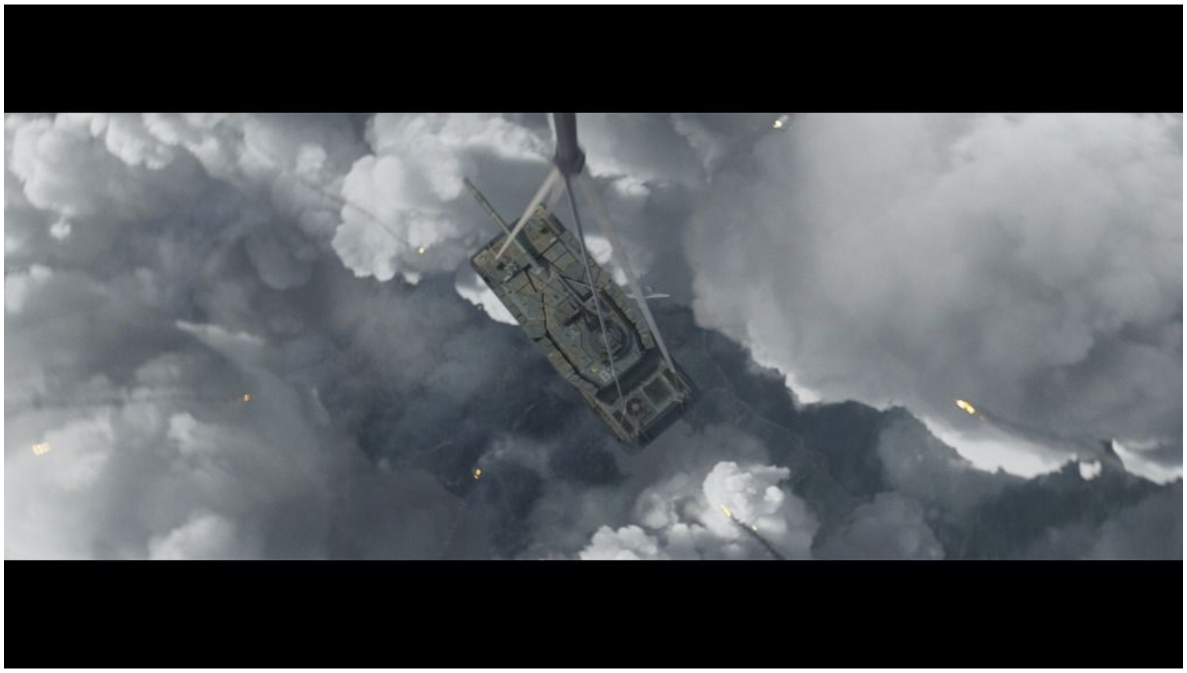


Figure 5.5: Clouds for The A-Team where modeled procedurally and shaded with our multiple scattering approximation.

[KN01] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182, London, UK, 2001. Springer-Verlag. 9

[KTJG08] Theodore Kim, Nils Thürey, Doug James, and Markus Gross. Wavelet turbulence for fluid simulation. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 50:1–50:6, New York, NY, USA, 2008. ACM. 4, 16

[LV00] Tom Lokovic and Eric Veach. Deep shadow maps. In *SIGGRAPH'00*, 2000. 11, 14

[NFJ02] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 721–728, New York, NY, USA, 2002. ACM. 11, 19

[PAS03] Simon Premože, Michael Ashikhmin, and Peter Shirley. Path integration for light transport in volumes. In *Eurographics Symposium on Rendering (EGSR)*, pages 52–63, 2003. 11, 12

[Sta95] Jos Stam. Multiple Scattering as a Diffusion Process. In *Eurographics Workshop on Rendering (EGWR)*, pages 41–50, 1995. 11

[T$^+$99] J. Tessendorf et al. Simulating ocean water. *SIGGRAPH course notes*, 2, 1999. 5

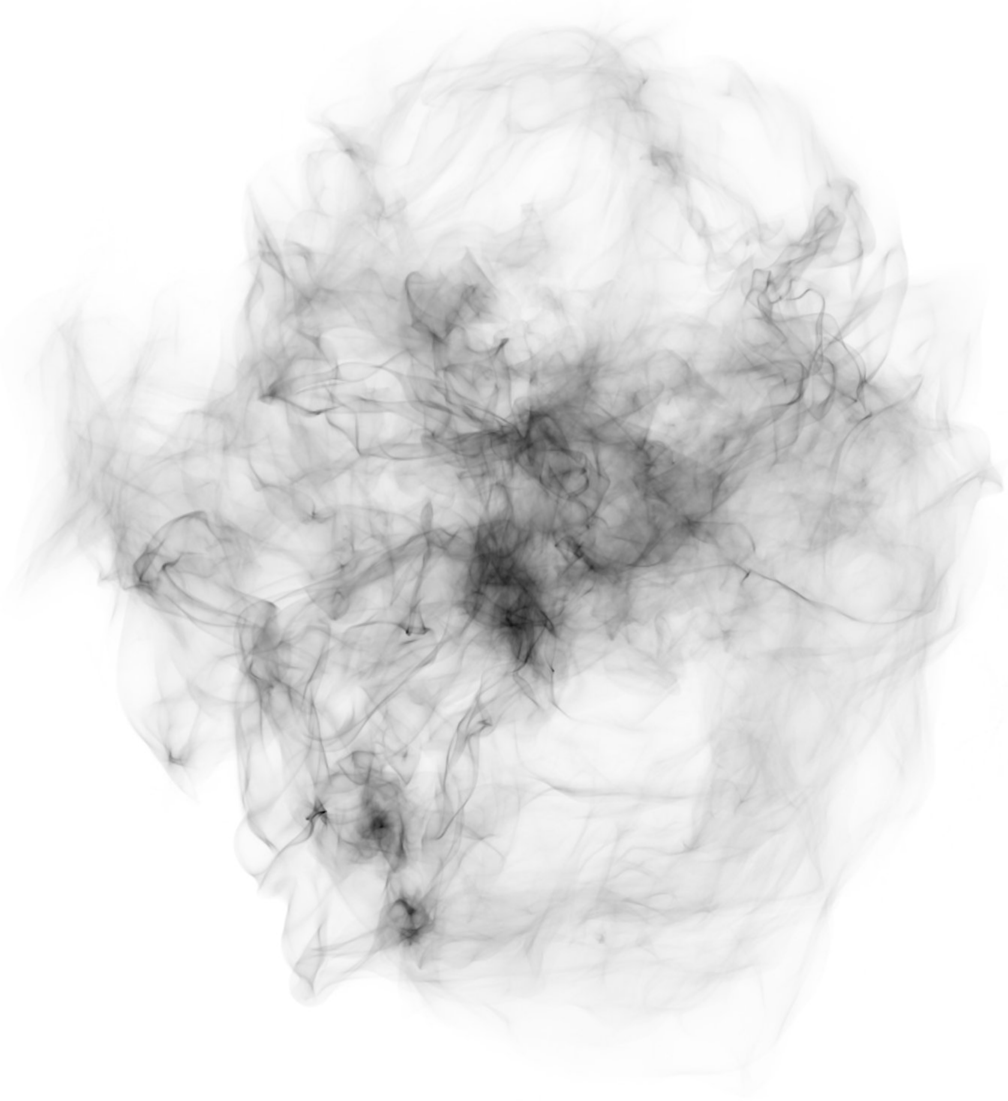
Resolution Independent Volumes

Jerry Tessendorf
School of Computing, Clemson University

Michael Kowalski
Rhythm and Hues Studios

July 24, 2011

=1
= 1.5
= -1.5
children = 1000000000
ves = 5
h = 0.5



This document can be found at http://people.clemson.edu/~jtessen/

Forward

These course notes make use of a volumetric scripting language called FELT, developed at Rhythm and Hues Studios over many years and continuing to be developed. In 2003 the earliest working version of the Rhythm and Hues Studios fluid solver, AHAB, had been built by Joe Mancewicz, Jonathan Cohen, Jeroen Molemaker, Junyong Noh, Peter Huang, and Taeyong Kim, and successfully used on the film *The Cat in the Hat*. At that point our group of simulation and volume rendering developers were thinking about what sort of tools we would need to be able to manipulate all of the volumetric data coming from simulations, and for that matter tools to create new volumetric data without simulations. We were very inspired by what TDs were telling us about Digital Domain's Storm, and its expression language in particular. But we could also see that if we were not careful about how we built a language, there might be real memory issues from creating and manipulating lots of grid-based volumes. At the same time, we could see that procedural operations like those in the area of implicit functions had a lot of nice strengths. We wanted the language to cleanly separate the application of mathematical operations on volumetric data from the discrete nature of the data. The same math – and the same code – should apply whether a volume is grid-based, particle-based, or procedural-based, and we should be able to freely mix volumes with different underlying data formats. We also wanted a language that TD's with programming knowledge could write code with, so we patterned it after shading languages, a bit of perl, and C.

By the fall of 2003, Michael Kowalski built an early version of the parser for the language, and Jonathan Cohen built the early version of the computational engine. To their great credit, years later FELT is still based on that early code with bug fixes and new features. We want to rewrite it for many reasons, not the least of which is that code under development for 7 years can get a little furry. But its quality is high enough that lots of other topics have always had higher priorities.

When the first version of FELT came out in the fall of 2003, Jerry Tessendorf inserted it into an experimental volume renderer called HOG, and started producing images of volumes generated using methods that we now refer to as gridless advection and SELMA. The imagery lead to applications for fire on *The Chronicles of Narnia: The Lion, The Witch, And The Wardrobe.* Figure 1 shows a very early test of converting hand-animated particles into a field of fire. The method worked because of its ability to create high resolution structure while simultaneously storing some of the data on grids. The design decisions allowing the mixture of data formats and resolutions were a critical success early in FELT's development.

This workflow using FELT inserted directly into volume rendering continues in production today.

In 2001, well before the conception of FELT, David Ebert invited Jerry Tessendorf to give a talk at a conference on implicit function methods. At the end of the talk he showed a photograph of a large cumulus cloud and speculated that implicit methods would allow the creation of detailed and realistic

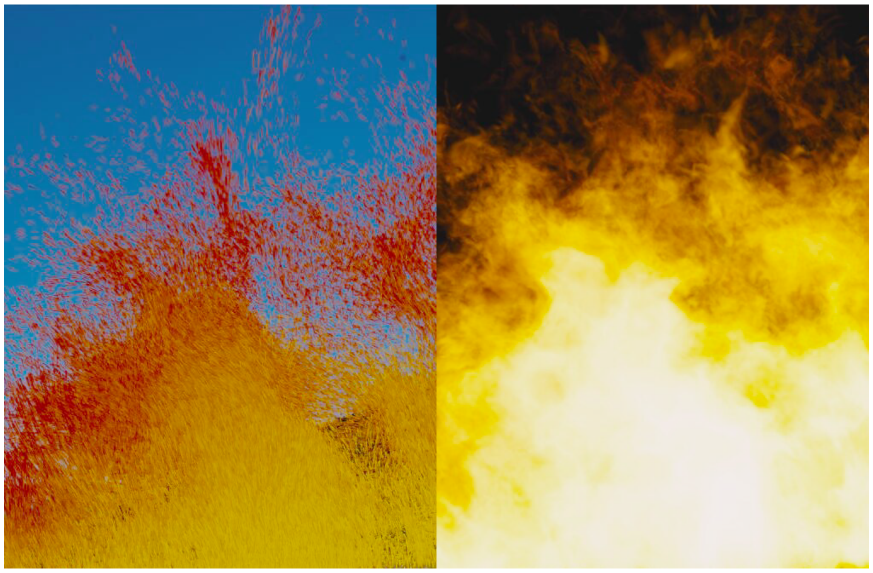


Figure 1: Early imagery showing the conversion of a particle system into a volumetric fire. The Felt algorithms used for this included early versions of gridless advection and SELMA.

cloud scenes within 10 years. Ironically, The A-Team was released in the summer of 2010, and indeed a large realistic cloud system had been constructed for the film using Felt's implicit function capabilities, just barely within the speculated time frame. The cloud modeling is described in chapter 3.

Felt has been in development for many years, and many people contributed to it as users, observers, and interested parties. Among those many people are Sho Hasegawa, Peter Huang, Doug Bloom, Eric Horton, Nathan Ortiz, Jason Iversen, Markus Kurtz, Eugene Vendrovsky, Tae Yong Kim, John Cohen, Scott Townsend, Victor Grant, Chris Chapman, Ken Museth, Sanjit Patel, Jeroen Molemaker, James Atkinson, Peter Bowmar, Bela Brozsek, Mark Bryant, Gordon Chapman, Nathan Cournia, Caroline Dahllof, Antoine Durr, David Horsely, Caleb Howard, Aimee Johnson, Joshua Krall, Nikki Makar, Mike O'Neal, Hideki Okano, Derek Spears, Bill Westinhofer, Will Telford, Chris Wachter, and especially Mark Brown, Richard Hollander, Lee Berger, and John Hughes.

Contents

List of Figures

Chapter 1

Introduction

These notes are motivated from the volumetric production work that takes place at Rhythm and Hues Studios. Over the past decade a set of tools, algorithms, and workflows have emerged for a successful process for generating elements such as clouds, fire, smoke, splashes, snow, auroras, and dust. This workflow has evolved through the production of many feature films, for example:

> The Cat in the Hat · Around the World in 80 Days · The Chronicles of Narnia: The Lion, the Witch, and the Wardrobe · Fast and Furious: Tokyo Drift · Fast and Furious 4 · Alvin and the Chipmunks · Alvin and the Chipmunks, The Squeakquel · Night at the Museum · Night at the Museum: Battle of the Smithsonian · The Golden Compass · The Incredible Hulk · The Mummy: Tomb of the Dragon Emperor · The Vampire's Assistant · Cabin in the Woods · Garfield · Garfield: A Tale of Two Kitties · The Chronicles of Riddick · Elektra · The Ring 2 · Happy Feet · Superman Returns · The Kingdom · Aliens in the Attic · Land of the Lost · Percy Jackson and the Olympians: The Lightning Thief · The Wolfman · Knight and Day · Marmaduke · The A-Team · The Death and Life of Charlie St. Cloud · Yogi Bear · Knight and Day

At the heart of this system is a multiprocessor-aware volumetric scripting language called Felt, or "Field Expression Language Toolkit". Felt has c-like syntax, and is intended to behave somewhat like a shading language for volume data. An important aspect of Felt is that it separates the notion of volumetric data from the need to store it as discrete sampled values. Felt allows purely procedural mathematical operations, and easily mixes procedural and sampled data. In this capacity, Felt scripts construct implicit functions and manipulate them, much like the methods described in [1].

In addition to modeling volume data, Felt also modifies geometry, particles, and volume data generated with other tools, including animations and simulations. This gives fine-tuning control over data in a post-process, similar to the way a compositor can fine-tune images after they are generated. Conversely, simulations can use FELT during their runtime to modify data and processing flow to suit special needs.

These tools also provide an excellent framework for prototyping new algorithms for volumetric manipulation, such as texture mapping, fracturing models, and control of simulation and modeling, which will be discussed in chapters 3, 4, 5.

1.1 A Brief on Volume Rendering

One of the primary uses of volumetric data is volume rendering of a variety of elements, such as clouds, smoke, fire, splashes, etc. We give a very brief summary of the volume rendering process as used in production in order to exemplify the kinds of volumetric data and the qualities we want it to possess. There are other uses of volumetric data, but the bulk of the applications of volumetric data is as a rendering element. A rendering algorithm commonly used for this type of data is accumulation of opacity and opacity-weighted color in ray marches along the line of sight of each pixel of an image. The color is also affected by light sources that are partially shadowed by the volumetric data.

The two fundamental volumetric quantities needed for volume rendering are the *density* and the *color* of the material of interest. The density is a description of the amount of material present at any location in space, and has units of mass per unit volume, e.g. g/m^3. The mathematical symbol given for density is $\rho(\mathbf{x})$, and it is assumed that $0 \leq \rho < \infty$ at any point of space. The color, $C_d(\mathbf{x})$, is the amount of light emittable at any point in space by the material.

The raymarch begins at a point in space called the near point, $\mathbf{x}_{near}$, and terminates at a far point $\mathbf{x}_{far}$ that is along the line connecting the camera and the near point. The unit direction vector of that line is $\mathbf{n}$, so the raymarch traverses points along the line

$$\mathbf{x}(s) \ = \ \mathbf{x}_{near} \ + \ s \ \mathbf{n}$$

with some step size Δs, for $0 \leq s \leq |\mathbf{x}_{far} - \mathbf{x}_{near}|$. In some cases, the raymarch can terminate before reaching the far point because the opacity of the material along the line of sight may saturate before reaching the far point. Raymarchers normally track the value of opacity and terminate when it is sufficiently close to 1.

The accumulation is an iterative update as the march progresses. The accumulated color, C_a and the transmissivity T are updated at each step as follows[1]:

$$\mathbf{x} \ += \ \Delta s \ \mathbf{n} \tag{1.1}$$

$$\Delta T \ = \ \exp\left(-\kappa \ \Delta s \ \rho(\mathbf{x})\right) \tag{1.2}$$

$$C_a \ += \ C_d(\mathbf{x}) \ T \ \frac{(1-\Delta T)}{\kappa} \ T_L(\mathbf{x}) \ L \tag{1.3}$$

$$T \ *= \ \Delta T \tag{1.4}$$

[1]See the appendix A for a justification of this algorithm

The field $T_L(\mathbf{x})$ is the transmissivity between the position of the light and the position $\mathbf{x}$ (usually pre-computed before the raymarch), κ is the extinction coefficient, L is the intensity of the light, and the opacity of the raymarch is $O = 1 - T$.

Flesh out the detail on the derivation of this formula. See the wiki page.

This simple raymarch update algorithm illustrates how volumetric data comes into play, in the form of the density $\rho(\mathbf{x})$ and color $C_d(\mathbf{x})$ at every point in the volume within the raymarch sampling. There is no presumption that the volume data is discrete samples on a grid or in a cloud of particles, and no assumption that the density is optically thin (although there is an implicit assumption that single scattering is a sufficient model of the light propagation). All that is needed of the volumetric data is that it can be queried for values at any point of interest in space, and the volumetric data will return reasonable values. So the data is free to be gridded, on particles, related to geometry, or purely procedural. This freedom in how the data is described is something we exploit in our resolution independent methods. The workflow consists of building the volume data for density and color in FELT, then letting the raymarcher query FELT for values of those fields.

There is an assumption in this raymarching model that the step size Δs has been chosen sufficiently small to capture the spatial detail contained in the density and color fields. If the fields are gridded data, then an obvious choice is to make the step size Δs equal to or a little smaller than the grid spacing. But we will see below several examples of fine detail produced by various manipulations of gridded data, for which the step size must be much smaller than might be expected from the grid resolution. This is a good outcome, because it means that grids can be much coarser than the final rendered resolution, and that reduces the burden on simulations and some grid-based volumetric modeling methods.

1.2 Some Conventions

There are several concepts worth defining here. A *domain* is a rectangular region, not necessarily axis-aligned, described by an origin, a length along each of its primary axes, and a rotation vector describing its orientation with respect to the world space axes. The domain may optionally have cell size information for a rectangular grid. A *field* is an object that can be queried for a value at every point in space. That does not mean that the value at all points has to be meaningful. A particular field might have useful values in some domain, but outside of that domain the value is meaningless, so it could be set to zero or some other convenient value. A *scalarfield* is a field for which the queried values are scalars. A *vectorfield* returns vectors from queries, and a *matrixfield* returns matrices. In the FELT scripting language, scalarfields, vectorfields, and matrixfields are “primitive” datatypes. You can define them and do calculations with them, but it is not necessary to explicitly program what happens at every point in space.

In these notes, scripts written in FELT will have a font and color like this:

```
scalarfield r = sqrt( identity()*identity() );
// Comments are in this color and use C++ comment symbols "//"
vectorfield normal = grad(r);
```

This simple script is equivalent to the mathematical notation:

$$\begin{aligned} r &= \sqrt{\mathbf{x} \cdot \mathbf{x}} \\ \mathbf{n} &= \nabla r \end{aligned}$$

because the function identity() returns a vectorfield whose value is equal to the position in space, and the * product of two vectorfields is the inner product.

For the times that it is useful to have data that consists of values sampled onto a grid, the companion objects to fields are *caches*, in the form of *scalarcache* and *vectorcache*.

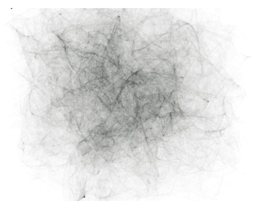
```
scalarfield r = sqrt( identity()*identity() );
vectorfield normal = grad(r);

// Create a domain: axis-aligned 2x2x2 box centered at the (0,0,0)
vector origin = (-1,-1,-1);
vector lengths = (2,2,2); //  2x2x2 box
vector orientation = (0,0,0); // Axis-aligned
float cellSize = 0.1;
domain d( origin, lengths, orientation, cellSize, cellSize, cellSize );

// Allocate two caches based on the domain
scalarcache rCache( d );
vectorcache normalCache( d );

// Sample fields r and normal into caches
cachewrite( rCache, r );
cachewrite( normalCache, normal );

// Treat caches like fields, using interpolation
scalarfield rSampled = cacheread( rCache );
vectorfield normalSampled = cacheread( normalCache );
```

In the last lines of this script the gridded data is wrapped in a field description, because interpolation schemes can be applied to calculate values in between grid points. But once this is done, they are essentially fields, and the gridded nature of the underlying data is completely hidden, and possibly irrelevant to any other processing afterward.

Note that the construction of the sampled normal field, normalSampled, could have been accomplished in a different, more compact approach:

```
scalarfield r = sqrt( identity()*identity() );

// Create a domain: axis-aligned 2x2x2 box centered at the (0,0,0)
vector origin = (-1,-1,-1);
vector lengths = (2,2,2); //  2x2x2 box
vector orientation = (0,0,0); // Axis-aligned
float cellSize = 0.1;
domain d( origin, lengths, orientation, cellSize, cellSize, cellSize );

// Allocate one cache based on the domain
scalarcache rCache( d );

// Sample field r into the cache
cachewrite( rCache, r );

// Treat the cache like a field, using interpolation
scalarfield rSampled = cacheread( rCache );

// Take the gradient of the sampled field rSampled
vectorfield normalSampled = grad( rSampled );
```

Here, only one cache is used and the gradient is applied to the sampled version of the distance rSampled. The two approaches are conceptually very similar, and numerically very similar, but not identical. In the previous method, the term grad(r) actually computes the mathematically exact formula for the gradient, and in that case normalCache contains exact values sampled at gridpoints, and normalSampled interpolates between exact values. In the latter method, grad(rSampled) contains a finite-difference version of the gradient, so is a reasonable approximation, but not exactly the same. For any particular application though, either method may be preferrable.



Chapter 2

The Value Proposition for Resolution Independence

In volume modeling, animation, simulation, and computation, resolution independence is a handy property for many reasons that we want to review here. But first, we need to be clear about what the term "resolution independent" means.

First the negative definition. Resolution independence does *not* mean the volume data is purely procedural. Procedurally defined and manipulated data are very useful, but not always the best way of handling volume problems. There are many times when gridded data is preferrable.

A system that manipulates volumes in a resolution independent way has two properties:

1. While the creation of volume data may sometimes require that a discrete representation be involved (e.g. a rectangular grid or a collection of particles), there are many manipulations that do not explicitly invoke the discrete nature that the data may or may not have. For example, given two scalarfields sf1 and sf2, a third scalarfield sf3 can be constructed as their sum:

   ```
   scalarfield sf3 = sf1 + sf2;
   ```

   But this manipulation does not require that we explicitly tell the code how to handle the discrete nature of the underlying data. Each scalarfield handles its own discrete nature and hides that completely from all other fields. In fact, there isn't even a reason why the scalarfields have to have the same discrete properties. This operation makes sense even if sf1 and sf2 have different numbers of gridpoints, different resolutions, different particle counts, or even if one or both are purely procedural. Which leads to the second property:

2. Resolution independence means that fields with different discrete properties can be combined and manipulated together on equal terms. This is analogous to the behavior of modern 2D image manipulation software, such as Photoshop or Nuke. In those 2D systems, images can be combined without having equal numbers of pixels or even common format. Vector graphics can also be invoked for spline curves and text. All of this happens with the user only peripherally aware that these differences exist in the various image data sets. The same applies to volumes. We should be able to manipulate, combine, and create volume data regardless of the procedural or discrete character of each volumetric object.

Resolution independent volume manipulation is a good thing for several reasons:

Performance Trade-Offs
: Some volumetric algorithms have many computational steps. If we have access only to discrete volumetric data, then each of these steps requires allocating memory for the results. In some cases the algorithm lets you optimize this so that memory can be reused, but in other cases the algorithm may require that multiple sets of discrete data be available in memory. This can be a severe constraint on the size of volumetric problem that can be tackled. The alternative offered by resolution independence is that the computational aspects are divorced from the data storage. Consequently, an arbitrary collection of computational steps can be implemented procedurally and evaluated numerically without storing the results of each individual step in discrete samples. Only the outcome of the collection need be sampled into discrete data, and only if the task at hand required it. This is effectively a trade-off of memory versus computational time, and there can be situations in which caching the computation at one or more steps has better overall performance. Resolution independence allows for all options, mixing procedural steps with discretely sampled steps to achieve the best overall performance, balancing memory and computational time freely. This performance trade-off is discussed in detail for the particular case of gridless advection and Semi-Lagrangian Mapping (SELMA) in chapters 7 and 8.

Targeted grid usage
: Manipulation of fields that are gridded does not automatically generate gridded results. The user has to explicitly call for sampling and caching of the the field into a grid. While this means extra effort when gridding is desired, it is a benefit because the user has full control over when grids are invoked, and even what type of gridding is used. This targeting of when data is sampled is illustrated by Semi-Lagrangian Mapping (SELMA), which solves performance problems encountered in gridless advection by a judicious choice of when and how to sample a mapping function onto a grid. This same reasoning applies to other forms of discretized data sampling as well.

Procedural high resolution

There are many procedural algorithms that enhance the visual detail of volumetric data. One example of this is gridless advection, discussed in chapter 7. This increased detail is produced whether the original data is discrete or procedural. So much detail can be generated that it can become difficult to properly render it in a raymarch.

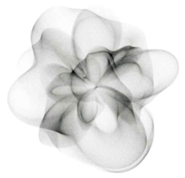
Cleaner coding of algorithms

When data is gridded or discretized, there are parameters involved that describe the discrete environment (cell size, number of points, location of grid, etc.). Manipulation of volume data just in terms of fields does not require invoking those parameters, and so allows for simplified code structure. Algorithms are developed and implemented without worrying about the concepts related to what format the data is in. For example, the FELT codes for warping fields and fracturing geometry in chapters 4 and 5 are completely ignorant of any notion that the input data is discretized, and make no accomodations for such. The FELT scripts are extremely compact as a result.

Calculations only where/when needed

Suppose you have a shot with the camera moving past a large volumetric element (or the element moving past the camera), and the element itself is animating. There may also be hard objects inside the volume that hide regions from view. You might handle this by generating all of the data on a grid for each frame. Or you might have a procedure for figuring out ahead of time which grid points will not be visible to the camera and avoid doing calculations on them. In the resolution independent procedures discussed here neither of those approaches is needed, because calculations are executed only at locations in space (on grid points or not) and at times in the processing at which actual values for the field are needed. In this case a raymarch render queries density and color, and field calculations are executed only at the locations of those queries at the time of each query.

In the remaining chapters, resolution independence is used as an integral part of each of the scripting examples discussed.



Chapter 3

Cloud Modeling

Natural looking clouds are *really* hard to model in computer graphics. Some of the reasons for it are physics-based: there is a broad collection of physical phenomena that are simultaneously important in the process of cloud formation and evolution - thermodynamics, radiative transfer, fluid dynamics, boundary layer conditions, global weather patterns, surface tension on water droplets, the wet chemistry of water droplets nucleating on atmospheric particulates, condensation and rain, ice formation, the bulk optics of microscopic water droplets and ice crystals, and more. There are also reasons related to the application: if you need to model the volumetric density and optics of clouds in 3D for production purposes, it usually means you need to model an entire cloud over distances of hundreds of meters to kilometers, but resolve centimeter-sized detail within it. Putting together a coherent 3D spatial structure than covers eight orders of magnitude in scale is not a straightforward proposition. Real clouds exhibit a variety of spatial patterns across those scales, some of them statistical in character and some more (fluid) dynamical. For production, we need tools that can mix all of that together while being controllable from point-to-point in space.

Volume modeling methods have developed sufficiently to take on this task. Levelsets and implicit surfaces provide a powerful and flexible description of complex shapes. The pyroclastic displacement method of Kaplan[2] captures some of the basic cauliflower-like structure in cumulous cloud systems. Gridless advection (chapter 7) generates fluid and wispy filaments around cloud boundaries. Procedural modeling with systems like FELT let us combine these with additional algorithms to produce enormous and complex cloud systems with arbitrary spatial resolution.

The algorithms presented in this chapter were used for the production of visual effects in the film *The A-Team* at Rhythm and Hues Studios. We begin with a look at some photos of cumulous clouds and a description of interesting features that we want the algorithms to incorporate.

3.1 Cumulous cloud structure of interest

Figure 3.1 shows two photographs of strong cumulous cloud systems viewed from above. The top photo shows a much larger cloud system than the bottom one. There are several features of interest in the photos that we want to highlight:

Clustering
Cumulous clouds look something like cauliflower in that they are bumpy, with a seemingly noisy distribution of the bumpiness across the cloud. This sort of appearance is achievable by a pyroclastic displacement of the cloud surface using Perlin or some other spatially smooth noise function.

Layering
The bumpiness is mutlilayered, with small bumps on top of large bumps. Pyroclastic displacement does not quite achieve this look by itself, but iterating displacements creates this layering, i.e., applying smaller scale displacements on top of larger ones.

Smooth valleys The deeper creases, or valleys, in a cumulous cloud appear to be smooth, without the layering of displacements that appears higher up on the bumps. The iterated displacements must be controllable so that displacements can be suppressed in the valleys, with controls on the magnitude of this behavior.

Advected material Despite the hard-edge appearance of many cumulous clouds, as they evolve the hardness gives way to a more feathered look because of advection of cloud material by turbulent wind. This advection occurs at different times and with different strengths within the cloud.

Spatial mixing All of the above features occur to variable degree throughout the cloud system, so that some parts of the cloud may have many layers of bumps while others are relatively smooth, and yet others are diffused from advection. The cloud modeling system needs to be able to mix all of these features at any position within a cloud to suit the requirements of the production.

Each of these features is discussed below. The algorithm is based on representing the overall shape of the cloud as a levelset, pyroclastically displacing that levelset multiple times, converting the levelset values into cloud density, then gridlessly advecting the density. Along with those major steps, all of the control parameters are spatially adjustable in the FELT implementation because the controls are scalarfields and vectorfields that are generated from point attributes on the undisplaced cloud geometry.

3.2 Levelset description of a cloud

Cloud modeling begins with a base shape for the smooth shape of the cloud. This can be in the form of simple polygonal geometry, but with sufficient quality

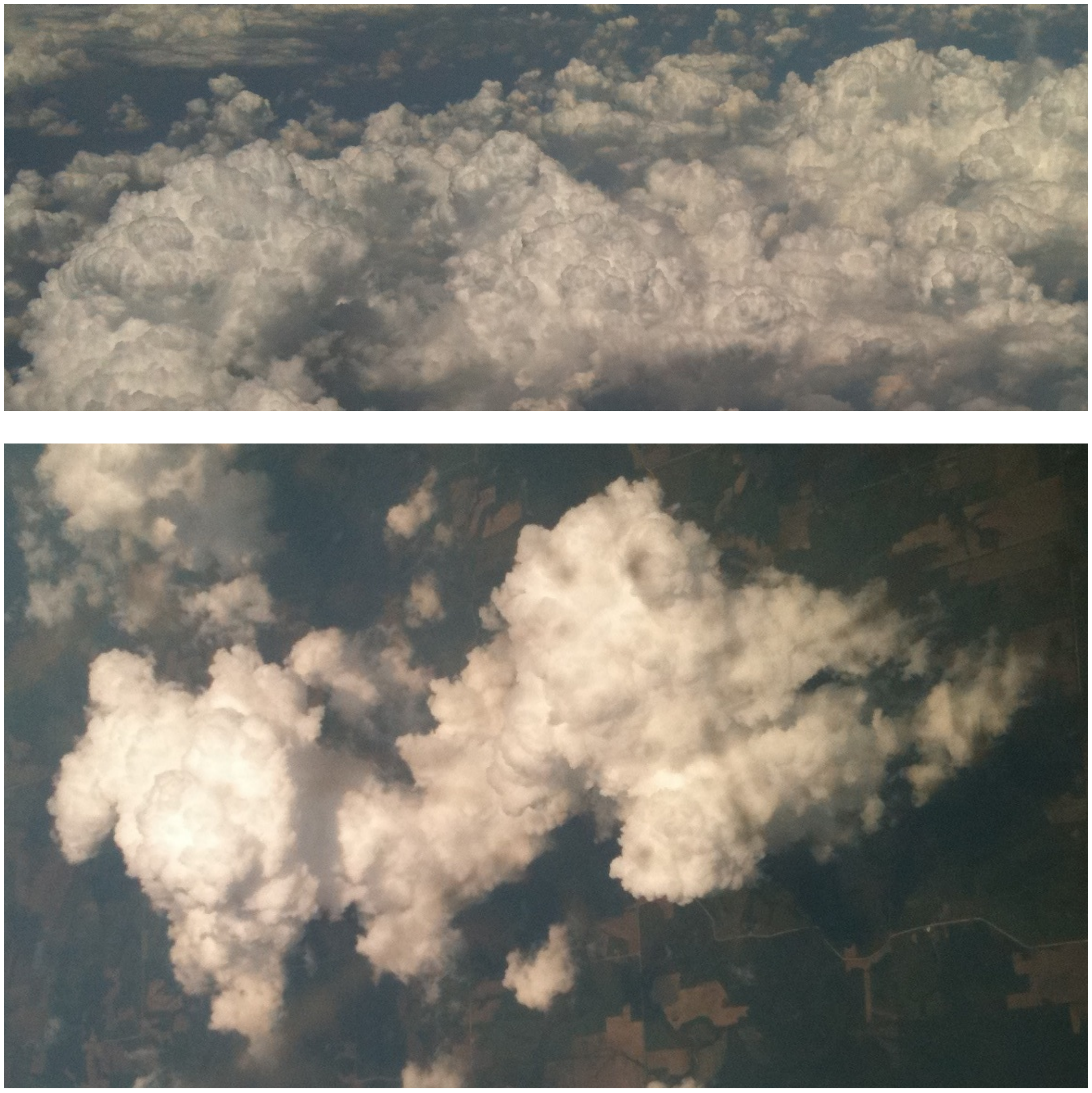


Figure 3.1: Aerial photos of cumulous clouds. Structures of interest: the pyroclastic-like buildup of clusters; the relatively smooth "valleys" between the clusters; dark fringes along the edges of clusters; bright bands of light in the "valleys"; softened regions due to advection of material.

that it can be turned into a scalarfield known as a levelset. The levelset of the base cloud, $\ell_{\text{base}}(\mathbf{x})$ is a signed distance function, with positive values inside the geometry and negative values outside. The spatial contour $\ell_{\text{base}}(\mathbf{x}) = 0$ is a surface corresponding to the model geometry for the cloud.

The volumetric density of the cloud can be obtained at any time by using a mask function to generate uniform density inside the cloud:

$$\rho_{\text{base}}(\mathbf{x}) = \text{mask}\left(\ell_{\text{base}}(\mathbf{x})\right) = \begin{cases} 1 & \ell_{\text{base}}(\mathbf{x}) > 0 \\ 0 & \ell_{\text{base}}(\mathbf{x}) \leq 0 \end{cases} \tag{3.1}$$

Of course, clouds are not uniformly dense in their interiors. For our purposes here, we will ignore that and generate clouds with uniform density in their interior. This limitation is readily removed by adding spatially coherent noise to the interior if desired.

3.3 Layers of pyroclastic displacement

The clustering feature has been successfully modeled in the past by Kaplan[2] using a Perlin noise field to displace the surface of a sphere. This effect is also refered to as a pyroclastic appearance. Figure 3.2 shows two examples of a spherical volume with the surface displaced by sampling Perlin noise on its surface. By adjusting the number of octaves, frequency, roughness, etc, a variety of very effective structures can be produced[4]. But for cloud modeling, we need to extend this approach in two ways. First, we need to be able to apply these displacements to arbitrary closed shapes, not just spheres, so that we can model base shapes that have complex structure initially and apply the displacements directly to those shapes. Second, to accomodate the layering feature in clouds, we need to be able to apply multiple layers of displacement noise in an iterative way. Both of these requirements can be satisfied by one process, in which the surface is represented by a levelset description. Applying displacements amounts to generating a new levelset field, and that can be iterated as many times as desired.

We describe the levelset approach based on the spherical example, then launch into more complex base shapes.

3.3.1 Displacement of a sphere

The algorithm for calculating the density of a pyroclastic sphere at any point in space is as follows:

1. Calculate the distance from the point of interest $\mathbf{x}$ to the center of the sphere $\mathbf{x}_{\text{sphere}}$:
$$d = |\mathbf{x} - \mathbf{x}_{\text{sphere}}| \tag{3.2}$$
2. Compare d to the displacement bound d_{bound} of the Perlin noise and the radius R of the sphere. If $d < R$, $\mathbf{x}$ is definitely inside the pyroclastic

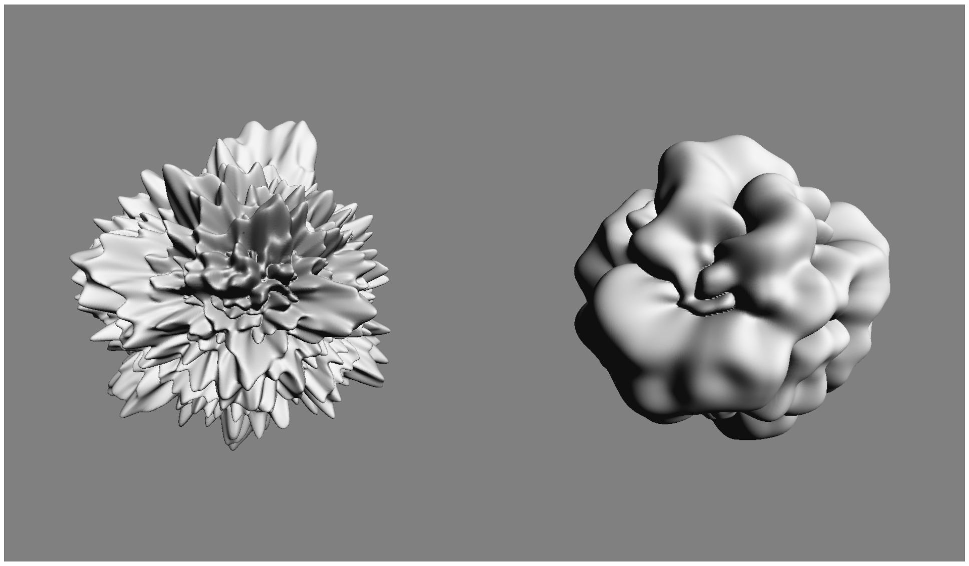


Figure 3.2: Examples of classic pyroclastically displaced spheres of density.

sphere, and the density is 1. If $d > R + d_{\text{bound}}$, then the point $\mathbf{x}$ is definitely outside of the pyroclastic sphere, density is 0.

3. If $0 < d - R < d_{\text{bound}}$, then compute the displacement: The point on the unit sphere surface is $\mathbf{n} = (\mathbf{x} - \mathbf{x}_{\text{sphere}})/d$. The displacement is $r = |\text{Perlin}(\mathbf{n})|$. If $d - R < r$, the point $\mathbf{x}$ is inside the pyroclastic sphere and the density it 1. Otherwise, the density is 0. The absolute value of the noise is used because it produces sharply cut "canyons" and smoothly rounded "peaks".

This algorithm is particularly clean because the base shape is a sphere, for which the mathematics is simple. More general base shapes would require some method of moving from a point in space $\mathbf{x}$ to a suitable corresponding point on the base shape, $\mathbf{x}_{\text{base}}$ in order to sample the displacement noise on the surface of the shape.

Layering provides an additional complication. For a sphere, you might imagine applying multiple layers of displacements by simply adding multiple displacements by $r_i = \text{Perlin}_\text{i}(\mathbf{n})$ for multiple choices of Perlin noise. But that would not really be sufficient, because successive layers should be applied by sampling the noise on the surface of the previously generated displaced surface, using the displaced normal to the base shape. For layering, the noise sampling of each layer should be on the surface displaced by previous layer(s), and the displacement direction should be the normal to the previously displaced surface. This leads to the same issue that the base shape for a displacement may be very complex.

Both of these issues are solved by expressing the algorithm in terms of levelsets.

3.3.2 Displacement of a levelset

Suppose you want to displace a shape that is represented by the levelset $\ell(\mathbf{x})$. The displacement will be based on the noise function $N(\mathbf{x})$ which is some arbitrary scalar field. Note that the field $\ell + N$ is also a levelset for some shape, but that shape need not resemble the original one in any way because the sum field can introduce new surface regions that are unrelated to the ℓ. For the pyroclastic style of displacement, we need to displace only by the value of the noise function on the surface of ℓ. The procedure is:

1. At position $\mathbf{x}$, find the corresponding point $\mathbf{x}_\ell(\mathbf{x})$ on the surface of ℓ. This is generally accomplished by an iterative march toward the surface:

$$\mathbf{x}_\ell^{n+1} = \mathbf{x}_\ell^n \ - \ \ell(\mathbf{x}_\ell^n)\frac{\nabla\ell(\mathbf{x}_\ell^n)}{|\nabla\ell(\mathbf{x}_\ell^n)|} \tag{3.3}$$

   for which typically 3-5 iterations are needed.

2. Evaluate the noise at the surface: $N(\mathbf{x}_\ell)$. Note that many locations $\mathbf{x}$ in general map to the same location $\mathbf{x}_\ell$ on the surface, and so have the same surface noise.

3. Create a new levelset field based on displacement by the noise at the surface:

$$\ell_N(\mathbf{x}) = \ell(\mathbf{x}) + |N(\mathbf{x}_\ell(\mathbf{x}))| \tag{3.4}$$

This levelset-based approach produces effectively the same algorithm as the one for the sphere when the levelset is defined as $\ell(\mathbf{x}) = R - |\mathbf{x} - \mathbf{x}_{\text{sphere}}|$, although it is not as computationally efficient for that special case.

This is a very powerful general algorithm that works for problems with huge ranges of spatial scales. It also provides the solution for layering. Suppose you want to apply M layers of displacement, with $N_i(\mathbf{x}), i = 1, \ldots, M$ being the displacement fields. Then we can apply the iteration

$$\ell_{N_{i+1}}(\mathbf{x}) \ = \ \ell_{N_i}(\mathbf{x}) \ + \ \left|N_{i+1}\left(\mathbf{x}_{\ell_{N_i}}(\mathbf{x})\right)\right| \tag{3.5}$$

to arrive at the final displaced levelset $\ell_{N_M}(\mathbf{x})$.

In terms of Felt code, this multilayer displacement algorithm is implemented in a function called *cumulo*, with inputs consisting of the base levelset, and an array of displacement scalarfields, and implements a loop

```
func scalarfield cumulo( scalarfield base, scalarfield[] displacementArray,
int iterations )
{
   scalarfield out = base;
```

```
       for( int i=0; i<size(displacementArray);i++ )
       {
          vectorfield surfaceX = levelsetsurface( out, iterations );
          out += compose(abs(displacementArray[i]), surfaceX );
       }
       return out;
    }
```

The FELT function `levelsetsurface( scalarfield levelset, int iterations )` generates a vectorfield that performs the iterations in equation 3.3 for the input levelset scalarfield, and `compose(A,B)` evaluates the field `A` at the locations in the vectorfield `B`.

Figure 3.3 illustrates the effect of layering pyroclastic displacements. This figure displays the geometry generated from the levelset data after layering has been applied. In this example, successive layers contain higher frequency noise.

3.3.3 Layering strategy

Just as important as the functionality to add layers of displacement, is the strategy for generating and applying those layers to achieve maximum efficiency and control the look of the layers. While equation 3.5 is implemented procedurally in the `cumulo` FELT code, a purely procedural implementation is not always the most efficient strategy for using `cumulo`. Judicious choices for when to sample and what data to sample onto a grid improve the speed without sacrificing quality.

In this subsection we look at the process of creating the displacement noise for each layer, and schemes for sampling intermediate levelset data onto grids to improve efficiency.

Fractal layering

One way to set up the layers of displacement is by analogy with fractal summed perlin noise[4]. For $N_{octaves}$, a base frequency f, frequency jump f_{jump}, and amplitude roughness r, the fractal sum of a noise field $PN(\mathbf{x})$ is

$$FS(\mathbf{x}) = \sum_{i=0}^{N_{octaves}-1} r^i \; PN\left(\mathbf{x} \; f \; f_{jump}^i\right) \tag{3.6}$$

This kind of fractal scaling is a natural-looking type of operation for generating spatial detail. It is also very flexible and easy to apply. Applying this to layering, each layer can be a scaled version of a noise function, i.e. each layer corresponds to one of the terms in the fractal sum:

$$N_i(\mathbf{x}) = r^i \; FS\left(\mathbf{x} \; f \; f_{jump}^i\right) \tag{3.7}$$

In terms of FELT code, we have:

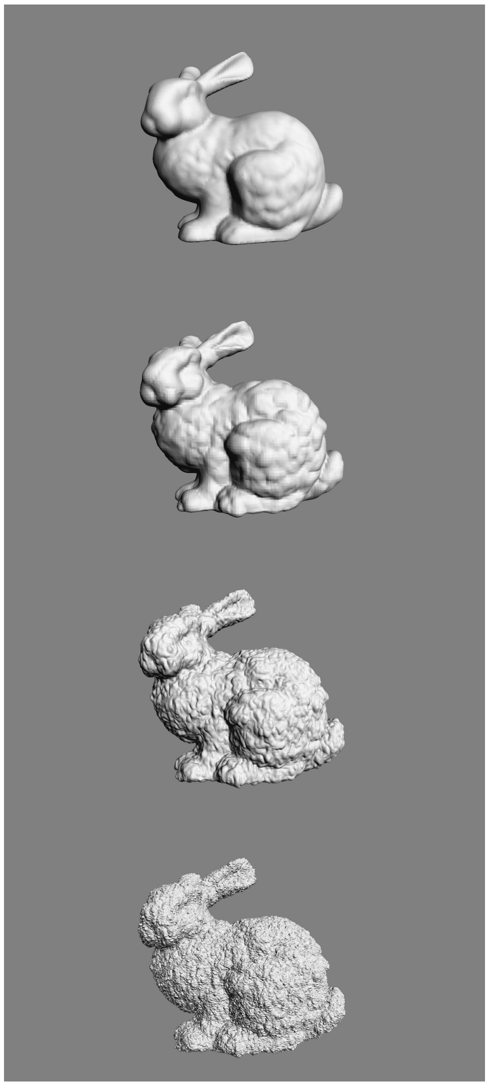


Figure 3.3: Illustration of layering of pyroclastic displacements. From top to bottom: No displacements; one layer of displacements; two layers; three layers. The displacements are applied to the levelset representation of the bunny, and the displaced bunny was converted into geometry for display.

```
 // Function to generate and array of displacement layers
func scalarfield[] NoiseLayers( int nbGenerations, scalarfield scale, scalarfield
fjump, scalarfield freq, scalarfield rough )
{
    scalarfield[] layerArray;
   // Choose a noise function as a field, e.g. Perlin, Worley, etc.
    scalarfield noise = favoriteNoiseField();
    scalarfield freqScale = freq;
    scalarfield ampScale = scalarfield(1.0);
    for( int i=0;i<nbGenerations;i++ )
    {
       layerArray[i] = compose( noise, identity()*freqScale ) * ampScale;
      // Fractal scaling of frequency and amplitude
       freqScale *= fjump;
       ampScale *= rough;
    }
    return layerArray;
}
```

This FELT code is more general than equation 3.7 because the fractal parameters fjump, freq, rough in the code are scalarfields. By setting these parameters up as scalarfields, we have spatially varying control of the character of the displacement layers.

Selectively sampling the levelset into grids

The purely procedural layering process embodied in equation 3.5 is compact, flexible, and powerful, but can also be relatively slow. We can exploit the fractal layer approach to speed up the levelset evaluation. The crucial property here is that the each fractal layer represents a range of spatial scales that is higher frequency that the previous layers. Conversely, an early layer has relatively large scale features. This implies that sampling the levelset into a grid that has sufficient resolution to capture the spatial features of one layer still allows subsequent layers to apply higher spatial detail displacements. Suppose we know that layer m has smallest scale Δx_m. We could build a grid with Δx_m as the spacing of grid points, sample the levelset ℓ_m into that grid, and replace ℓ_m with the gridded version. This replacement would be relatively harmless, but evaluating ℓ_m in subsequent processing would be much faster because the evaluation amount to interpolated sampling of the gridded data. This process can be applied at each level, so that the layered levelset equation 3.5 is augmented with grid sampling, and the FELT code is augmented to

```
func scalarfield cumulo( scalarfield base, scalarfield[] displacementArray,
int iterations, domain[] doms )
{
   scalarfield out = base;
```

```
    for( int i=0; i<size(displacementArray);i++ )
    {
        vectorfield surfaceX = levelsetsurface( out, iterations );
        out += compose(abs(displacementArray[i]), surfaceX );
        // Sample the levelset to a cache.
        // Each cache has a different resolution in its domain.
        scalarcache outCache( doms[i] );
        cachewrite(outCache, out);
        out = cacheread(outCache);
    }
    return out;
}
```

This change can increase the speed of evaluating the levelset dramatically, and if the domains are chosen reasonably there need be no significant loss of detail. It also provides a way to save the levelset to disk so that it can be generated once and reused.

3.4 Clearing Noise from Canyons

Within the "canyons" in the reference clouds in figure 3.1 the amount of finescale noisy displacement is much less than around the "peaks" of the cloud pyroclastic displacements. We need a method of suppressing displacements within those valleys. It would be very tedious if we had to analyze the structure of the multiply displaced levelset to identify the canyons for subsequent noise suppression. Fortunately there is a much simpler way of do it that can be applied efficiently.

If we look at the noise function in equation 3.5, the clearing can happen if we modulate that expression by a factor that goes to zero in the regions where all of the previous layers of noise also go to zero. At the same time, away from the zero-points of the previous layers, we want this layer to have its own behavior driven by its noise function. Both of these goals are accomplished modifying N_i to a cleared version N_i^c as

$$N_i^c(\mathbf{x}) \;=\; N_i(\mathbf{x}) \left(\text{clamp}\left(\frac{N_{i-1}^c(\mathbf{x})}{Q}, 0, 1 \right) \right)^{billow} \tag{3.8}$$

In this form, the factor Q is a scaling function that is dependent on the noise type. The exponent *billow* controls the amount of clearing that happens. This additional factor modulates the current layer of noise by a clamped value of the previous layer, reduces the current layer to zero in regions where the previous layer is zero. Once the previous layer of noise reaches the value Q, the clamp saturates at 1 and the current layer is just the noise prescribed for it. Figure 3.4 shows a wedge of billow settings, visualized after converting the levelset into geometry. These same results are shown as volume renders in figure 3.5. Note that for large billow values the displacements are almost completely cleared over most of the volume, with the exception of narrow regions at the peak of displacement.

3.5 Advection

Another tool for cloud modeling is gridless advection, which is described in detail in chapter 7. Even the hardest-edged cumulous cloud evolves over time to have ragged boundaries and softened edges due to advection of the cloud material in the turbulent velocity field in the cloud's environment. We can emulate that effect by generating a noisy velocity field and applying gridless advection at render time. The gridless advection also produces very finely detailed structure in the cloud, as seen in the foreground clouds in figure 3.6 from the production work on the film *The A-Team*. In fact, the detail is sufficient that the hard-edged cumulo structure could be modeled using layered pyroclastic displacements down to scales of 1 meter, then gridless advection carried the detail down to the finest resolved structure (about 1 cm) rendered in the production.

A suitable noisy velocity field can be built from Perlin noise by evaluating the noise at three slightly offset positions, i.e.

$$\mathbf{u}_{noise}(\mathbf{x}) \;=\; (\mathrm{Perlin}(\mathbf{x}),\ \mathrm{Perlin}(\mathbf{x}+\Delta x_1),\ \mathrm{Perlin}(\mathbf{x}+\Delta x_2)) \tag{3.9}$$

where Δx_i are two offsets chosen for effect. This velocity field is not incompressible and so might not be adequate for some applications. But for gridlessly advecting cumulous cloud models, it seems to be sufficient. Figure 3.7 shows gridlessly advected cloud for several magnitudes of the noisy velocity field. In the strongest one you can clearly see portions of cloud separated from the main body. A wide variety of looks can be created by adjusting the setting of each octave of the noisy velocity field.

3.6 Spatial control of parameters

Clouds have extreme variations in their structure, even within a single cloud system or cumulous cluster. Even if the basic structural elements were limited to just the ones we have built in this chapter, the parametric dependence varies dramatically from region to region in the cloud. To accomodate this variability, we implemented the FELT script for the noise layers using scalarfields for the parameters. This field-based parameterization can also be extended to generating the advection velocity and canyon clearing billow parameter. Figure 3.8 shows a bunny-shaped cloud with uniform density inside, and spatially varying amounts of pyroclastic displacement of the volume. The control for this was several procedural fields for ramps and local on-switches to precisely isolate the regions and apply different parameter settings.

But given this extension, we also need a mechanism for creating these fields for the basic parameters. An approach that has been successful uses point

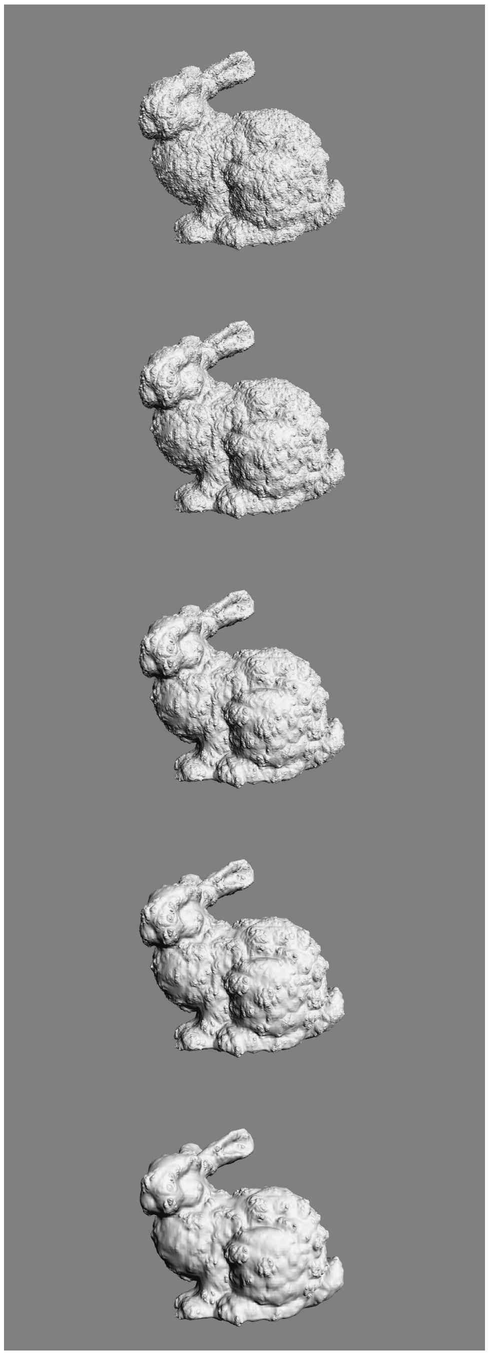


Figure 3.4: Illustration of clearing of displacements in the valleys using the billow parameter. The bottom of figure 3.3 illustrates the three layers of displacement with no billow applied. The noise is FFT-based, and $Q = 1$. From top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

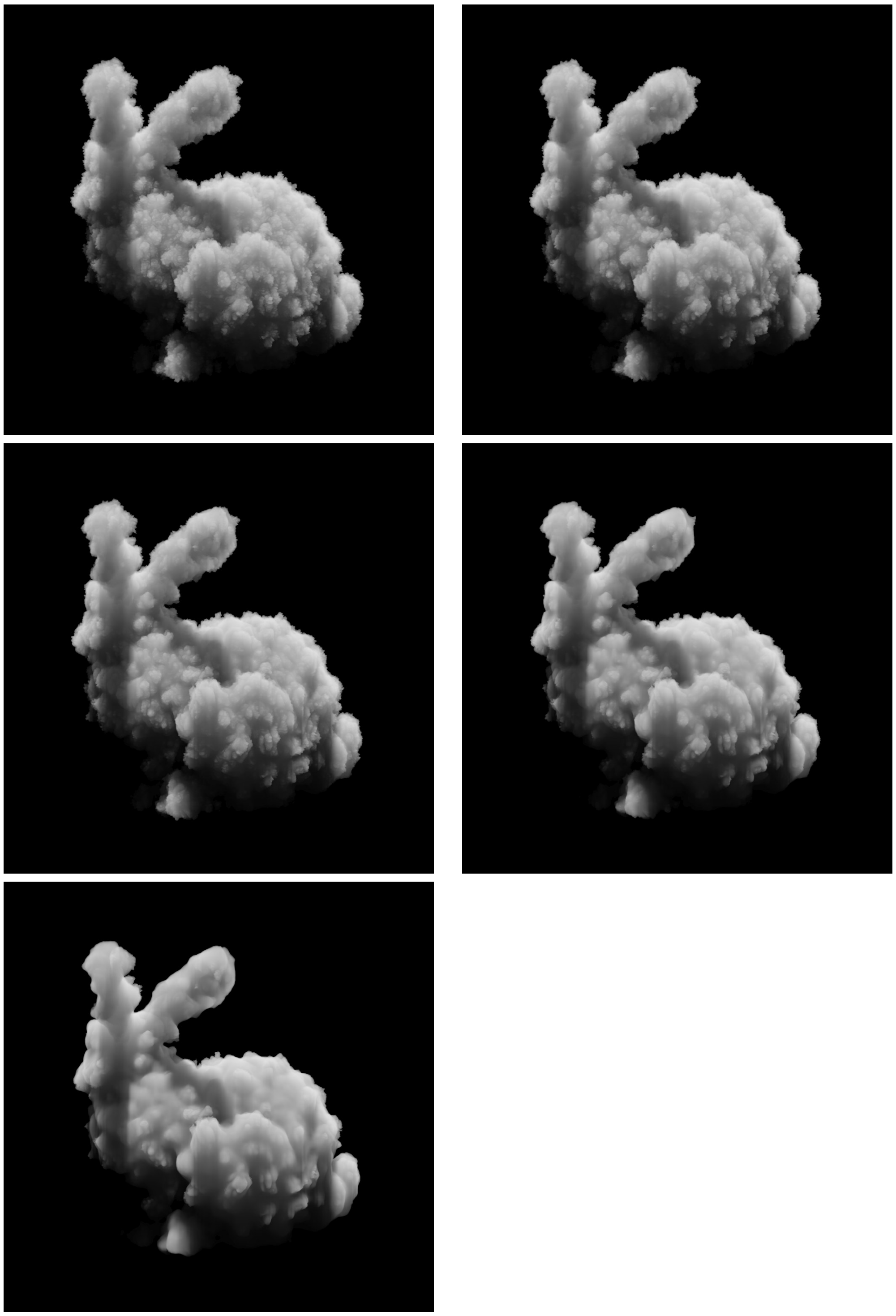


Figure 3.5: Volume renders with various values of billow. Left to right, top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

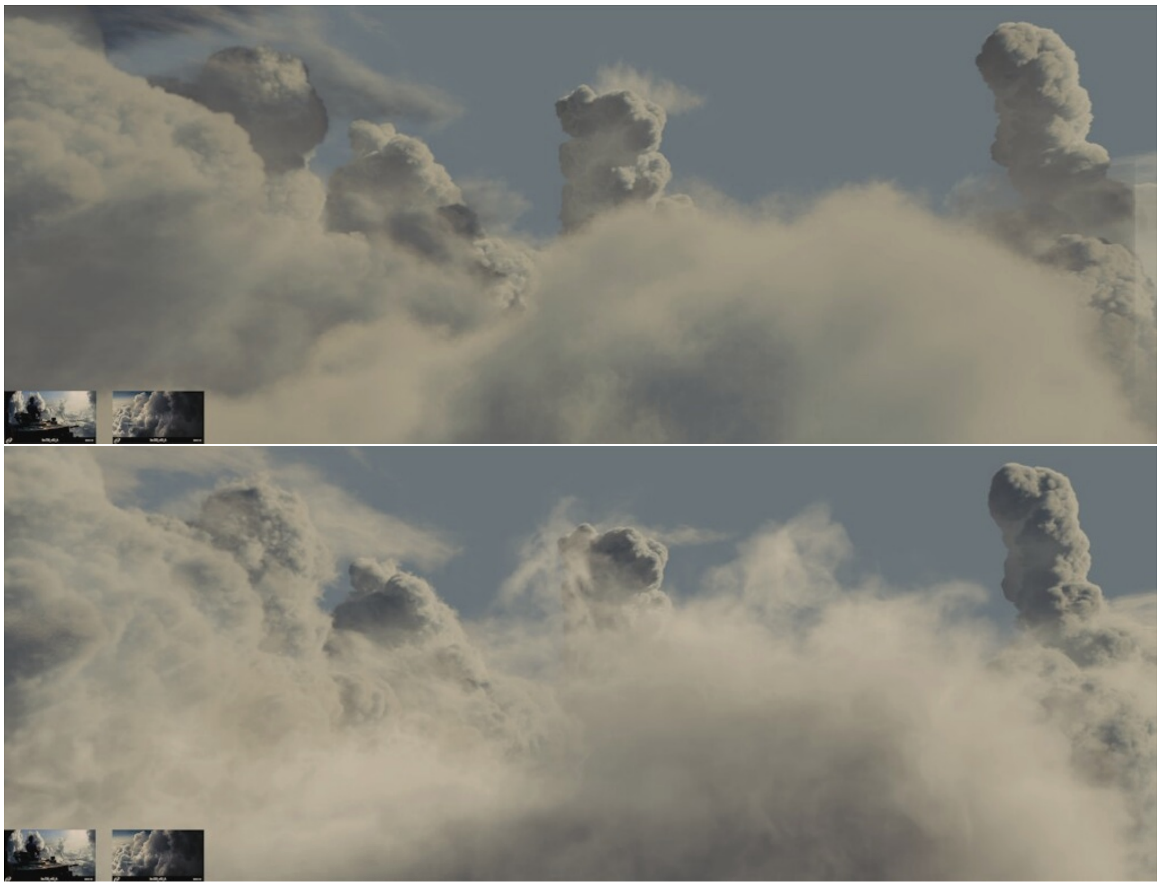


Figure 3.6: Clouds rendered for the film *The A-Team* using gridless advection to make their edges more realistic. Top: foreground clouds without advection; bottom: foreground clouds after gridless advection.

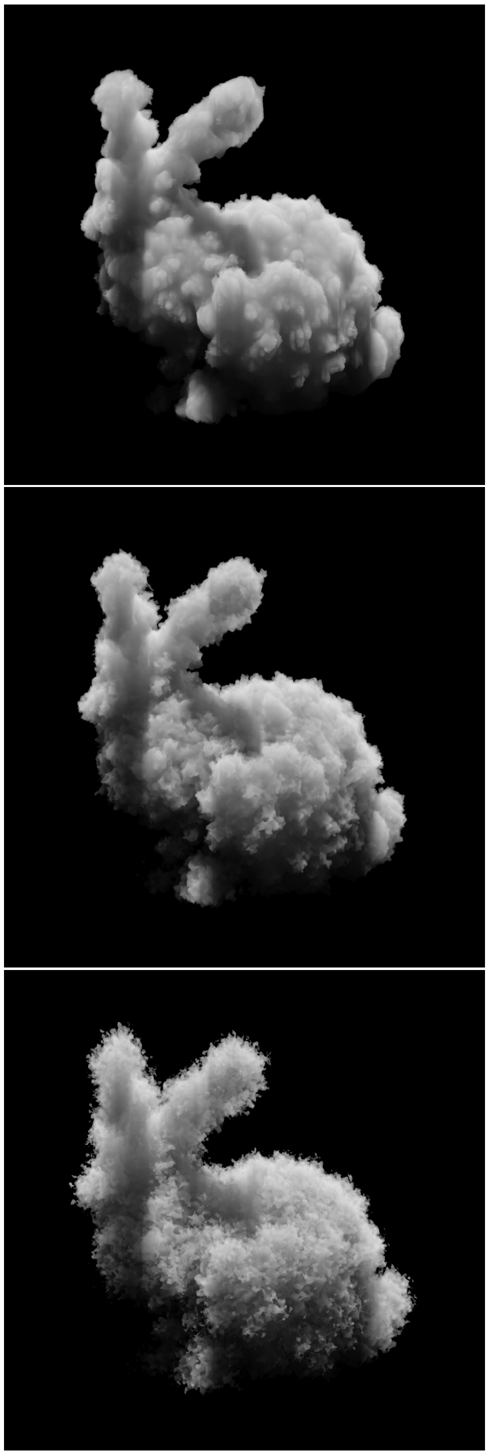


Figure 3.7: Volume renders with various setting of advection, for billow=1. Top to bottom: No advection, medium advection, strong advection.

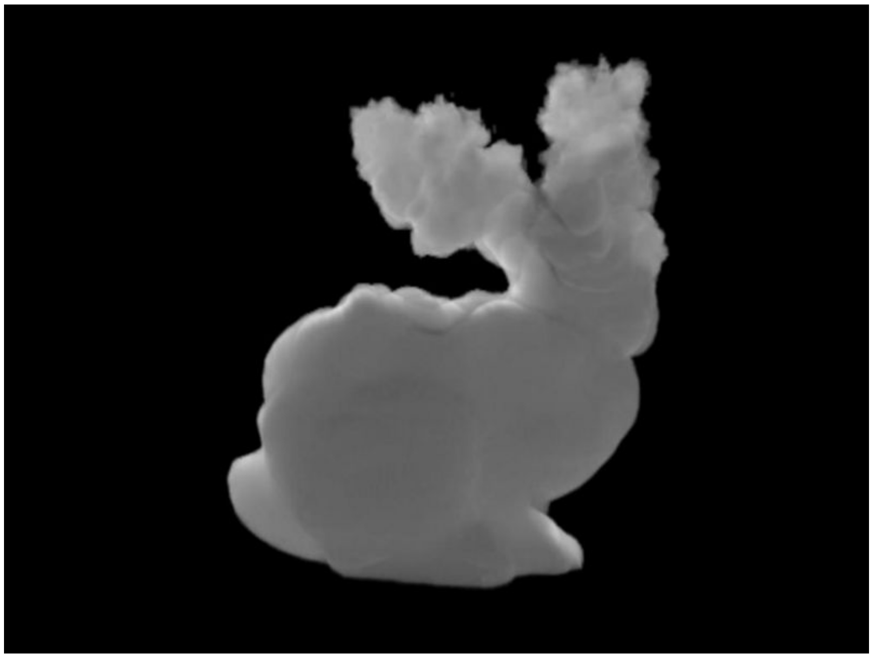
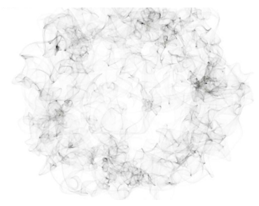


Figure 3.8: Volumetric bunny with spatial control over the pyroclastic displacement.

attributes attached to the base geometry of the cloud shape. The values of each of the parameters are encoded in the point attributes. Simple fields of these attribute values are created by adding a spherical volume of the attribute value to a gridded cache enclosing the cloud. This allows simple control based on surface properties.



Chapter 4

Warping Fields

Here we explore a procedure for transfering attributes from one shape to another. This problem is not volumetric per se, but a very nice solution involving levelsets is presented here.

Suppose you have a complex geometric object with vertices $\mathbf{x}_i^O,\ i = 1, \ldots N^O$ on its surface. For rendering or other purposes you would like to have a variety of attribute values attached to each vertex, but because of its complexity, building a smooth distribution of values by hand is a tedious process. A controllable method to generate values would be handy. As an input, suppose that there is a reference shape with vertices $\mathbf{x}_i^r,\ i = 1, \ldots N^r$ and attribute values already mapped across its surface. The goal then is to find a way to transfer the attributes from the reference surface to the object surface, even if the two surfaces have wildly different topology. The approach we illustrate here generates a smooth function $\mathbf{X}(\mathbf{x})$ which warps the reference shape into the object shape. However, this is not a map from the vertices of the reference to the vertices of the object, but a mapping between the levelset representations of the two surfaces. This Nacelle algorithm (it generates warp fields) works well even when the topology of the two shapes is very different. In the next section the mathematical formulation of the algorithm is shown, and after that a short Felt script for it.

4.1 Nacelle Algorithm

The algorithm assumes that the two shapes involved can be converted into levelset representations. This means that there are two levelsets, one for the reference shape $L_r(\mathbf{x})$ and one for the object shape $L_O(\mathbf{x})$. These two levelsets are signed distance functions that are smooth (i.e. C^2). The nacelle algorithm postulates that there is a warping function $\mathbf{X}(\mathbf{x})$ which maps between the two levelsets:

$$L_O(\mathbf{x}) = L_r(\mathbf{X}(\mathbf{x})) \tag{4.1}$$

The goal of the algorithm is an iterative procedure for approximating the field $\mathbf{X}$. Each iteration generates the approximate warping field $\mathbf{X}_n(\mathbf{x})$. The natural choice for the initial field is $\mathbf{X}_0(\mathbf{x}) = \mathbf{x}$.

Given the warp field $\mathbf{X}_n$ from the n-th iteration, we compute the (n+1)-th approximation by looking at an error term $\mathbf{u}(\mathbf{x})$ with $\mathbf{X} = \mathbf{X}_n + \mathbf{u}$. Putting this into the equation 4.1 gives

$$L_O(\mathbf{x}) = L_r(\mathbf{X}_n(\mathbf{x}) + \mathbf{u}(\mathbf{x})) \tag{4.2}$$

Expanding this to quadratic order in Taylor series gives

$$L_O(\mathbf{x}) - L_r\left(\mathbf{X}_n(\mathbf{x})\right) = \mathbf{u}(\mathbf{x}) \cdot \nabla L_r(\mathbf{X}_n(\mathbf{x})) + \frac{1}{2}\sum_{ij} u_i(\mathbf{x}) u_j(\mathbf{x}) \frac{\partial^2}{\partial x_i \partial x_j} L_r(\mathbf{X}_n(\mathbf{x})) \tag{4.3}$$

Define matrix $\mathbf{M}$ as

$$M_{ij}(\mathbf{x}) = \frac{\partial^2}{\partial x_i \partial x_j} L_r(\mathbf{x}) \tag{4.4}$$

so the Taylor expansion up to quadratic is

$$L_O(\mathbf{x}) - L_r(\mathbf{X}_n(\mathbf{x})) = \mathbf{u}(\mathbf{x}) \cdot \nabla L_r(\mathbf{X}_n(\mathbf{x})) + \frac{1}{2}\mathbf{u}(\mathbf{x}) \cdot \mathbf{M}(\mathbf{X}_n(\mathbf{x})) \cdot \mathbf{u}(\mathbf{x}) \tag{4.5}$$

Setting $\mathbf{u}(\mathbf{x}) = A(\mathbf{x})\ \nabla L_r(\mathbf{X}_n)$, we get the quadratic equation for the scalar field $A(\mathbf{x})$

$$\frac{L_O(\mathbf{x}) - L_r(\mathbf{X}_n(\mathbf{x}))}{\left|\nabla L_r(\mathbf{X}_n(\mathbf{x}))\right|^2} = A(\mathbf{x}) + \frac{1}{2}A^2(\mathbf{x}) \frac{\nabla L_r(\mathbf{X}_n(\mathbf{x})) \cdot \mathbf{M}(\mathbf{X}_n(\mathbf{x})) \cdot \nabla L_r(\mathbf{X}_n(\mathbf{x}))}{\left|\nabla L_r(\mathbf{X}_n(\mathbf{x}))\right|^2} \tag{4.6}$$

which has the solution

$$A(\mathbf{x}) = \frac{1}{\Gamma}\left\{-1 + \left[1 + 2\Delta\Gamma\right]^{1/2}\right\} \tag{4.7}$$

with the abbreviations

$$\Delta = \frac{L_O(\mathbf{x}) - L_r(\mathbf{X}_n(\mathbf{x}))}{\left|\nabla L_r(\mathbf{X}_n(\mathbf{x}))\right|^2} \tag{4.8}$$

$$\Gamma = \frac{\nabla L_r(\mathbf{X}_n(\mathbf{x})) \cdot \mathbf{M}(\mathbf{X}_n(\mathbf{x})) \cdot \nabla L_r(\mathbf{X}_n(\mathbf{x}))}{\left|\nabla L_r(\mathbf{X}_n(\mathbf{x}))\right|^2} \tag{4.9}$$

Then the next approximation is

$$\mathbf{X}_{n+1}(\mathbf{x}) \ = \ \mathbf{X}_n(\mathbf{x}) \ + \ A(\mathbf{x})\ \nabla L_r(\mathbf{X}_n) \tag{4.10}$$

In practice, this scheme converges in 1-3 iterations even for complex warps and topology differences.

4.2 Numerical implementation

Numerical implementation of the nacelle algorithm requires code for equations 4.7 – 4.10. These four equations are implemented in the following six lines (plus comments) of FELT script:

```
// Definitions
vectorfield B = compose(grad(Lr), Xn);
matrixfield M = compose(grad(grad(L1)), Xn);
// Equation 4.8
scalarfield del = (Lo - compose(Lr, Xn))/(B*B);
// Equation 4.9
scalarfield Gamma = (B*M*B)/(B*B);
// Equation 4.7
scalarfield A = (scalarfield(-1) + (scalarfield(1) + 2.0*del*Gamma)^0.5)/Gamma;
// Equation 4.10
vectorfield Xnplus1 = Xn + A*B;
```

The compose function evaluates the field in the first argument at the location of the vectorfield in the second argument.

There are ways to speed up this implementation, at the cost of some accuracy. For example, the quantities B*B and B*M*B are scalarfields that are computationally expensive. Significant speed improvements come from sampling them into grids and using the gridded scalarfields in their place. The modified FELT script to accomplish that is

```
// Definitions
vectorfield B = compose(grad(Lr), Xn);
matrixfield M = compose(grad(grad(L1)), Xn);
// ============ NEW CODE =====================
// Create scalar caches over some domain "dom"
scalarcache BBc( dom );
scalarcache BMBc( dom );
// Sample B*B and B*M*B onto grids
cachewrite(BBc, B*B);
cachewrite(BMBc, B*M*B);
// Replace fields with gridded versions
scalarfield BB = cacheread(BBc);
scalarfield BMB = cacheread(BMBc);
// ============ END NEW CODE =================
// Equation 4.8
scalarfield del = (Lo - compose(Lr, Xn))/BB;
// Equation 4.9
scalarfield Gamma = BMB/BB;
// Equation 4.7
scalarfield A = (scalarfield(-1) + (scalarfield(1) + 2.0*del*Gamma)^0.5)/Gamma;
```

```
// Equation 4.10
vectorfield Xnplus1 = Xn + A*B;
```

4.3 Attribute transfer

The mapping function $\mathbf{X}(\mathbf{x})$ allows us to do a number of things:

Warp Levelsets

The object levelset is now approximated by $L_r(\mathbf{X}(\mathbf{x}))$. For example, figure 4.1(a) shows a complex object shape consisting of two linked torii and a cone, with the cone intersecting one of the torii. The reference shape in figure 4.1(b) is a sphere. Both of these shapes have levelset representations, so that the mapping function can be generated. After one iteration, the levelset field $L_r(\mathbf{X}_1(\mathbf{x}))$ was used to generate the geometry shown in figure 4.1(c), which is essentially identical to the input object shape. In testing with other complex shapes, no more than five iterations has ever been needed to get highly accurate convergence of algorithm.

Attribute transfer

The mapping function provides a method to perform attribute transfer from the reference shape to the object shape. Using the vertices $\mathbf{x}_i^O$, $i = 1, \ldots N^O$ on the surface of the object shape, the corresponding mapped points

$$\mathbf{x}_i^M \equiv \mathbf{X}(\mathbf{x}_i^O) \tag{4.11}$$

are points that lie on the surface of the reference shape. Assuming the reference shape has attributes attached to its vertices, and a method of interpolating the attributes to points on the surface between the vertices, the reference shape attributes can be sampled at the locations $\mathbf{x}_i^M$, $i = 1, \ldots N^O$ and assigned to the corresponding vertices on the object shape. Figure 4.2 shows the object shape with a texture pattern mapped onto it. The texture coordinates were transfered from the reference shape.

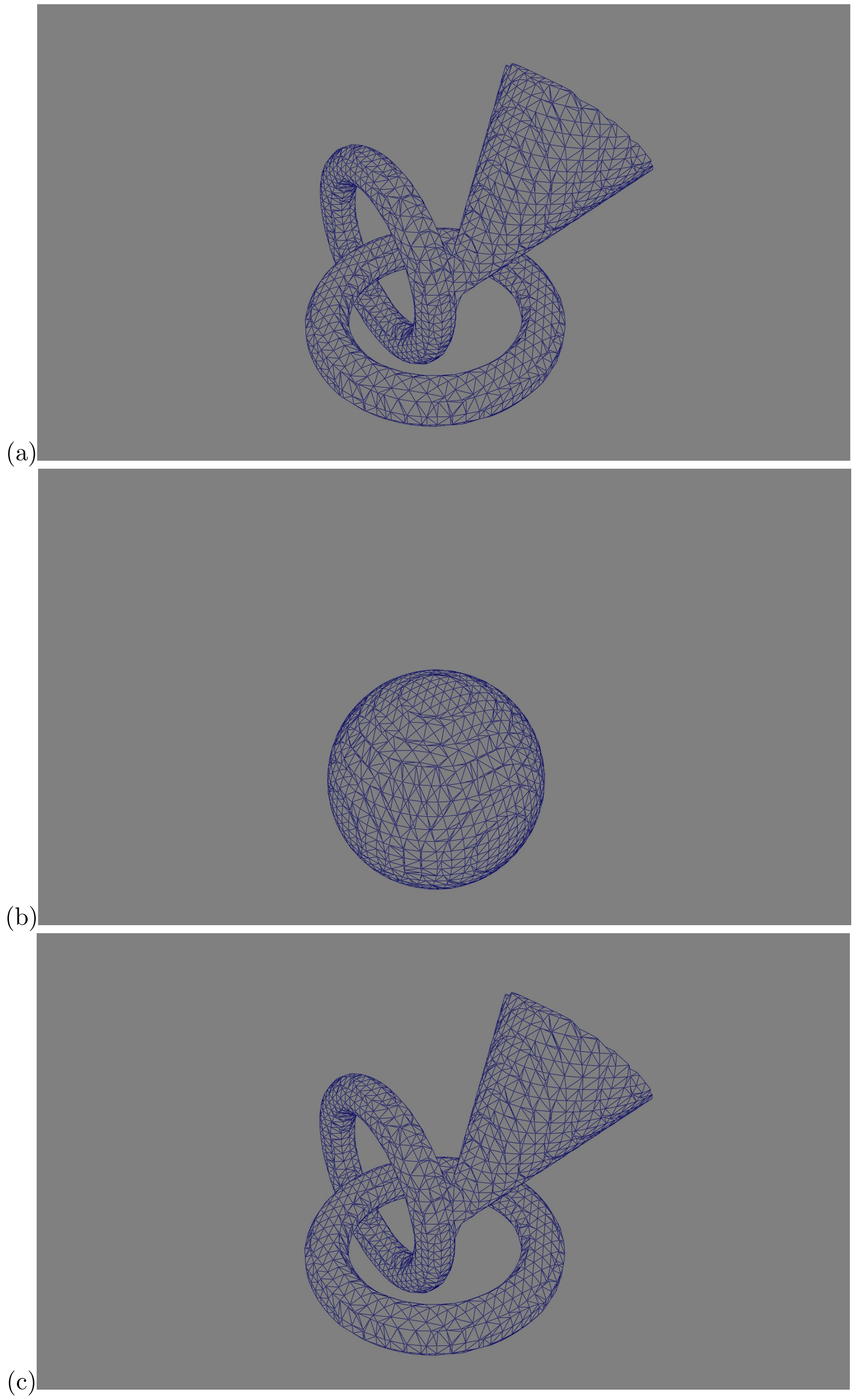


Figure 4.1: Warping of a reference sphere into a complex shape (cone and two torii). (a) Object shape; (b) Reference sphere; (c) Warp shape output from 1 iteration.

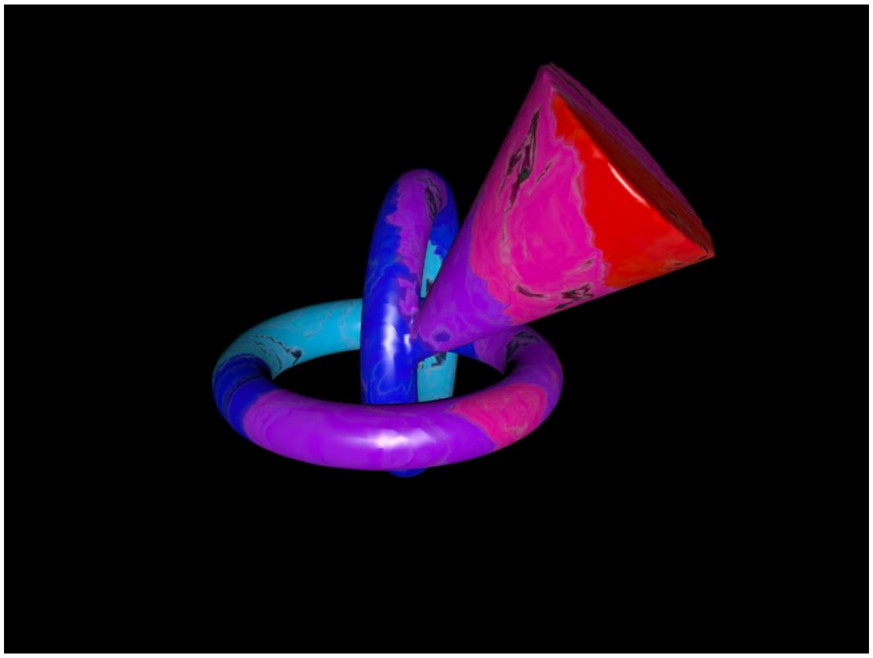


Figure 4.2: Texture mapping of the object shape by transfering texture coordinates from the reference shape.

Chapter 5

Cutting Up Models

Levelsets and implicit functions in general are particularly excellent, powerful tools for cutting up geometry into many pieces. This is very useful for models of fracture, surgery, and explosions. The technique was shown in film application by Museth[3]. Here we introduce the theory in steps by modeling knives in terms of implicit functions, then cut geometry with a single knife, two knives, and arbitrarily many knives.

The essential reason that implicit function based cutting works is that implicit functions separate the world into two (non-contiguous) regions: those for which the implicit function knife is positive, and those for which the implict function knife is negative. Cutting takes place by separating the geometry into the parts that correspond to those two regions. To do this, the geometry must be represented by a levelset, so we assume that has already been done and it is called $\ell_0(\mathbf{x})$.

5.1 Levelset knives

A knife for our purposes is simply a levelset or implicit function. It can be procedural or grid-based. The essential feature is that, within the volume of the geometry you wish to cut, the knife has both positive and negative regions. The zero-value surface(s) of the knife are the knife-edge, or boundary between the cuts in the geometry.

For example, a simple straight edge is the signed distance function of a flat plane:

$$K_{straight\ edge}(\mathbf{x}) \ = \ (\mathbf{x} - \mathbf{x}_P) \cdot \mathbf{n} \tag{5.1}$$

for a plane with normal $\mathbf{n}$ and $\mathbf{x}_P$ on the surface of the plane.

5.2 Single cut

A knife $K(\mathbf{x})$ separates the geometry $\ell_0(\mathbf{x})$ into two regions. Because we are using levelsets, the feature that distinquishes the two regions is their signs: positive in one region, negative in the other. Note that the product function

$$F(\mathbf{x}) \;=\; \ell_0(\mathbf{x})\; K(\mathbf{x}) \tag{5.2}$$

has positive and negative regions, but does not quite sort the regions the way we would like. This product actually defines four regions:

1. $\ell_0 > 0$ and $K > 0$
2. $\ell_0 < 0$ and $K < 0$
3. $\ell_0 < 0$ and $K > 0$
4. $\ell_0 > 0$ and $K < 0$

and lumps together regions 1 and 2, and regions 3 and 4. What we actually want for a successful cut is to get only regions inside the geometry, separated into the two sides of the knife.

A useful tool in building this is the mask function, which is essentially a Heaviside step function for scalarfields. For a scalar field f, the mask is a field with the value of 0 or 1:

$$\mathsf{mask(f)}(\mathbf{x}) = \begin{cases} 1 & f(\mathbf{x}) \geq 0 \\ 0 & f(\mathbf{x}) < 0 \end{cases} \tag{5.3}$$

With the mask function, we can build two fields that identify the inside and outside of the levelset geometry l0:

```
scalarfield inside = mask( l0 );
scalarfield outside = scalarfield(1.0) - mask( l0 );
```

The next thing to realize is that we only want the knife to cut the levelset inside the geometry: there is no need to cut when outside the geometry. A good way to accomplish this is by the product of the scalarfield for the knife and the inside function:

```
scalarfield insideKnife = inside * knife;
```

Now we need to generate a levelset function that is unaffected by the knife outside of the geometry, but is cut by the knife inside. This scalarfield does that:

```
scalarfield cutInside = ( outside + inside*knife ) * l0;
```

Outside of the geometry, this field has the value of the levelset l0. Inside the geometry, it has the value of knife*l0. So when interpreted as a levelset, this field identifies the part of the geometry that is also inside the knife, i.e. the positive regions of the knife. The complementary field

```
scalarfield cutOutside = ( outside - inside*knife ) * l0;
```

similary generates geometry that is inside the original and outside of the knife. So cutInside and cutOutside are the two regions of the original geometry that you get when you cut it with the knife. You can then recover the geometry of the cut shapes by converting the levelset functions back into geometry:

```
shape cutInsideShape = ls2shape( cutInside );
shape cutOutsideShape = ls2shape( cutOutside );
```

You should recognize that the two geometric structures, cutInsideShape and cutOutsideShape are not necessarily simple, connected shapes. Depending on the structure of the original geometry, and the shape and positioning of the knife function, each output shape may have many disconnected portions, or even be empty.

5.3 Multiple cuts

Suppose we want to cut geometry with more than one knife. The process is an iteration: the cut with the first knife produces the two levelsets cutInsideShape and cutOutsideShape. Then cut each of those with the second knife, producing two for each of those, for a total of four levelsets . Each cut doubles the number of levelsets, so for N knives, you generate 2^N levelsets, each for a collection of pieces. Figure 5.1 shows the result of cutting a sphere with 5 flat blades, with the orientation and location of each knife randomly chosen. While 5 blades produce $2^5 = 32$ levelsets, the output actually contains only 22 actual pieces. Some of levelsets are empty of geometry.

The question might arise as to whether the results depend on the order in which knives are applied. Mathematically, the results are identical no matter what order is used.

For computational efficiency however, it could be useful to examine the output of each cut to see if there are levelsets that are actually empty of pieces of the geometry. If empty levelsets are found, they can be discarded from further cutting, possibly improving speed and memory usage. In this context of efficiency, the order in which knives are applied may impact the performance.

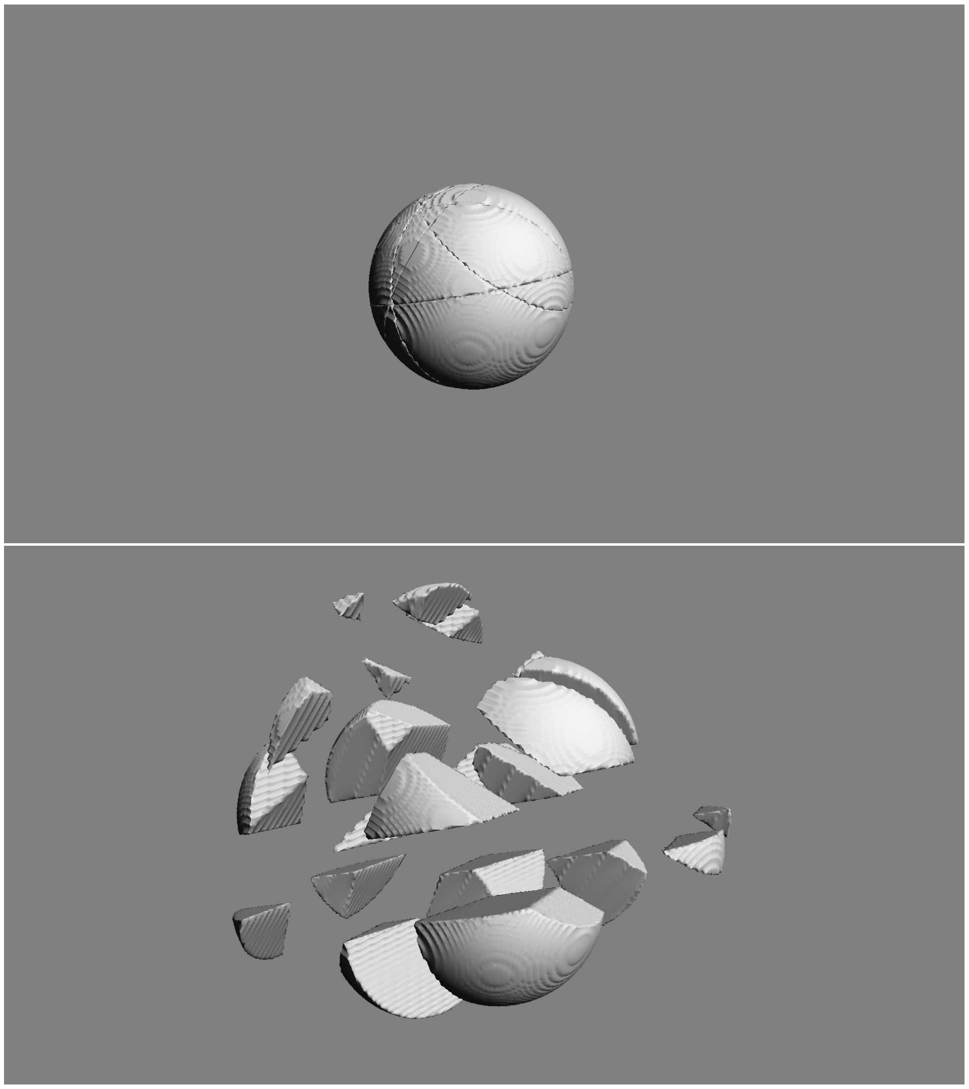


Figure 5.1: A sphere carved into 22 pieces using 5 randomly placed and oriented flat blades. The top shows the sphere with the cuts visible. The bottom is an expanded view of the pieces.

Chapter 6

Fluid Dynamics

Fluid dynamics is generally associated with high performance computing, even in graphics applications. Solving the Navier-Stokes equations for incompressible flow is no small task, and computationally expensive. There are a variety of solution methodologies, which produce visually different flows. The stability of the various methodologies also varies widely. The two solution methods known as Semi-Lagrangian advection and FLIP advection are unconditionally stable, and so are very desireable approaches for some graphics-oriented simulation problems. QUICK is conditionally stable, but has minimal numerical viscosity and even for small grids generates remarkably detailed flow patterns that persist and are desireable for some graphics simulation problems as well.

In terms of volumetric scripting, it is possible to create simple scripts that efficiently solve the incompressible Navier-Stokes equations. Additionally, the ability to choose when and where to represent a field as gridded data or not can have a significant impact on the character of the simulation. In this chapter we look at simple solution methods, based on Semi-Lagrangian advection and generalizations, and introduce the concept of gridless advection. The next chapter examines gridless advection in more detail.

6.1 Navier-Stokes solvers

The basic simulation situation we look at in this chapter is the flow of a bouyant gas. The gas has a velocity field $\mathbf{u}(\mathbf{x}, t)$ which initially we set to 0. The density of the gas $\rho(\mathbf{x}, t)$ is lighter than the surrounding static medium, and so there is a gravitational force upward proportional to the density. The equations of motion are

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \rho = S(\mathbf{x}, t) \tag{6.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p = -\mathbf{g}\, \rho \tag{6.2}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{6.3}$$

A Semi-Lagrangian style of solver for this problem splits the problem into multiple steps:

1. Advect the density with the current velocity

$$\rho(\mathbf{x}, t + \Delta t) = \rho(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\ \Delta t, t) + S(\mathbf{x}, t)\ \Delta t \tag{6.4}$$

2. Advect the velocity and add external forces

$$\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t)\ \Delta t, t) - \mathbf{g}\ \rho(\mathbf{x}, t + \Delta t)\ \Delta t \tag{6.5}$$

3. Project out the divergent part of the velocity, using FFTs, conjugate gradient, or multigrid algorithms

These steps can be reproduced in a FELT script as the following:

```
// Step 1, equation 6.4
density = advect( density, velocity, dt );
// Write density to cache
cachewrite( density Cache, density );
// Set density to the value in the cache
density = cacheread( density Cache );
// Step 2, equation 6.5
velocity = advect( velocity, velocity, dt ) - dt*gravity*density ;
// Step 3, fftdivfree uses FFTs to remove the divergent part of the field
velocity = fftdivfree( velocity, region );
```

The function advect evaluates the first argument at a position displaced by the velocity field (the second argument) and time step dt (the third argument). There is no need to explicitly write the velocity field to a cache after its self-advection because the function fftdivfree returns a velocity field that has been sampled onto a grid.

6.1.1 Hot and Cold simulation scenario

A variation on the bouyant flow scenario is shown in figure 6.1. There are two density fields, one for hot gas with a red color, and one for cold gas with a blue color. The cold gas falls from the top, and the hot gas rises from the bottom. Both are continually fed new density at their point of origin. The two gases collide in the center and displace each other as shown. The FELT script is

```
hot = advect( hot, velocity, dt ) + inject(hotpoint, dt );
// Write hot density to cache
scalarcache hotCache(region);
cachewrite( hotCache, hot );
// Set hot density to the value in the cache
hot = cacheread( hotCache );
```

```
cold = advect( cold, velocity, dt ) + inject(coldpoint, dt);
// Write cold density to cache
scalarcache coldCache(region);
cachewrite( coldCache, cold );
// Set cold density to the value in the cache
cold = cacheread( coldCache );

velocity = advect( velocity, velocity, dt ) + dt*gravity*(cold-hot);
// fftdivfree uses FFTs to remove the divergent part of the field
velocity = fftdivfree( velocity, region );
```

The two densities force the velocity in opposite directions (hot rises, cold sinks). We have also added a continuous injection of new density via the user-defined function inject, defined to insert a solid sphere of density at a location specified by the first argument:

```
func scalarfield inject( vector center, float dt )
{
    vectorfield spherecenter = identity() - vectorfield(center);
   // Implicit function of a unit sphere centered at the input location
   scalarfield sphere = scalarfield(1.0) - spherecenter*spherecenter;
   // mask() function returns 0 outside implicit function, 1 inside
   scalarfield inject = mask(sphere);
   return inject*dt;
}
```

The advection process used for this simulation example is Semi-Lagrangian advection, which is highly dissipative because of the linear interpolation process. As figure 6.2 shows, the simulation produces a diffusive looking mix of the two gases. A simulation with higher spatial resolution would produce a different spatial structure with more of a sense of vortical motion and finer detail, but still not avoid the diffusive mixing.

6.2 Removing the grids

The power of resolution independent scripting provides a new option, gridless advection, which we introduce here and expand on in the next chapter. Because of the procedural aspects of resolution independence, we can rebuild the script for the hot/cold simulation, and remove the sampling of the densities onto grids. Removing those steps, you are left with the code:

```
hot = advect( hot, velocity, dt )  + inject(hotpoint, dt );
cold = advect( cold, velocity, dt ) + inject(coldpoint, dt);
velocity = advect( velocity, velocity, dt ) + dt*gravity*(cold-hot);
// fftdivfree uses FFTs to remove the divergent part of the field
velocity = fftdivfree( velocity, region );
```

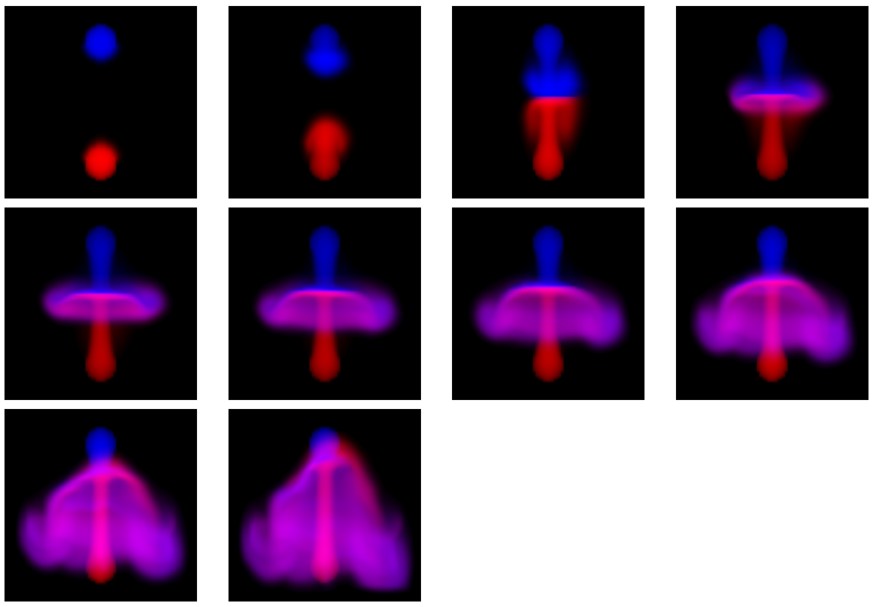


Figure 6.1: Simulation sequence for hot and cold gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

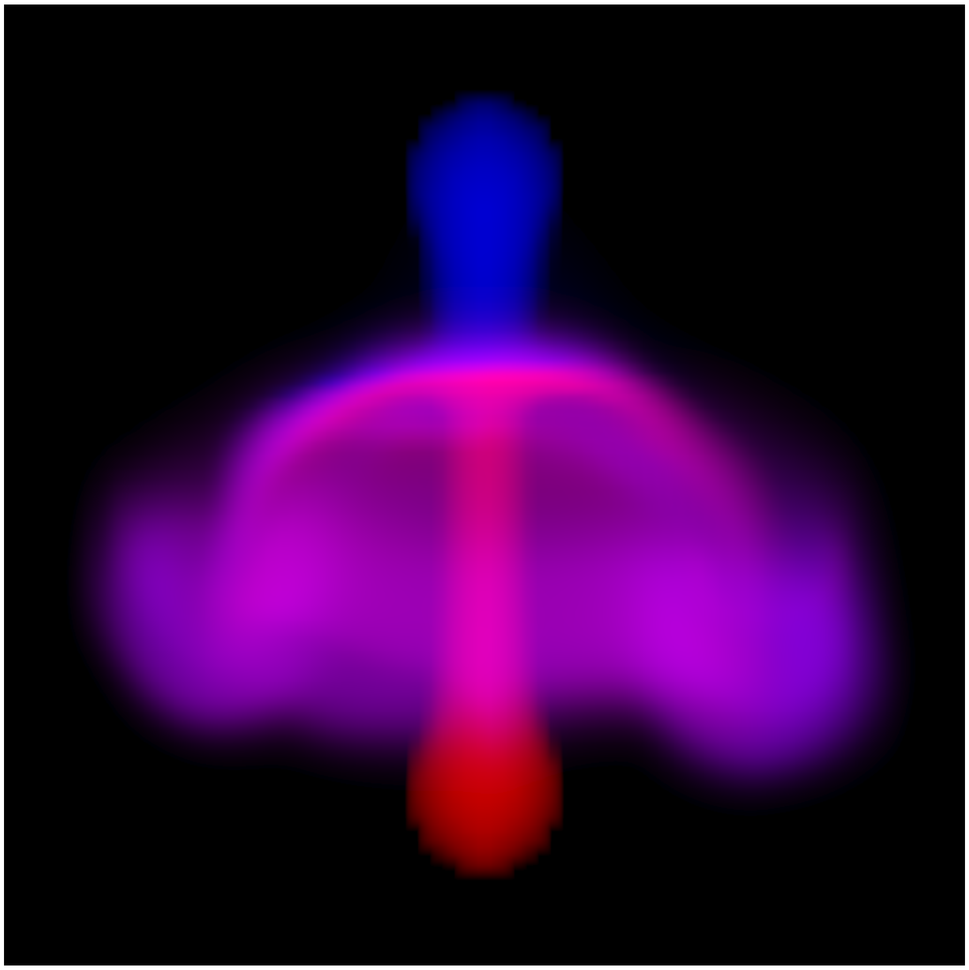


Figure 6.2: Frame of simulation of two gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

What happens here is that the evolution of the densities over multiple time steps is evaluated in a purely procedural processing chain. The history of velocity fields is implicitly retained and applied to advect the density through a series of points along a path through the volume. This path-track happens every time the value of the densities at the current frame are requested (e.g. by the volume renderer or some other processing). The velocity continues to be sampled onto a grid because the computation to remove the divergent portion of the field requires sampling the velocity onto a grid. All of the existing algorithms for removing divergence require a gridded sampling of the velocity, so there is presently no method to avoid grids for the velocity field in this situation. However, the densities in this simulation are never sampled onto a grid.

The hot/cold simulation produced by removing the gridding of the density is shown in figure 6.3, with a frame shown larger in figure 6.4. The spatial details and motion timing are dramatically different, as seen in a side-by-side comparison in figure 6.5. Symmetries in the simulation scenario are better preserved in the gridless implementation, and the fingers of the flow contain more vorticity (though not as much as possible, because gridding of the velocity field continues to dissipate vorticity) and fine filaments and sheets.

The downside of this simulation approach is that the memory grows linearly with the number of frames, and the time spent evaluating the density grows linearly with the number of frames. So there is a tradeoff to consider between achieving fine detail vs computational resources. This is also a tradeoff that must be addressed in traditional high performance simulation, but the trends in the tradeoff are different: computational cost is essentially constant per frame in traditional simulation, whereas gridless advection cost grows linearly per frame. But traditional simulation has visual detail limited by the resolution of the grid(s), and gridless advection generates much finer detail.

6.3 Boundary Conditions

In addition to free-flowing fluids, FELT scripting can also handle objects in a simulation that obstruct the flow of the fluid. This is handled very simply by reflecting the velocity about the normal of the object. Any objects can be represented as a levelset, $O(\mathbf{x})$, which we will take to be negative outside of the object and positive inside. At the boundary and the interior of the object, if the velocity of the fluid points inward it should be reflected back outward. The outward pointing normal of the object is $-\nabla O$, so the velocity should be unchanged (1) at points outside the object ($O(\mathbf{x})$ is negative), and (2) if the component of velocity at the object is outward flowing (i.e. $\mathbf{u} \cdot \nabla O < 0$). The mask() function in FELT provides the switching mechanism for testing and acting on these conditions. When the flow has to be reflected, the new velocity is

$$\mathbf{u}_{reflected} = \mathbf{u} \; - \; 2\frac{(\mathbf{u} \cdot \nabla O)}{|\nabla O|^2} \; \nabla O \qquad (6.6)$$

The FELT code for this is

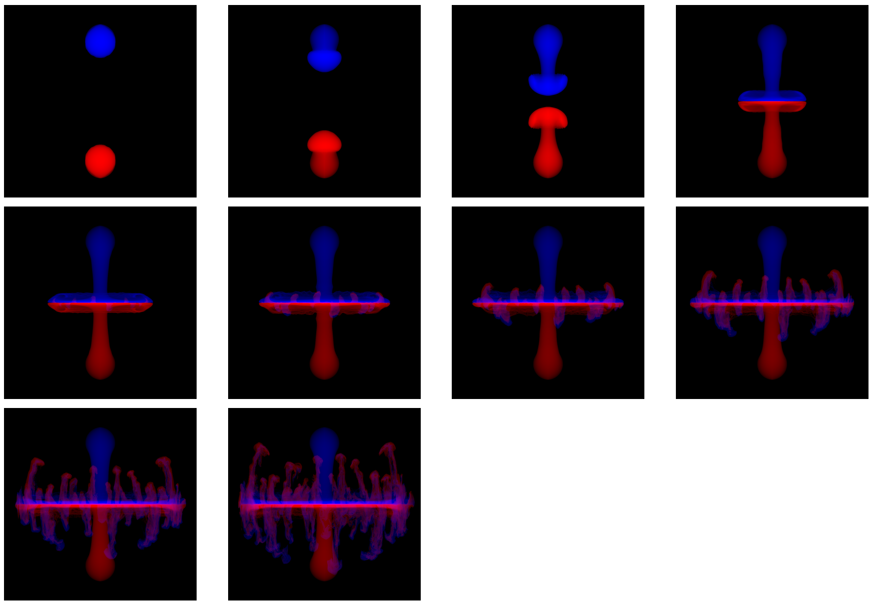


Figure 6.3: Sequence of frames of a simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected but not sampled onto a grid, i.e. gridlessly advected in a procedural simulation process. The grid resolution for velocity is $50 \times 50 \times 50$.

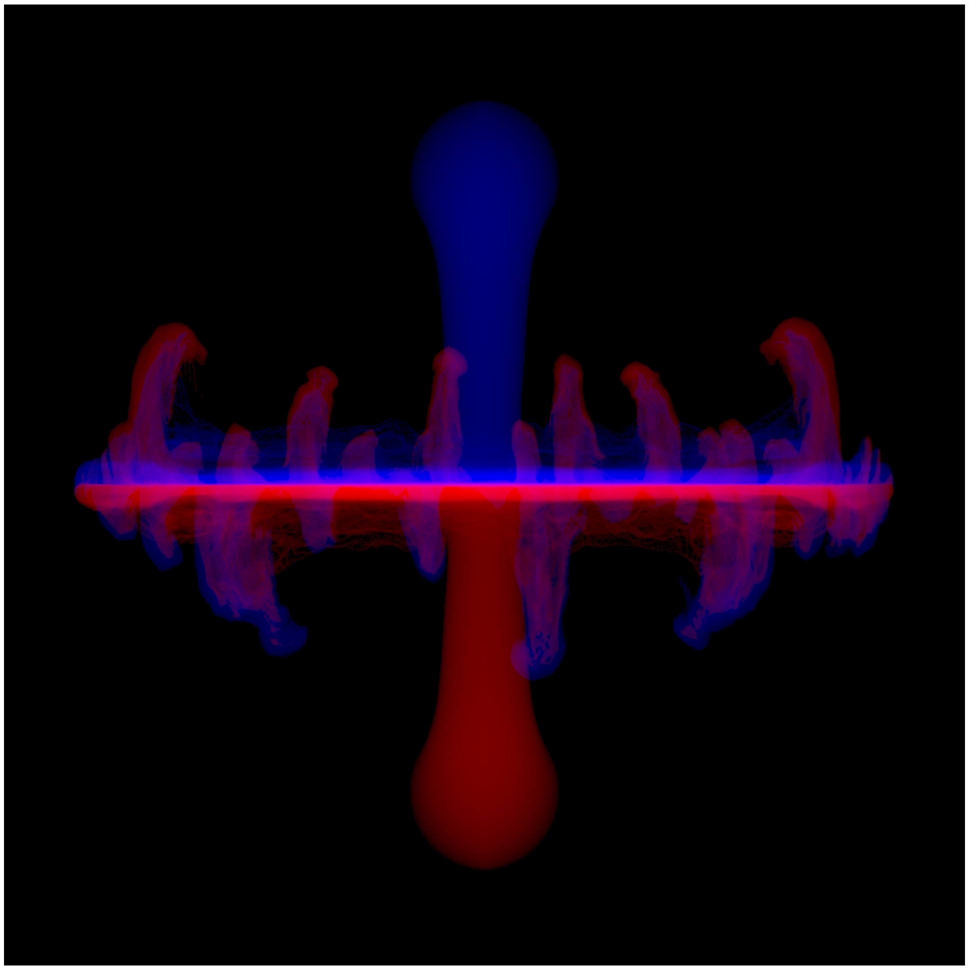


Figure 6.4: Frame of simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected but not sampled onto a grid, i.e. gridlessly advected in a procedural simulation process. The grid resolution for velocity is $50 \times 50 \times 50$.

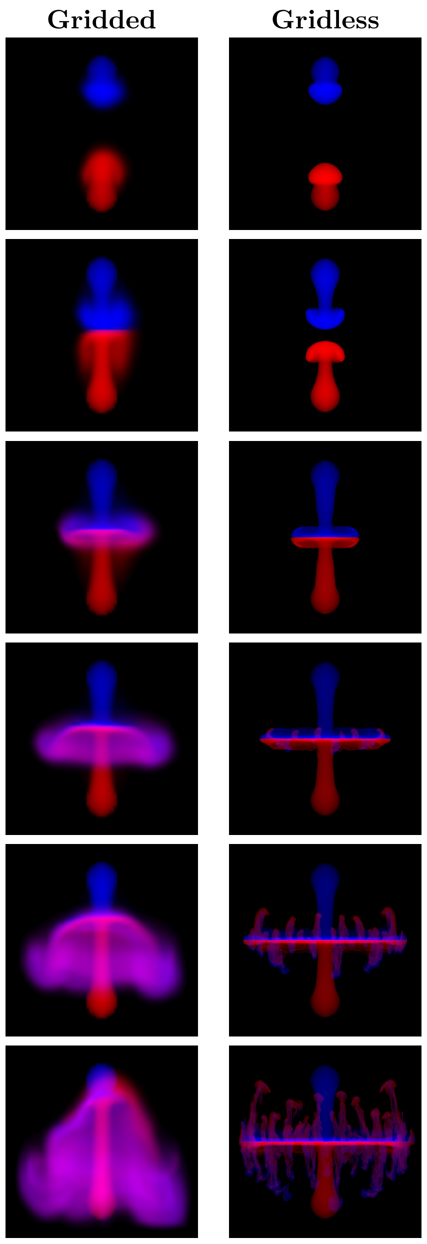


Figure 6.5: Simulation sequences with density gridded (left) and gridless (right). The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution is $50 \times 50 \times 50$.

```
vectorfield normal = -grad(object)/sqrt( grad(object)*grad(object) );
scalarfield normalU = velocity*normal;
velocity -= mask(normalU)*mask(object)*2.0*normalU*normal;
```

To illustrate the effect, figure 6.6 shows a sequence of frames from a simulation in which a bouyant gas is confined inside a box, and encounters a rectangular slab that it must flow around. To capture detail, the density was handled with gridless advection. The slab diverts the flow downward, where the density thins as it spreads, and the bouyancy force weakens because of the thinner density. The slab also generated vortices in the flow that persist for the entire simulation time.

This volume logic is suitable to impose other boundary conditions as well. For example, sticky boundaries reflect only a fraction of the velocity

$$\mathbf{u}_{sticky} = \mathbf{u} \; - \; (1+\alpha)\frac{(\mathbf{u}\cdot\nabla O)}{|\nabla O|^2}\;\nabla O \tag{6.7}$$

with $0 \leq \alpha \leq 1$ being the fraction of velocity retained.

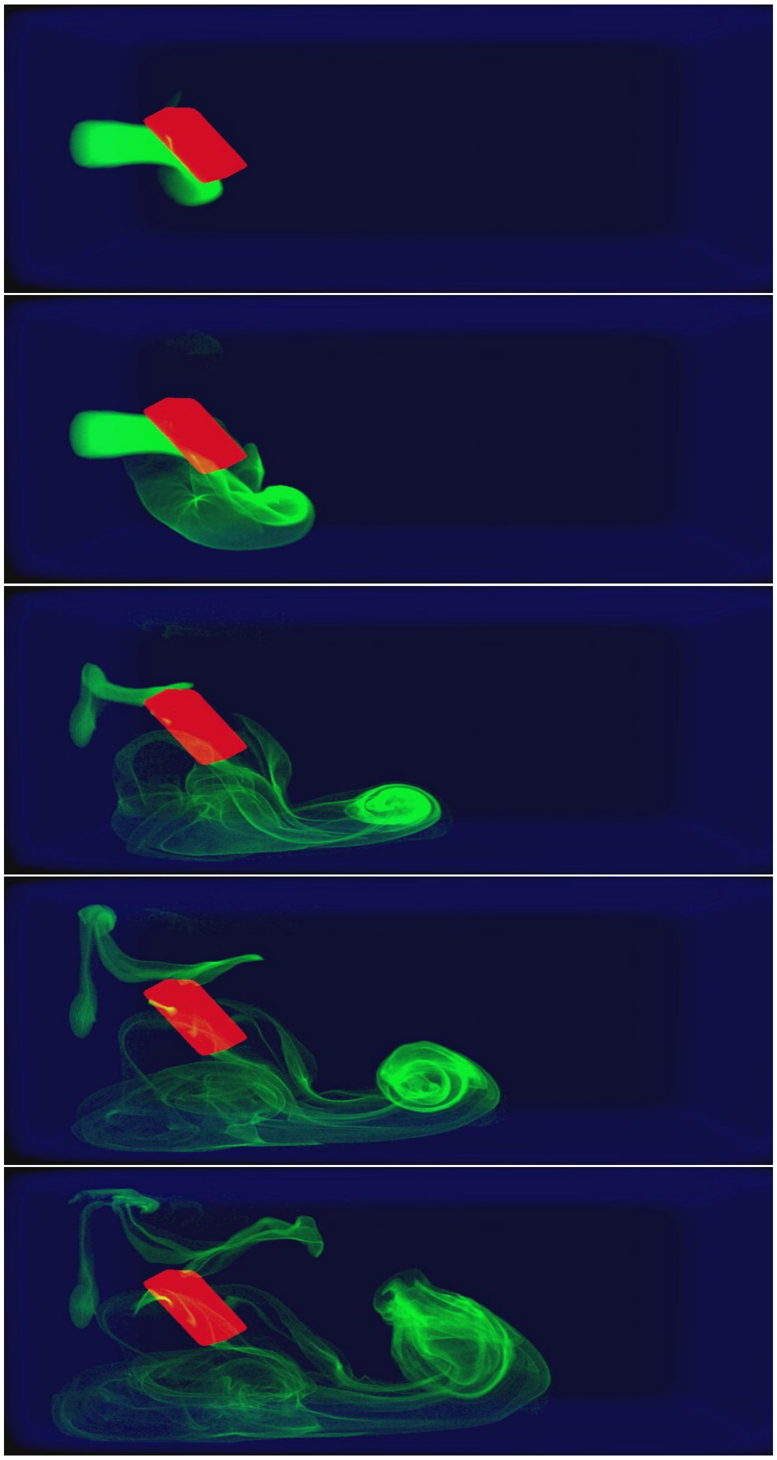


Figure 6.6: Time series of a simulation of bouyant flow (green) confined within a box (blue boundary) and flowing around a slab obstacle (red). Frames 11, 29, 74, 124, 200 from a 200 frame simulation.

Chapter 7

Gridless Advection

In this chapter we examine the benefits and costs of gridless advection in more detail. For some situations there is only a minor cost with very worthwhile improvements in image quality. In the extreme, gridless advection may be too expensive. This discussion also points the way to the chapter on Semi-Lagrangian Mapping (SELMA), which provides an efficient compromise enabling detail beyond grid dimensions while returning to a cost that is constant per frame. SELMA produces nearly the full benefits of gridless advection while suffering only the cost of gridded calculations.

Note that gridless advection is not a method of simulating fluid dynamics. It is a method of applying, at render time, the results of simulations in order to have more control of the look of the rendered volume. For the discussion in this chapter, we limit ourselves to just the application of velocity fields (simulated or not) to density fields. Gridless advection is more widely applicable though.

7.1 Spatial Gradients

Before getting into the algorithm for gridless advection, it is worthwhile to discuss a few concepts that motivate using it in the first place.

The value of fluid simulations in production is the combination of spatial structure and motion that they produce. The underlying physical model, the Navier-Stokes equations, tightly couple the structure and motion on many scales, transfering energy and momentum from large scales to small scales in a process called a *cascade*. This cascade is an important phenomenon that identifies the combined structure and motion as being fluid-like.

But fluid simulators have limits to how much spatial detail and motion they can simulate and cascade, and that limit is readable to observers as artifical motion or excessing numerical dissipation.

There are models of the energy cascade that are based on statistical arguments. Conceptually the turbulent motion of the fluid can be treated as a random process from which correlation functions can be built. While these models

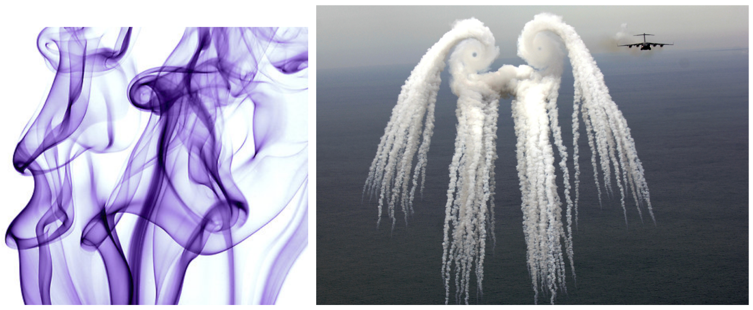


Figure 7.1: Examples of filaments and sheets forming in fluid flow.

provide very specific predictions of the ensemble fluid behavior, actual motion in any member of the ensemble is very different from the correlation. Another, more useful, way of characterizing the cascade is through the size of spatial gradients of quantities that undergo fluid motion, e.g. the spatial gradients of smoke density or the velocity field. As a fluid evolves, the density field acquires spatial structures in the form of one-dimensional filaments and two-dimensional sheets. As the evolution continues, these filaments and sheets become thinner, interact, generate new structures with greater spatial gradients, and ultimately reach the dissipation scale where they are converted into heat. Examples of these filaments and sheets are show in figure 7.1.

The elongation of filaments and thinning of the sheets have large spatial gradients in the vicinity of these features. The purpose of gridless advection is to try to preserved these gradients and prevent their numerical dissipation.

7.2 Algorithm

We begin with a look at the impact of one step of gridless advection. Imagine you have produced a velocity field $\mathbf{u}(\mathbf{x}, t)$, which may be from a simulation, from some sort of procedural algorithm, or from data. Imagine also that you have a field of density $\rho(\mathbf{x})$ that you want to "sweeten" by applying some advection. A single step of advection generates the new field

$$\rho_1(\mathbf{x}) = \rho(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_1)\ \Delta t) \tag{7.1}$$

where the time step Δt serves to control the magnitude of the advection to suit your taste. The advected density ρ_1 is not sampled onto a grid. Equation 7.1 is a procedural algorithm to be evaluated when the density is used during a volume render or some other application. Figure 7.2 shows a simple spherical volume of uniform density after advection by a noisy velocity field. For the velocity field in the example, we generated a noise vector field that is gaussian distributed, with spatial correlation and divergence-free. Extreme advection like this can transform simply shaped densities into complex organic distributions.

This can be extended to two steps of advection:

$$\rho_2(\mathbf{x}) = \rho(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_2)\ \Delta t - \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_2)\ \Delta t, t_1)\ \Delta t) \tag{7.2}$$

and to three steps of advection:

$$\rho_3(\mathbf{x}) = \rho(\mathbf{x}-\mathbf{u}(\mathbf{x}, t_3)\ \Delta t-\mathbf{u}(\mathbf{x}-\mathbf{u}(\mathbf{x}, t_3)\ \Delta t, t_2)\ \Delta t-\mathbf{u}(\mathbf{x}-\mathbf{u}(\mathbf{x}, t_3)\ \Delta t-\mathbf{u}(\mathbf{x}-\mathbf{u}(\mathbf{x}, t_3)\ \Delta t, t_2)\ \Delta t, t_1)\ \Delta t) \tag{7.3}$$

The iterative algorithm for $n+1$ gridless advection steps comes from the results for n steps as

$$\rho_{n+1}(\mathbf{x}) = \rho_n(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_{n+1})\Delta t) \tag{7.4}$$

but, despite the simplicity of this expression, you can see from equation 7.3 that the algorithm grows linearly in complexity with the number of steps taken. This causes the evaluation time to grow linearly as well, so that a large number of advection steps become impractically slow for productions. In that case, the alternative SELMA algorithm can be employed (chapter 8).

7.3 Spatial Gradients in Gridless Advection

So how does this algorithm handle the spatial gradients in the fluid motion? How does it compare to not using gridless advection?

First lets look at not using gridless advection. Supose we have simulated the motion of a density field ρ on a rectangular grid. Spatial gradients of the density are determined by the specifics of the advection algorithm employed in the simulation. For example, for semi-lagrangian advection, the gradient is bounded by

$$O\left(\mid \nabla\rho_n \mid\right) \sim \frac{\rho_{max}}{\Delta x} \quad \text{(semi-lagrangian)} \tag{7.5}$$

where ρ_{max} is the maximum initial value of the density field at any grid point, and Δx is the cell size of the grid. This is purely an upper bound that does not take into account the numerical dissipation that interpolation induces in semi-lagrangian advection. For a minimally viscous advection scheme like Quick, the density gradient also depends on the velocity gradient, which in turn is limited by the CFL stability condition, so that the bound is

$$\begin{aligned} O\left(\mid \nabla\rho_n \mid\right) &\sim \Delta t \,|\nabla \mathbf{u}_n|\ |\nabla\rho_{n-1}| \\ &\sim \Delta t \frac{u_{CFL}}{\Delta x}\,|\nabla\rho_{n-1}| \\ &\sim |\nabla\rho_{n-1}| \\ &\sim \frac{\rho_{max}}{\Delta x} \quad \text{(quick)} \end{aligned} \tag{7.6}$$

Quick spatial gradients stay essentially constant over time and dissipate very little. Ultimately the gradient limit is the finite difference limit for densities on a grid. For both examples these estimates are upper bounds, and in practice numerical dissipation prevents these bounds from being reached.

How does the gradient for gridless advection look? From the iterative equation 7.4, the density gradient is exactly

$$\nabla \rho_{n+1} \;=\; (\mathbf{1} - \Delta t \; \nabla \mathbf{u}_{n+1}) \;\cdot\; \nabla \rho_n \tag{7.7}$$

where $\mathbf{1}$ is the 3×3 identity matrix. We want to see if gridless advection can increase the spatial gradient anywhere in the volume. This would be indicated if the magnitude of any component of $\nabla \rho_{n+1}$ is greater than that for the corresponding component of $\nabla \rho_n$. It is useful to look at the eigenvalues of the matrix $(\mathbf{1} - \Delta t \; \nabla \mathbf{u}_{n+1})$, which are based on the real eigenvalues of the matrix $\nabla \mathbf{u}_{n+1}$, which we call λ_i. The eigenvalues are then

$$1 \;-\; \Delta t \; \lambda_i \tag{7.8}$$

Note that if the fluid velocity is incompressible, then by definition $\sum_{i=1}^{3} \lambda_i \;=\; 0$. This means that if any of the eigenvalues λ_i are not zero (i.e. there is a velocity gradient), then some of the λ_i are positive and some are negative. In that case, in the eigendirection(s) with negative gradient eigenvalue, the component $1 \;-\; \Delta t \; \lambda_i \;\; > 1$, which means in those direction(s), the spatial gradient of the density grows. Physically, the condition that $\lambda_i < 0$ is that the flow is stretching in that particular direction, and stretching induces higher spatial gradients. Note that one or two of the λ_i can be negative, but not all three in order to keep the flow incompressible. When only one component is negative, a filament is created; when two components are negative a thin sheet is created.

So where ever a flow creates filaments and sheets, gridless advection amplifies increased spatial gradients and enhances the visual appearance of the structure. The amount of increase of the spatial gradients is not limited by any spatial grid either, and so can grow enormously high. Further, that growth is related to the spatial structure of the flow field, and so is naturally related to the physical simulation. The growth can exceed physical limits however, because it dones not feed back any forcing of the velocity field dynamics, and does not respond to physical dissipation at very small scales.

7.4 Examples

We can illustrate the impact of advection with some examples. A common use for gridless advection is to apply it to an existing simulation to sharpen edges. Figure 7.3 shows a density distribution consisting of a wall of small spheres of density. Each row has a different color. A fluid simulation unrelated to this density field has been created, and when we advect the density and sample it to a grid every time step, then the advected density field after 60 frames looks like figure 7.4. There has been a substantial loss of density due to numerical dissipation, but also the density distribution looks soft or diffused. Even the density in the top left and bottom right, which has gone through very little advection, has blurred substantially. If we used gridded sampling of the advected density on the first 59 frames, then gridless advection on the last frame via equation

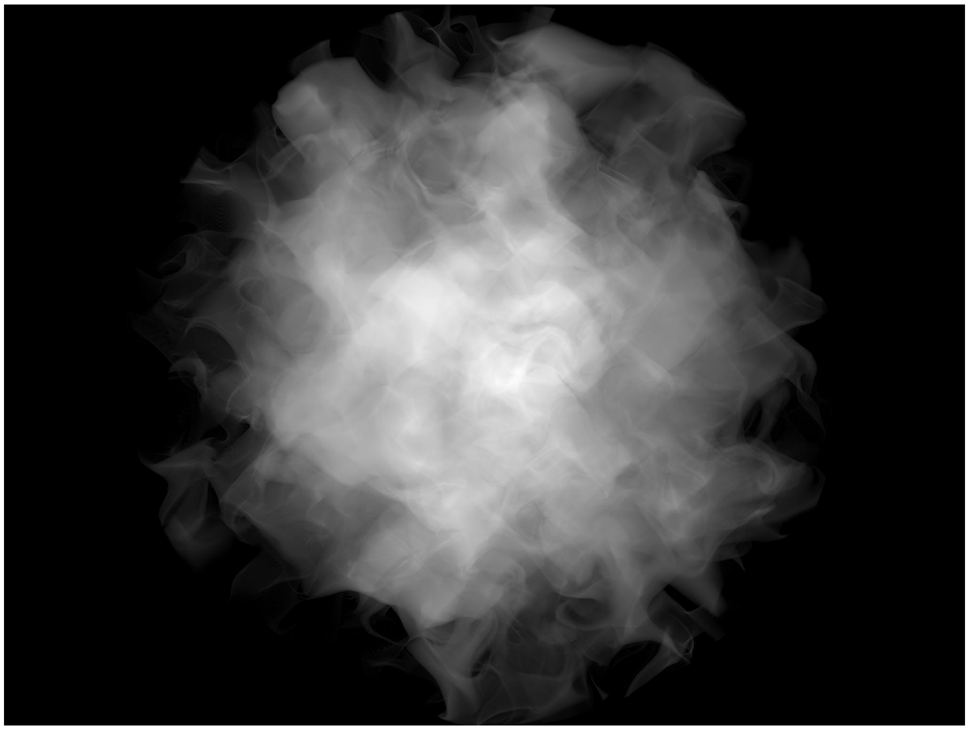


Figure 7.2: Illustration of the effect of a single step of gridless advection. The unadvected density field is a sphere of uniform density.

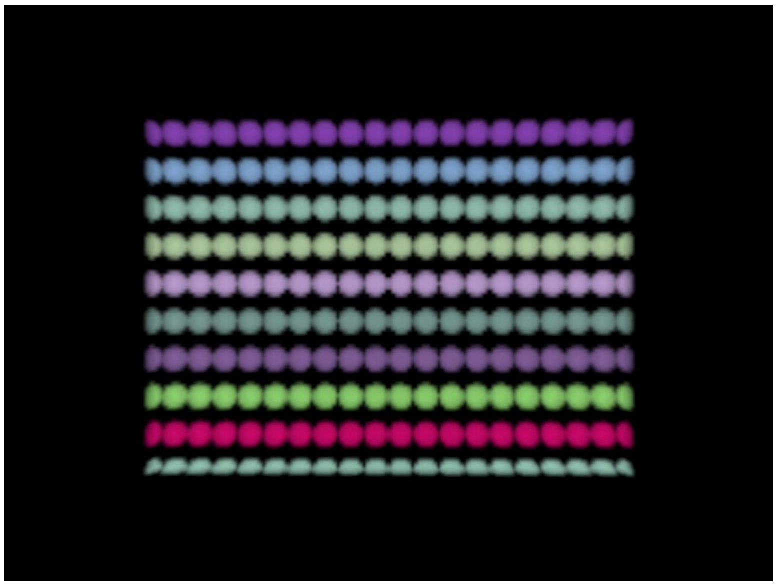


Figure 7.3: Unadvected density distribution arranged from a collection of spherical densities.

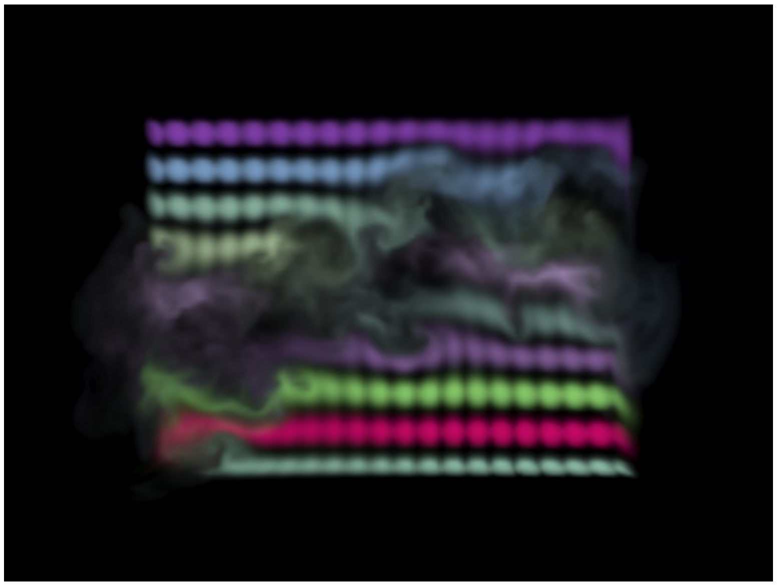


Figure 7.4: Density distribution after 60 frames of advection and sampling to a grid each frame.

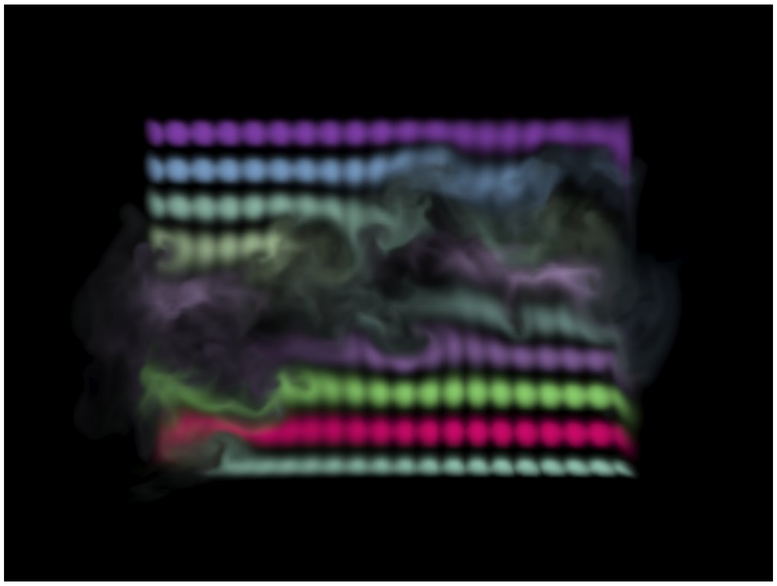


Figure 7.5: Density distribution after 59 frames of advection and sampling to a grid each frame, and one frame of gridless advection. The edges of filaments have been subtley sharpened.

7.1, the result is in figure 7.5. There is a slight sharpening of edges in the gas structure. This is more noticeable if we advect and sample for 50 frames, then gridlessly advect for 10 frames, as in figure 7.6. In fact, the image shows a lot of aliasing because the raymarch step size is not able to pick up the fine details in the density. This is corrected in figure 7.7 by raymarching with a step size 1/10-th the grid resolution. Finally, just to carry it to the extreme, figure 7.8 shows the density field after all 60 frames have been gridlessly advected. The raymarch is finely sampled to reduce aliasing of fine structures in the field, although some are still visible. Also very important is the fact that gridless advection generates structures in the volume that have more spatial detail than the original density distribution or velocity field. This is a very valuable effect, as it provides a method to simulate at relatively coarse resolution, then refine at render time via gridless advection. Further, this refinement does not dramatically alter the gross motion or features of the density distribution, whereas rerunning a simulation at higher resolution generally produces a completely different flow from the lower resolution simulation. A variation on this is to gridlessly advect a volume density with a random velocity field in order to make it more "natural" looking, as was done in figure 7.9.

We can evaluate the relative performance of various options, e.g. how many gridless steps to take, using the graph in figure 7.10, showing the amount of RAM and the CPU time cost for the raymarch render for each option. The execution time for setting up the gridless advection processing is essentially

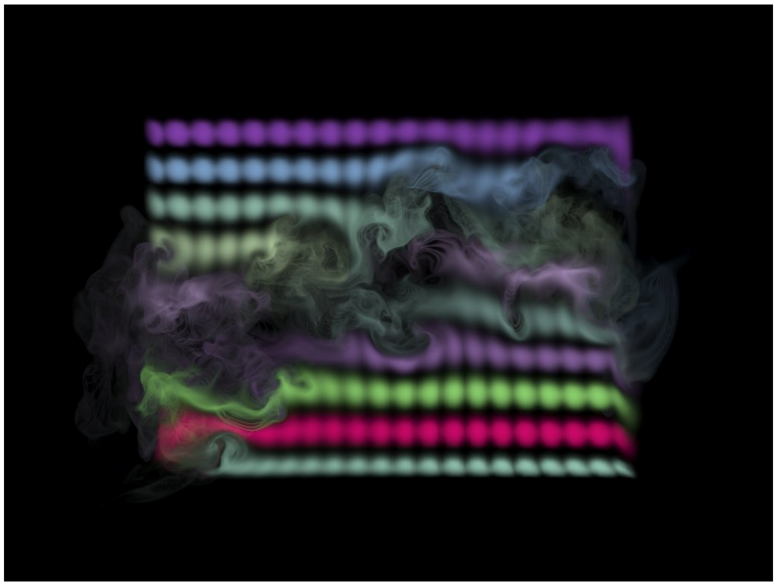


Figure 7.6: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The sharpening of details has increased to the point that the detail is finer than the raymarch stepping, causing significant aliasing in the render.

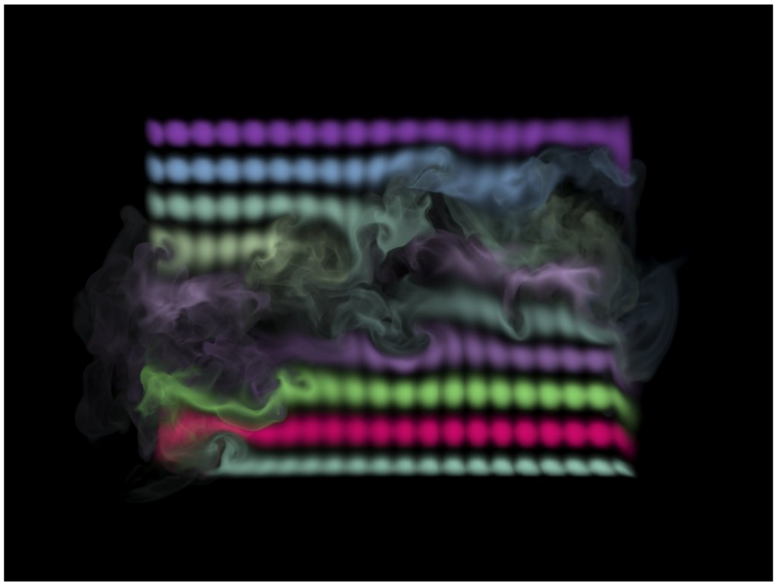


Figure 7.7: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The fine detail in the density field is now resolved by using a finer raymarching step (1/10-th the grid resolution).

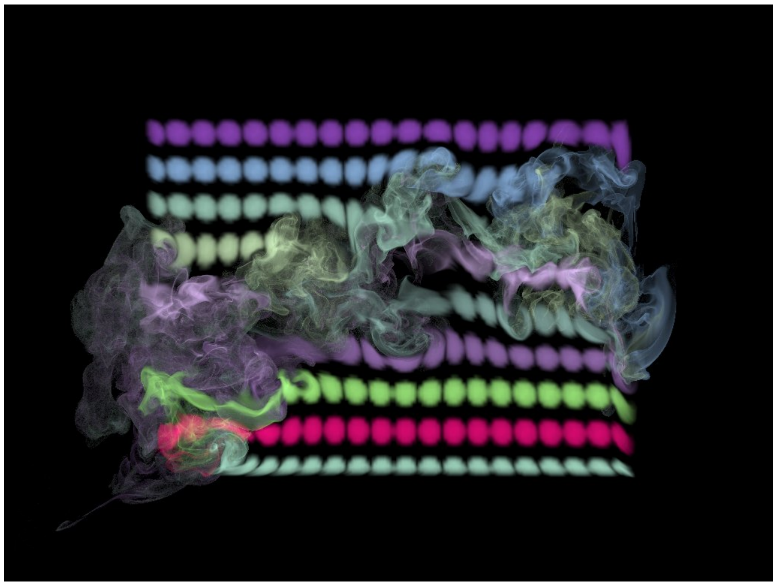


Figure 7.8: Density distribution after 60 frames of gridless advection. The fine detail in the density field is resolved by using a fine raymarching step.

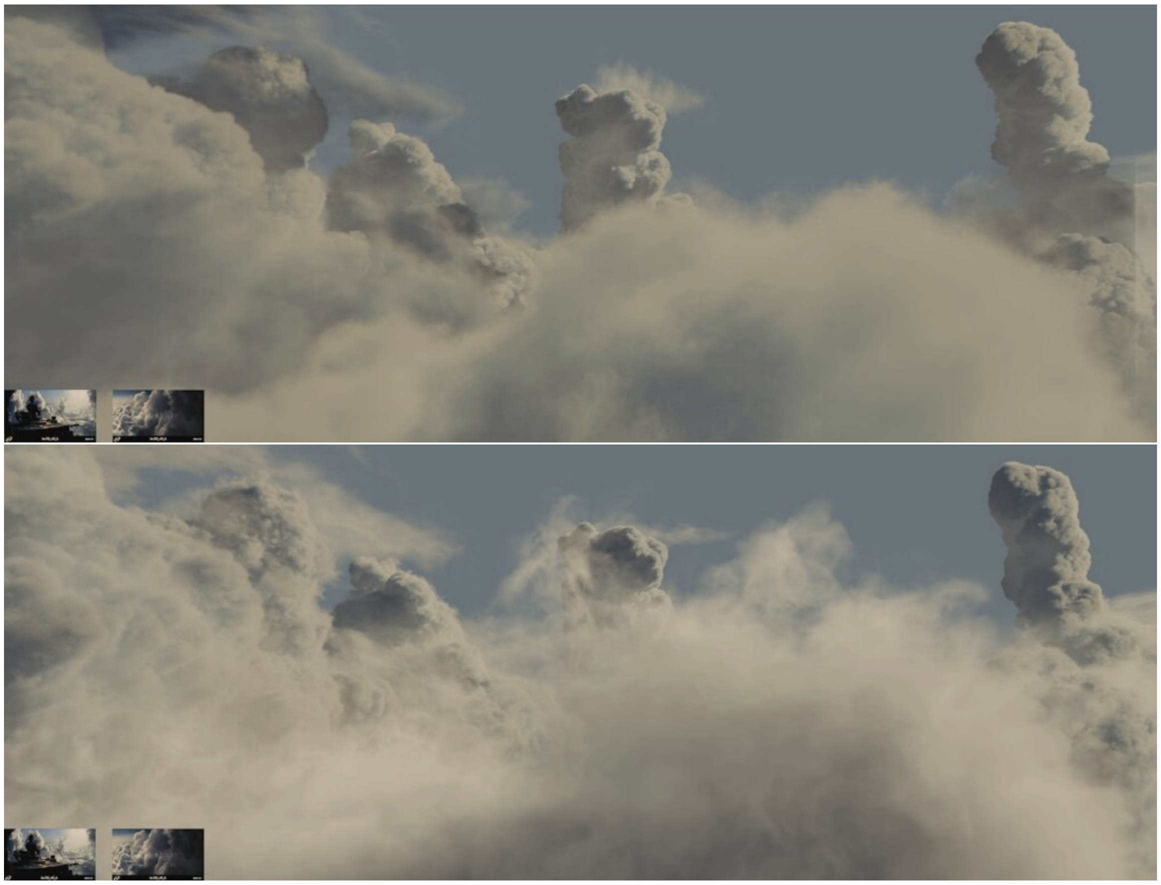


Figure 7.9: Clouds rendered for the film *The A-Team* using gridless advection to make their edges more realistic. The velocity field was based on Perlin noise. Top: foreground clouds without advection; bottom: foreground clouds after gridless advection.

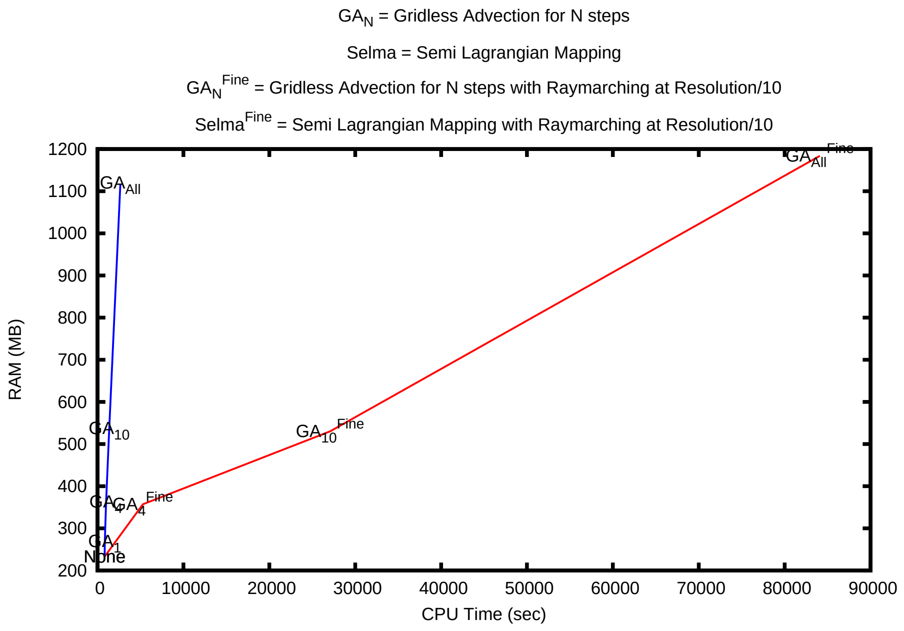


Figure 7.10: Performace of gridless advection as the number of advection frames grows. The steep blue line is gridless advection rendered with the raymarch step equal to the grid resolution. The red line is a raymarch step equal to one-tenth of the grid resolution. These results are not from a production-optimized renderer, so time and memory values should be taken as relative measures only.

negligible compared to the time spent evaluating the fields during the render. The raymarcher used for this data is a simple one not optimized for production use, so the results should be indicative of relative behavior only, not actual production resource costs. The blue line is the performance for gridless advection as the number of gridless steps increase, while leaving the raymarch step size equal to the cell size of the velocity field. Note that RAM increases linearly with the number of gridlessly advected frames, because the velocity fields of those frames must be kept available for the evaluation of the advections. With a large number of advections, the spatial detail generated includes fine filaments and curved sheets that are so thin that raymarch steps equal to the grid resolution are insuffient to resolve that fine detail in the render. Using 10 times finer steps in order to capture detail, the images look much better and the red line performance is produced. The longest time shown is over 80000 seconds, nearly 1 cpu day. This scale of render time is not practicable. In practice using gridless advection for more than about 5-10 steps extends the render time, due to the additional advection evaluations and the finer raymarch stepping, to the limit that most productions choose to take.

Fortunately there is a practical compromise, called Semi-Lagrangian Mapping (SELMA).

Chapter 8

SEmi-LAgrangian MApping (SELMA)

The key to finding a practical compromise between gridless advection and sampling the density to a grid at every frame is to recognize that gridless advection is a remapping of the density field to a warped space. You can see that by rewriting equation 7.1 as

$$\rho_1(\mathbf{x}) = \rho\left(\mathbf{X}_1(\mathbf{x})\right) \tag{8.1}$$

where the warping vector field $\mathbf{X}_1$ is

$$\mathbf{X}_1(\mathbf{x}) = \mathbf{x} - \mathbf{u}(\mathbf{x}, t_1)\ \Delta t \tag{8.2}$$

Similarly, the equations for ρ_2 and ρ_3 also have forms involving warp fields:

$$\rho_2(\mathbf{x}) = \rho\left(\mathbf{X}_2(\mathbf{x})\right) \tag{8.3}$$

where

$$\mathbf{X}_2(\mathbf{x}) = \mathbf{x} - \mathbf{u}(\mathbf{x}, t_2)\ \Delta t - \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_2)\ \Delta t, t_1)\ \Delta t \tag{8.4}$$

and

$$\rho_3(\mathbf{x}) = \rho\left(\mathbf{X}_3(\mathbf{x})\right) \tag{8.5}$$

where

$$\mathbf{X}_3(\mathbf{x}) = \mathbf{x} - \mathbf{u}(\mathbf{x}, t_3)\ \Delta t - \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_3)\ \Delta t, t_2)\ \Delta t - \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_3)\ \Delta t - \mathbf{u}(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_3)\ \Delta t, t_2)\ \Delta t, t_1)\ \Delta t \tag{8.6}$$

Finally, for frame n, the density ρ_n has a warp field also:

$$\rho_n(\mathbf{x}) = \rho\left(\mathbf{X}_n(\mathbf{x})\right) \tag{8.7}$$

with an iterative form for the mapping:

$$\mathbf{X}_n(\mathbf{x}) = \mathbf{X}_{n-1}\left(\mathbf{x} - \mathbf{u}(\mathbf{x}, t_n)\Delta t\right) \tag{8.8}$$

So the secret to capturing lots of detail in gridless advection is that the mapping function $\mathbf{X}(\mathbf{x})$ carries information about how the space is warped by the fluid motion. The gridless advection iterative algorithm is equivalent to executing the iterative equation 8.8, so the Felt code

```
density = advect( density, velocity, dt );
```

is mathematically and numerically equivalent to code that explicitly invokes a mapping function like:

```
Xmap = advect(Xmap, velocity, dt);
density = compose(initialdensity, Xmap);
```

as long as the map Xmap is a vectorfield initialized in an earlier code segment as

```
vectorfield Xmap = identity();
```

The practical advantage of recasting the problem as a map generation is that it allows us to take one more step. Sampling the density onto a grid at every frame leads to substantial loss of density and softening of the spatial structure of the density. But now we have the opportunity to instead sample the map $\mathbf{X}(\mathbf{x})$ onto a grid at each frame. This limits the fine detail within the map, because it limits structures within the map to a scale no finer than grid resolution. However, what is left still generates highly detailed spatial structures in the density. For example, returning to the example of figures 7.3 through 7.8, applying gridding of the mapping function produces the highly detailed result in figure 8.1. The change to the Felt code is relatively small:

```
Xmap = advect(Xmap, velocity, dt);
// Sample map onto into a grid
vectorcache XmapCache(region);
cachewrite( XmapCache, Xmap );
// Replace Xmap with the gridded version
Xmap = cacheread(XmapCache);
density = compose(initialdensity, Xmap);
velocity = advect( velocity, velocity, dt ) + dt*gravity*density ;
velocity = fftdivfree( velocity, region );
```

where XmapCache is a vectorcache into which we sample the Semi-Lagrangion mapping function $\mathbf{X}$. This restructuring of the density advection based on a mapping function that is grid-sampled is given the name SELMA for SEmi-LAgrangian MApping.

How does SELMA constitute a good compromise between sampling the density onto a grid at each time step, with relatively low time and memory resources but limited spatial detail, and gridlessly advection, with higher time and memory requirements but very high spatial detail? There are benefits

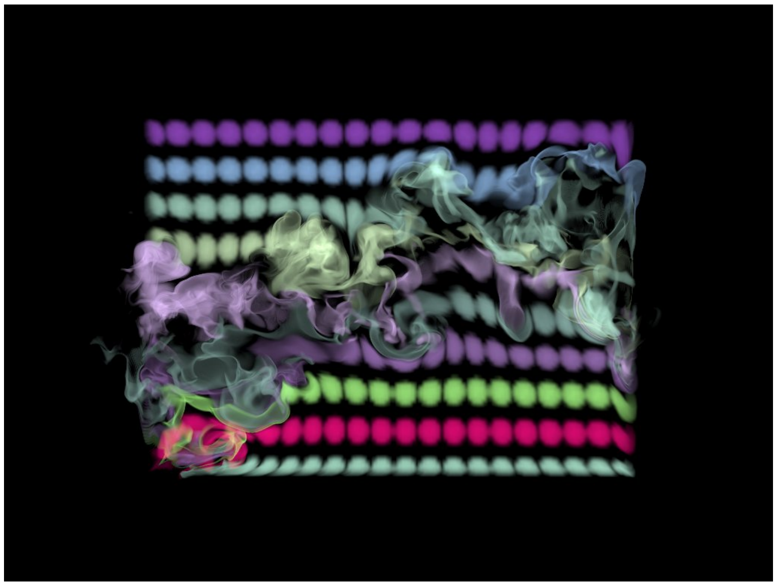


Figure 8.1: Density distribution after 60 frames of SELMA advection. The fine detail in the density field is resolved by using a fine raymarching step.

in both memory and speed. Because the mapping function is sampled to a grid each time step, the collection of velocity fields need no longer be kept in memory, so the memory requirement for SELMA is both lower than gridless advection and constant over time (whereas it grew linearly with the number of time steps in gridless advection). For speed, SELMA has to perform a single interpolated sampling of the gridded mapping function each time the density value is queried, and the cost for this is fixed and constant for each simulation step. Comparatively, gridless advection requires evaluating a chain of values of each velocity field along a path through the volume, the cost of which grows linearly with the number of time steps. These improvements in performance are clear in figure 8.2, which compares the performance of gridless advection and SELMA. The increase in RAM for the case "Selma$^{\text{Fine}}$" is because the grid for the SELMA map was chosen to be finer than for the velocity field.

Figure 8.3 shows SELMA as used for the production of *The A-Team.* An aircraft passing through cloud material leaves behind a wake disturbance in the cloud. The velocity field is from a fluid simulation that does not include the presence of the cloud. The cloud was modeled using the methods in chapter 3, then displaced using SELMA.

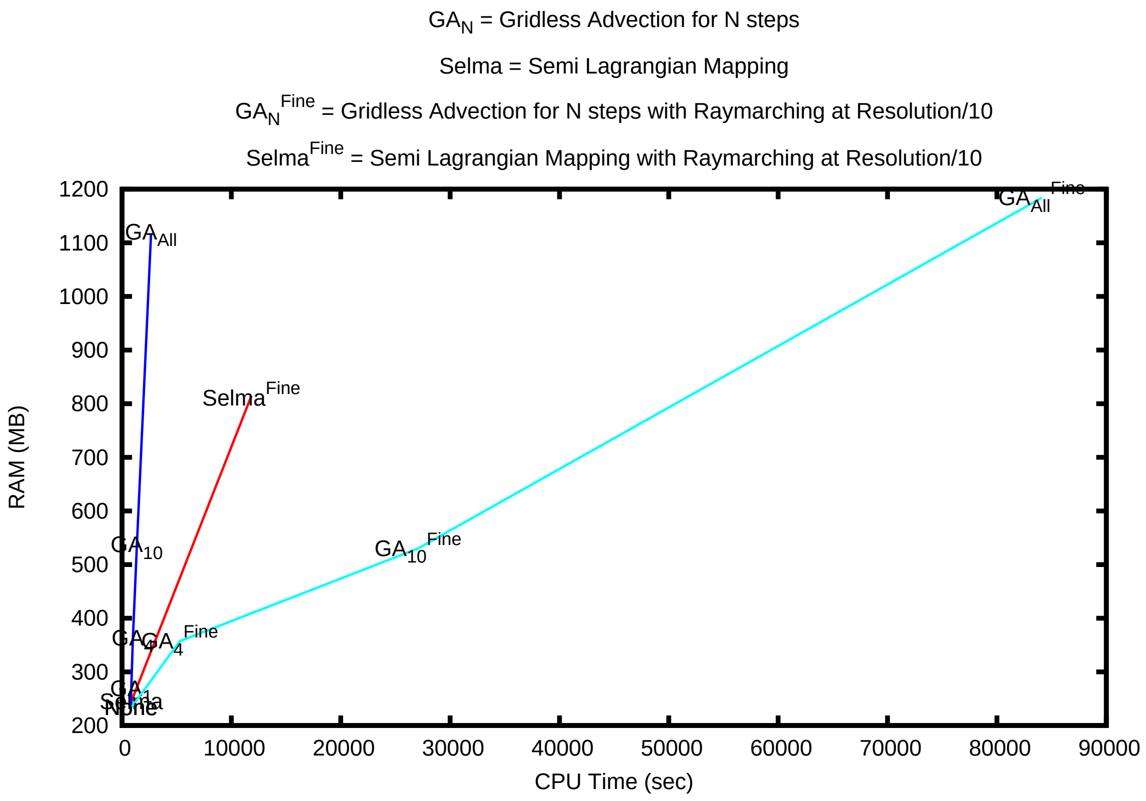


Figure 8.2: Comparison of the performace of Gridless Advection and SELMA.

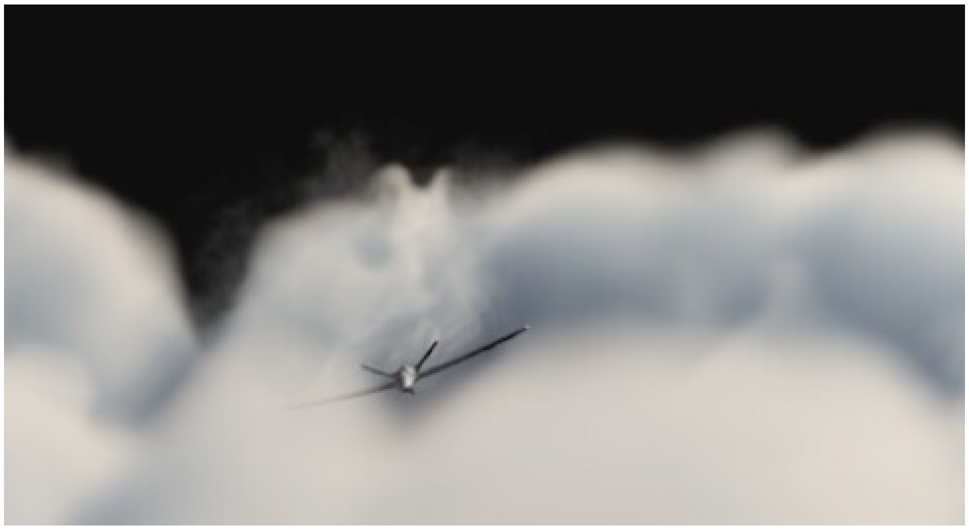


Figure 8.3: Example of SELMA used in the production of *The A-Team* to apply a simulated turbulence field to a modeled cloud volume as an aircraft passes through.

Appendix A

Appendix: The Ray March Algorithm

A.0.1 Rendering Equation

The algorithm for ray marching in volume rendering is essentially just the numerical approximation of the rendering equation for the amount of light $L(\mathbf{x}_C, \mathbf{n}_P)$ received by a camera located at position $\mathbf{x}_C$, at the pixel that is looking outward in the direction $\mathbf{n}_P$. The rendering equation accumulates light emitted by the volume along the line of sight of the pixel. The accumulation is weighted by the volumetric attenuation of the light between the volume point and the camera, and by the scattering phase function which scatters light from the light source into all directions. The rendering equation in this context is a single-scatter approximation of the fuller theory of radiative transfer:

$$L(\mathbf{x}_C, \mathbf{n}_P) \;=\; \int_0^\infty ds\; C^T(\mathbf{x}(s))\; \rho(\mathbf{x}(s))\; \exp\left\{ -\int_0^s ds'\; \kappa\; \rho(\mathbf{x}(s')) \right\} \tag{A.1}$$

The density $\rho(\mathbf{x})$ is a material property of the volume, representing the amount of per unit volume present at any point in space. Note that anywhere that the density is zero has no contribution to the light seen by the camera. The ray path $\mathbf{x}(s)$ is a straight line path originating at the camera and moving outward along the pixel direction to points in space a distance s from the camera.

$$\mathbf{x}(s) \;=\; \mathbf{x}_C \;+\; s\; \mathbf{n}_P \tag{A.2}$$

The total color is a combination of the color emission directly from the volumetric material, and the color from scattering of external light sources by the material.

$$C^T(\mathbf{x}(s)) \;=\; C^E(\mathbf{x}(s)) \;+\; C^S(\mathbf{x}(s)) \otimes C^I(\mathbf{x}(s)) \tag{A.3}$$

Both C^E and C^S are material color properties of the volume, and are inputs to the rendering task. The illumination factor C^I is the amount of light from any light sources that arrives at the point $\mathbf{x}(s)$ and multiplies against the color of the material. For a single point-light at position $\mathbf{x}^L$, the illumination is the color of the light times the attenuation of the light through the volume, and times the phase function for the relative distribution of light into the camera direction

$$C^I(\mathbf{x}) \ = \ C^L \ T^L(\mathbf{x}) \ P(\mathbf{n} \cdot \mathbf{n}^L) \tag{A.4}$$

with the light transmissivity being

$$T^L(\mathbf{x}) = \exp\left\{ - \int_0^D ds' \kappa \ \rho(\mathbf{x} + s\mathbf{n}^L) \right\} \tag{A.5}$$

where is the distance from the volume position $\mathbf{x}$ and the position of the light: $D = |\mathbf{x} - \mathbf{x}^L|$, and $\mathbf{n}^L$ is the unit vector from the volume position to the light position:

$$\mathbf{n}^L = \frac{\mathbf{x}^L - \mathbf{x}}{|\mathbf{x}^L - \mathbf{x}|} \tag{A.6}$$

For N light sources, this expression generalizes to a sum over all of the lights:

$$C^I(\mathbf{x}) \ = \ \sum_{i=1}^{N} C_i^L \ T_i^L(\mathbf{x}) \ P(\mathbf{n} \cdot \mathbf{n}_i^L) \tag{A.7}$$

The phase function can be any of a variety of shapes, depending on the material properties of the volume. One common choise is to ignore it as an additional degree of freedom, and simply use $P(\mathbf{n} \cdot \mathbf{n}^L) = 1$. Another choice that introduces only a single control parameter g is the Henyey-Greenstein phase function

$$P_{HG}(\mathbf{n} \cdot \mathbf{n}^L) \ = \frac{1}{4\pi} \ \frac{1 - g^2}{(1 \ + g^2 \ - \ 2g\mathbf{n} \cdot \mathbf{n}^L)^{3/2}} \tag{A.8}$$

This function is plotted in figure A.1 for several values of g. As $g \to 1$, the phase function becomes sharply peaked in the forward direction,i.e. $\mathbf{n} \cdot \mathbf{n}^L \sim 1$. As $g \to -1$, the strong peak is in the backward direction, $\mathbf{n} \cdot \mathbf{n}^L \sim -1$. Phase functions have been measured and calculated for many natural materials, such as clouds, water, and tissues [6]. A model phase function called the Fournier-Forand phase function fits many natural materials well:

$$\begin{aligned}
P_{FF}(\Theta) \ &= \ \frac{1}{4\pi(1-\delta)^2\delta^\nu} \ \left[\nu(1-\delta) - (1-\delta^\nu) + (\delta(1-\delta^\nu) - \nu(1-\delta)) \, / \ \sin^2\left(\frac{\Theta}{2}\right) \right] \\
&+ \ \frac{1-\delta_{180}^\nu}{16\pi(\delta_{180}-1)\delta_{180}^\nu} \ \left\{ 3\cos^2\Theta \ - \ 1 \right\} && \text{(A.9)} \\
\delta \ &= \ \frac{4}{3(n-1)^2} \ \sin^2\left(\frac{\Theta}{2}\right) && \text{(A.10)} \\
\delta_{180} \ &= \ \frac{4}{3(n-1)^2} && \text{(A.11)}
\end{aligned}$$

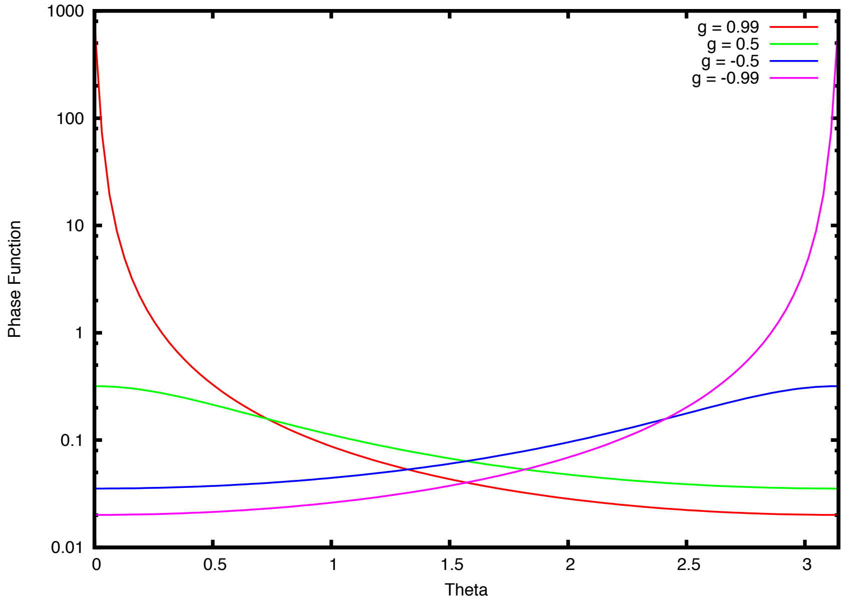


Figure A.1: The Henyey Greenstein phase function for $g = 0.99, 05, -0.5, -0.99$.

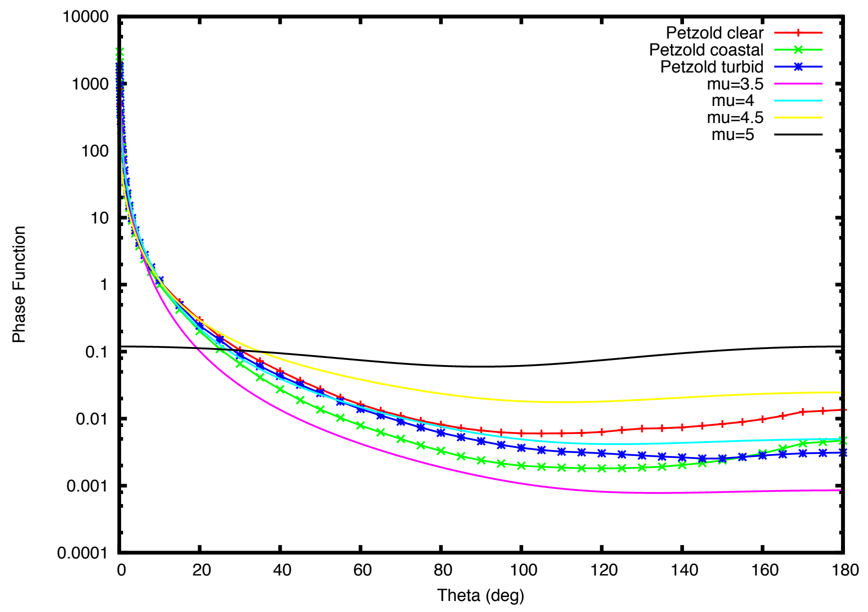


Figure A.2: The Fournier-Forand phase function for $\mu = 0.35, 0.4, 0.45, 0.5$. The parameter n has the value 1.05. Petzold's measured phase functions for clear, coastal, and turbid ocean waters are shown also.

$$\nu \quad = \quad \frac{3-\mu}{2} \tag{A.12}$$

and ν, μ, and n are physical parameters. Figure A.2 illustrates this phase function for several values of μ, along with plots of Petzold's phase function data for 3 ocean water conditions [7].

Finally, recognizing that the volumetric material occupies a finite volume of space, it is not necessary to integrate along a path from the camera to infinity. There is a point $s_0 \geq 0$ where the density starts, and a maximum distrance s_{max} past which the density is zero. So the render equation can be reduced to evaluating the integral just within those bounds:

$$L(\mathbf{x}_C, \mathbf{n}_P) \;=\; \int_{s_0}^{s_{max}} ds\; C^T(\mathbf{x}(s))\; \rho(\mathbf{x}(s))\; \exp\left\{ -\int_0^s ds'\; \kappa\; \rho(\mathbf{x}(s')) \right\} \tag{A.13}$$

A.0.2 Ray Marching

Discretizing the rendering equation A.13 leads to the ray march algorithm used in production volume rendering. The rendering equation A.13 is decomposed into a set M of small steps of length Δs, with $M\Delta s \;=\; s_{max} - s_0$. Without approximation, the rendering equation becomes

$$L(\mathbf{x}_C, \mathbf{n}_P) \;=\; \sum_{j=0}^{M-1} T_j \int_0^{\Delta s} ds\, C^T(\mathbf{x}_j{+}s\mathbf{n}_P)\; \rho(\mathbf{x}_j{+}s\mathbf{n}_P)\; \exp\left\{ -\int_0^s ds'\; \kappa\; \rho(\mathbf{x}_j + s'\mathbf{n}_P) \right\} \tag{A.14}$$

where

$$\mathbf{x}_j = \mathbf{x}_C \;+\; j\Delta s\mathbf{n}_P \tag{A.15}$$

and the transmissivity factor T_j is

$$T_j = \prod_{k=0}^{j-1} \Delta T_k \tag{A.16}$$

and

$$\Delta T_k \;=\; \exp\left\{ -\int_0^{\Delta s} ds\; \kappa\; \rho(\mathbf{x}_k + s\mathbf{n}_P) \right\} \tag{A.17}$$

Note that we can construct these quantities iteratively through the relationships

$$\mathbf{x}_j \;=\; \mathbf{x}_{j-1} \;+\; \Delta s\mathbf{n}_P \tag{A.18}$$

$$T_j \;=\; T_{j-1}\; dT_{j-1} \tag{A.19}$$

with the initial conditions

$$\mathbf{x}_0 \;=\; \mathbf{x}_C \tag{A.20}$$

$$T_0 \;=\; 1 \tag{A.21}$$

which define the ray march process.

One of the first graphics papers on this problem is by Kajiya [5]. In that paper an approximation for optically thin density is applied, i.e. it is assumed that the density across a short path segment is relatively small. In these notes we do not make that assumption. In fact, only one significant assumption is made here, namely that the color field is constant across the length of a short path segment. We do not assume the optically thin approximation that Kajiya chose. This leads to a simple but significant improvement to the algorithm that solves difficulties in how the edges of clouds/smoke/whatever are handled in compositing.

The discretization step takes the form of choosing a march step size Δs that is sufficiently small that we can assume that the color C^T is constant within the length of the step Δs. With that single choice, the rendering equation reduces to

$$L(\mathbf{x}_C, \mathbf{n}_P) \;=\; \sum_{j=0}^{M-1} \; C^T(\mathbf{x}_j)\; T_j\; \frac{1 - \Delta T_j}{\kappa} \tag{A.22}$$

This sum also can be handled via an iterative update of L. Combined with the iterations for $\mathbf{x}_j$ and T_j the complete iteration is

$$\begin{aligned} \mathbf{x}_j &= \mathbf{x}_{j-1} + \Delta s \mathbf{n}_P && (A.23) \\ L &+= C^T(\mathbf{x}_j)\, T_j\, \frac{1 - \Delta T_j}{\kappa} && (A.24) \\ T_{j+1} &= T_j\, dT_j && (A.25) \end{aligned}$$

which is the same as equations 1.1-1.4 when the positions $\mathbf{x}_j$ and transmissivities T_j are sored in a single vector and float with running updates.

Comparing to the optically-thin approach chosen by Kajiya, this algorithm is identical to that one *except* for the factor $(1-\Delta T_j)/\kappa$, which does not appear in Kajiya's treatment. However, if we apply an optically thin approximation, namely that $\Delta s \kappa \rho \ll 1$, then our factor reduces in the limit to just $\Delta s \rho(\mathbf{x}_j)$ which is the factor that appears in Kajiya's approach. So this ray march algorithm is an extension of Kajiya's which removes the optically-thin assumption. In practical use in production, it also has the benefit that it is easier to composite clouds rendered with this approach, because the edges of the clouds fade in opacity more correctly than the optically-thin approximation does.

The one item left to work out is the values of ΔT_j. This depends on how the density varies along the short path segment. The simplest approximation is to assume that the density is constant along the path. In that case

$$\Delta T_j = \exp(-\kappa\ \rho(\mathbf{x}_j)\ \Delta s) \quad (A.26)$$

Another possibility is that the density varies linearly along the short path segment. Supose the density varies linearly from $\rho^0(\mathbf{x}_j)$ at the beginning of the path and $\rho^1(\mathbf{x}_j)$ at the end of the segment, then the result is similar to the constant case, but with the constant density replaced by the average density along the path.

$$\Delta T_j = \exp(-\kappa\ (\rho^0(\mathbf{x}_j) + \rho^1(\mathbf{x}_j))\ \Delta s/2) \quad (A.27)$$

In more general situations with the density having a complex behavior along the short path segment, we can take inspiration from the linear variation case. We can evaluate an average density $\langle\rho\rangle(\mathbf{x}_j)$ along the path segment, and arrive at

$$\Delta T_j = \exp(-\kappa\ \langle\rho\rangle(\mathbf{x}_j)\ \Delta s) \quad (A.28)$$

The average density can be evaluated, for example, by sampling the density at random positions along the path, i.e.

$$\langle\rho\rangle(\mathbf{x}_j) = \frac{1}{N_s} \sum_{i=1}^{N_s} \rho(\mathbf{x}_j + r_j \Delta s \mathbf{n}_P) \quad (A.29)$$

where the N_s numbers r_j are random numbers between 0 and 1.

If the color cannot be assumed to be constant in the interval Δs, then one approach to this is to subdivide the interval further. Here again the random sampling idea can be brought to bear. Suppose we decide to subdivide into N_s subsegments, within each we can assume that the color and density are constant. The procedure can be as follows

- generate $N_s - 1$ random numbers r_j and order them so that $r_1 < r_2 < r_3 < \ldots < r_{N_s-1}$. For this notation, we can define $r_0 = 0$.
- Accumulate through the subintervals $j = 1, \ldots, N_s - 1$ exacly as for the primary intervals:

$$\begin{aligned}
\mathbf{x} \quad &+= \quad r_j \; \Delta s \; \mathbf{n}_P \\
\Delta T \quad &= \quad \exp\left\{-(r_j - r_{j-1}) \; \Delta s \; \rho(\mathbf{x}) \; \kappa\right\} \\
L \quad &+= \quad C(\mathbf{x}) \; T \; \frac{(1 - \Delta T)}{\kappa} \\
T \quad &*= \quad \Delta T
\end{aligned}$$

Bibliography

[1] *Introduction to Implicit Surfaces*, Jules Bloomenthal (ed.), Morgan Kaufmann, (1997).

[2] Alan Kapler, "Avalanche! snowy FX for XXX," *ACM SIGGRAPH 2003 Sketches and Applications*, (2003)

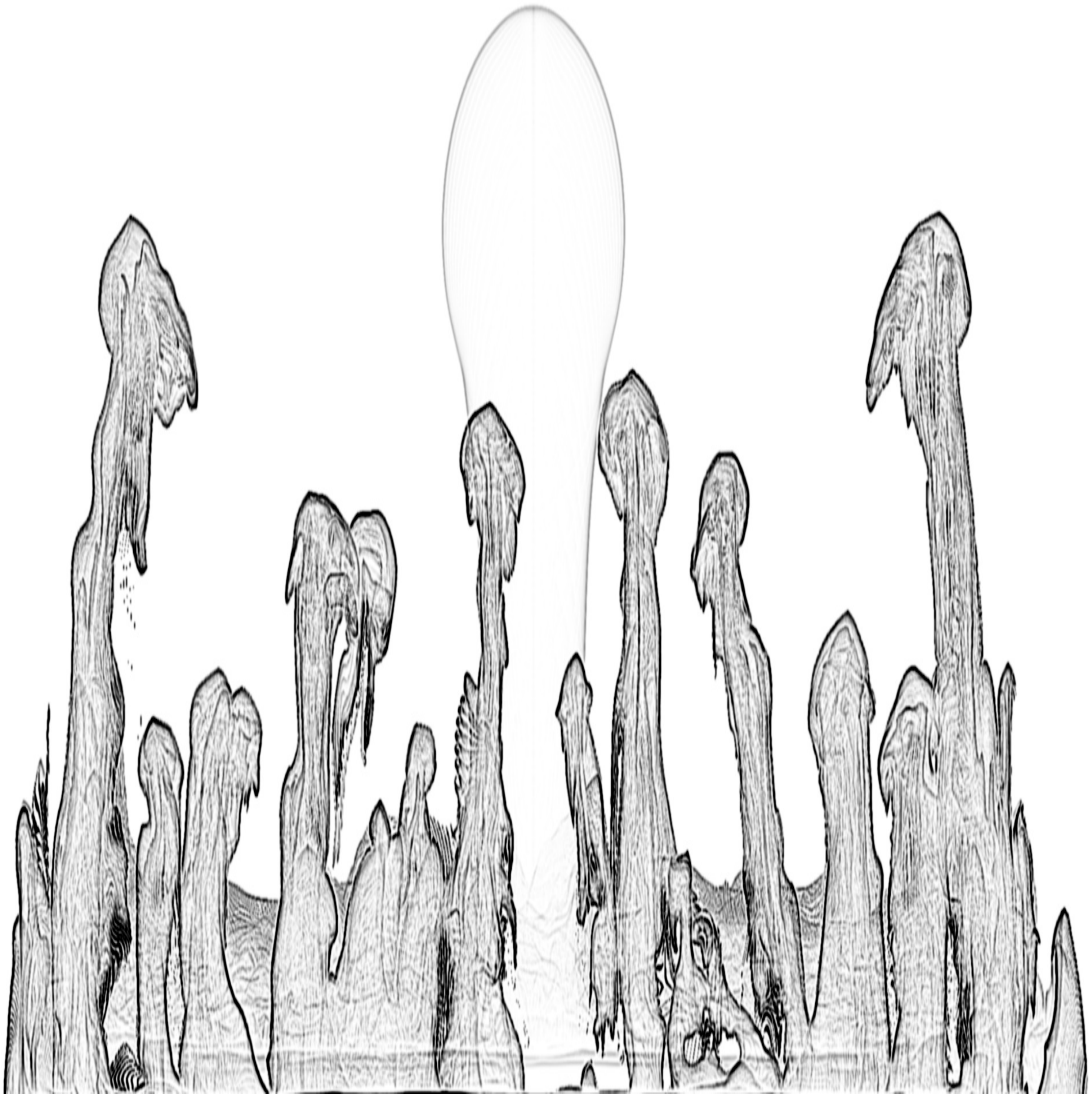
[3] Ken Museth and Michael Clive, *CrackTastic: fast 3D fragmentation in "The Mummy: Tomb of the Dragon Emperor"*, International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH, Los Angeles, California, 2008.

[4] David S. Ebert, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, Steven Worley, *Texturing & Modeling, A Procedural Approach*, 3rd edition, Morgan-Kaufmann, (2002).

[5] Kajiya paper on rendering equation

[6] Ocean Optics Webbook: Scattering. The Fournier-Forand phase function

[7] Ocean Optics Webbook: Scattering. Petzold's Measurements



Index

```
amp=0
corr = 0
levy = -0.5
numchildren = 1000000000
octaves = 5
rough = 0.5
```

Mantra Volume Rendering

Andrew Clinton

August 2, 2011

SIDE EFFECTS SOFTWARE

Table of Contents

1 History of Mantra Volume Rendering

Mantra is a production renderer originally developed for PRISMS in the early 1990s. The software has evolved over the years and is now provided as the built-in renderer within the Houdini 3D animation package. Mantra was designed to roughly follow the traditional RenderMan pipeline – with programmable shaders for surface, displacement, lights, and fog. The shading language, VEX, is an interpreted C-like language and provides a full suite of graphics programming operations.

Volumetric effects have traditionally been achieved in Mantra using either sprite rendering or fog shaders. Sprite rendering produces volume-like renders by rendering polygons with a texture map applied - usually with an alpha mask to smoothly blend with the background. Fog shaders can produce volumetric effects if the shader is instrumented with a ray marcher to march through user-defined volumetric data. Both techniques suffer from a number of drawbacks which limit their usefulness. Sprite rendering, although simple to render, will often generate physically inaccurate images due to the underlying 2D nature of the geometry. A ray marcher embedded in a fog shader can generate correct volume renders, but is not tightly integrated with the renderer's sampling techniques making it difficult to simulate effects such as motion blur and to generate holdouts.

These notes will attempt to lay out the motivation and technical basis for Mantra's latest volumetric rendering capability (originally introduced in Houdini 9 in 2007). It was designed for production and with the intent to overcome the most significant deficiencies in the existing techniques. We'll discuss the technical trade-offs that were made and hopefully provide a good example of how volumetric rendering can be successfully integrated into an existing production renderer.

1.1 Motivating Factors

The deficiencies of the existing volume rendering approaches (sprites and fog shaders) were what primarily motivated the design of a new volume rendering algorithm in Mantra. The goals for the new design included:

- Support for true 3D volumetric effects
- Direct rendering of volumetric data as a native primitive type
- True motion blur and depth of field
- Full integration with surface rendering
- Support for deep image export (export of transparent sample lists)
- Unified shading for ray tracing and micropolygon rendering

1.2 Limitations

Some capabilities are useful in a volume renderer but were not considered essential for our design, partly due to the impact they would have on the rendering architecture. These include:

- Multiple scattering – although possible to simulate with appropriately written shaders, an explicit multiple scattering algorithm was not developed with the volume renderer.
- Varying IOR - the ability to continuously change the direction of a ray as it traverses the volume (for example, to render a mirage).
- The ability to query volume data from within a shader at positions other than the current shading position. This is similar to the limitation that when rendering surfaces, the surface shader is only initialized with global data at the currently shaded parametric coordinate.

1.3 Volume Renderer Design

Often the most important design criteria when adding any new feature to Mantra is consistency. Consistency ensures that the existing architecture is integrated as seamlessly as possible with the new functionality. In the case of volumetric rendering, we attempted to reuse the existing technology (and associated controls/parameters) in three main areas:

- Geometry: volumes can be represented as geometric primitives to the renderer like any other surface primitive.
- Shading: the same shading pipeline that applies to surfaces also applies to volumes. The surface shading context, with a few very specific extensions, was reused for volume primitives.
- Sampling: the same sampling pipeline, both for raytracing and micropolygon rendering, applies to volumes. This means that the same motion blur algorithm that produces motion blur for surfaces also works with volumes.

Treating volume rendering as an extension of the existing rendering architecture provides a significant benefit to users, in that understanding volume rendering can be viewed as a simple extension of surface rendering. Much like reuse of code, it also ensures that where the existing surface rendering techniques are robust, the volumetric extensions should also show good stability, predictability, and performance.

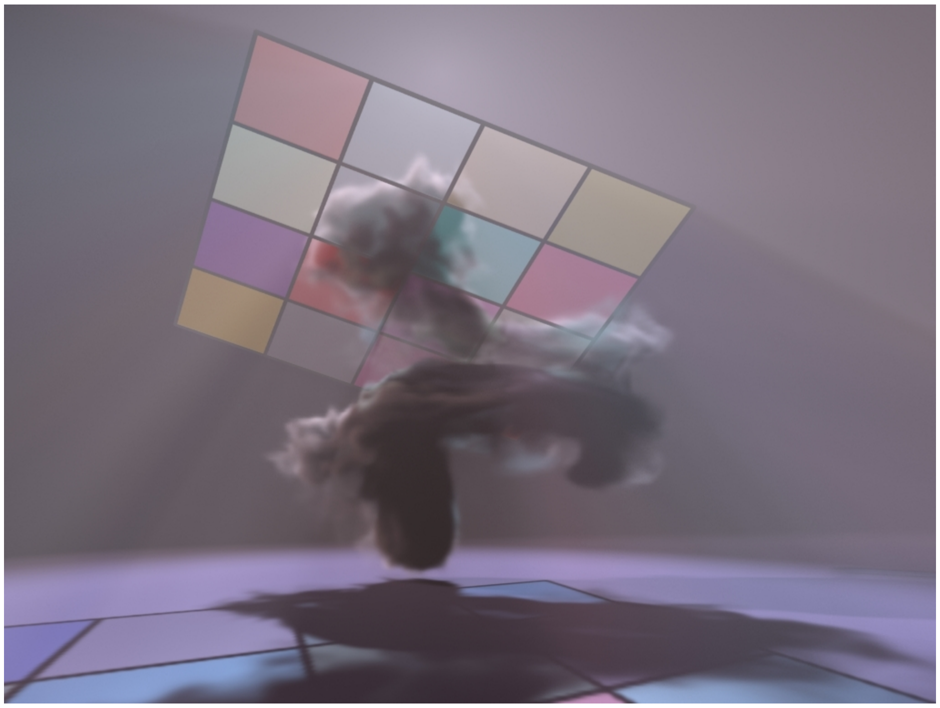


Figure 1: Volumes and surfaces

The following sections will discuss how each phase of rendering was extended to natively handle volume primitives.

2 Volume Geometry

2.1 Volume Procedural Interface

Volume primitives in Mantra are all defined through a simple, general interface based on point sampling and an optionally defined volume structure. The API comprises:

```
class Volume {
public:
    virtual void         evaluate(const Vector3 &pos,
                                  const Filter &filter,
                                  float radius, float time,
                                  int idx, float *data) const = 0;

    virtual void         getBoxes(RefArray<BoundingBox> &boxes,
                                  float radius,
                                  float dbound,
                                  float zerothreshold = 0) const = 0;

    virtual void         getAttributeBinding(StringArray &names,
                                             IntArray &sizes) const = 0;
}
```

The evaluate() function is provided with a position and channel index and is responsible for providing the filtered value of the specified field at that position. The getBoxes() operation is an optional operation that can be used to provide a list of boxes that describe the structure of the volume. Mantra will use these boxes to construct an acceleration data structure for empty space culling. The getAttributeBinding() operation provides a list of the channel names and vector sizes for the attributes defined by the volume. The binding produced by this interface is used by the VEX parameter binding algorithm to provide the shader with volumetric data matched by name and vector size.

2.2 Rendering Primitives for Volumes

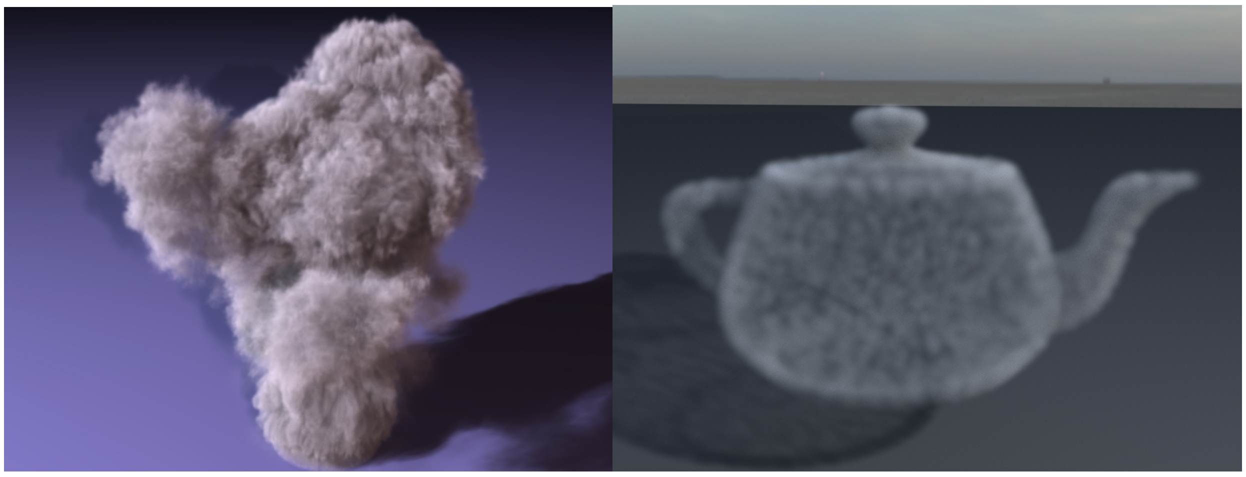


Figure 2: Voxel grid (left) and metaballs (right)

Default volume geometry types are provided in Mantra for voxel grids, metaballs, and shader-defined attributes through built-in volume primitives that implement the volume API.

Voxel grids are evaluated either using trilinear interpolation or using a filter kernel (such as a gaussian). The getBoxes() operation creates a box for each voxel, then recursively combines them using a min/max mipmap. This minimization of boxes helps to reduce the complexity of the acceleration data structure. Before boxes are created, the voxel data is filtered to account for the evaluation filter and displacement bound to ensure that all potential non-zero parts of the volume have coverage.

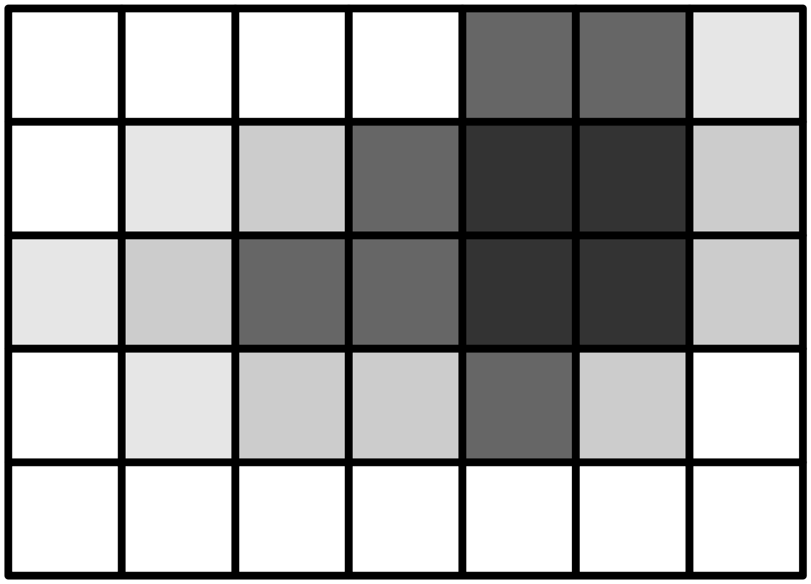


Figure 3: Example voxel field, with white empty voxels

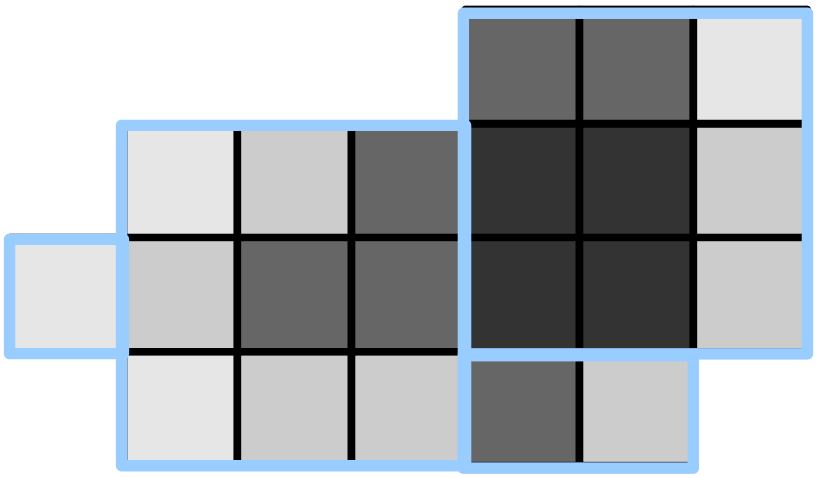


Figure 4: Possible clustering of 2D voxels (gray) into representative boxes (blue)

2.3 Voxel Grid Storage

Voxel grids are stored as a sequence of 16x16x16 tiles containing floating point data. Data within a tile is stored contiguously, while the entire voxel grid stores an index of the tiles. Tiles can independently apply a number of different compression techniques:

- Constant tile compression: if all values in a tile are the same value, the tile storage is compressed to a single value.
- Bit depth compression: if the range of the data in a tile is very small, a lossy compression algorithm can reduce storage by reducing the number of bits used to encode the data. A tolerance parameter controls how much compression can occur – so that adjacent tiles do not

show discontinuities. To ensure the compression is unbiased, an ordered dithering algorithm is used to eliminate bias and to allow reconstruction (via filtering) that preserves as much of the original data as possible. The images below show the same voxel data compressed to 5 bits per voxel and 1 bit per voxel.

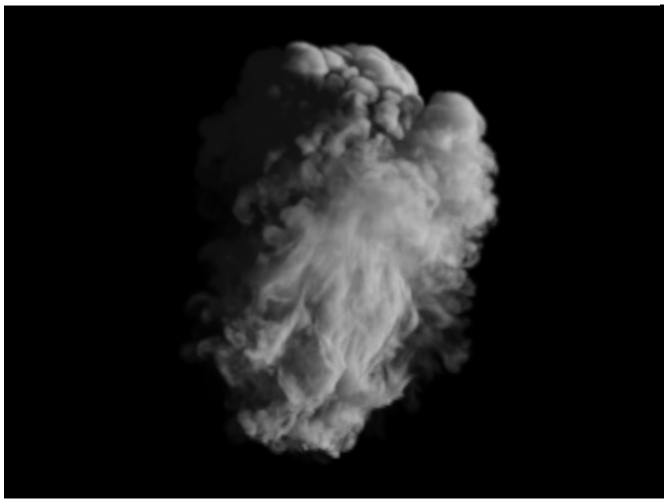


Figure 5: 5-bit ordered dither

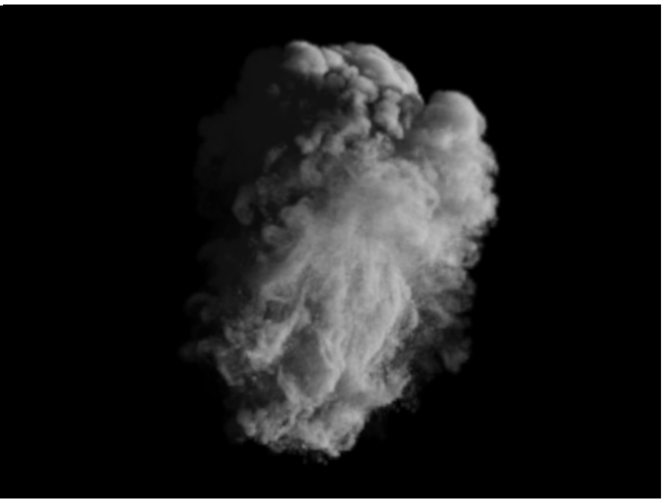


Figure 6: 1-bit ordered dither

It is important to choose a dithering pattern for bit depth compression that doesn't produce artifacts in renders, since a simple ordered dither will usually introduce aliasing artifacts when the volume is viewed along an axis or diagonal. A simple way to eliminate aliasing is to use a low discrepancy sequence for tiling. We've found that randomly swapped copies of the following sequence (for 2x2x2 ranges) eliminates aliasing while also providing a high quality reconstruction:

```
      float permute[] = {0,4,6,2,5,1,3,7}
```

Voxel grids are merged into higher-level volume primitives through a naming and vector consolidation operation. For example, a volume primitive may consist of 2 attributes (named “density” and “vel”) - though there would be 4 voxel grids comprising this primitive:

```
{
      name = “density”
      resolution = 128x128x128
      transform = ...
}
{
      name = “vel.x”
      resolution = 65x64x64
      transform = ...
}
{
      name = “vel.y”
      resolution = 64x65x64
      transform = ...
}
{
      name = “vel.z”
      resolution = 64x64x65
      transform = ...
}
```

Using separate fields for individual components of a vector field allows each component to use a different resolution and transform, at the expense of less efficient voxel value lookups (some index computation must be duplicated when performing a lookup into a vector field).

3 Rendering Algorithms

Mantra provides both a ray tracing and microvoxel rendering algorithm for volumes. Both share the same shading pipeline and use the same underlying volume data and volume acceleration data structure. This section will first discuss the general acceleration data structure used by both algorithms, and then discuss the details of the ray tracing and microvoxel engines.

3.1 Empty Space Culling

Volume primitives describe their coarse structure to the renderer through an API function that returns a list of boxes whose union encloses the volume to be rendered. These boxes are similar to the boxes that would be provided for polygons or other surface-based primitives, and are used to construct an acceleration data structure like the one used to accelerate surface ray tracing.

Mantra uses a KD-Tree to accelerate common operations related to volume rendering. These operations include:

- Testing whether a point or box is entirely outside the volume (box-in-tree)
- Calculating a sequence of entry and exit points for a ray that intersects the volume (ray tracing interval)

The construction algorithm for the KD-Tree is similar to the KD tree construction used for surfaces, and uses the surface area heuristic to optimize the splitting process (For construction algorithms, see [1]). To further improve performance of the interval calculation and occupancy test, partitions in the resulting KD-Tree are marked with a flag indicating whether the partition is fully enclosed by the boxes that it overlaps. Partitions that are fully enclosed can be used to provide a quick exit test for the box-in-tree query, and can be quickly skipped when calculating ray tracing intervals.

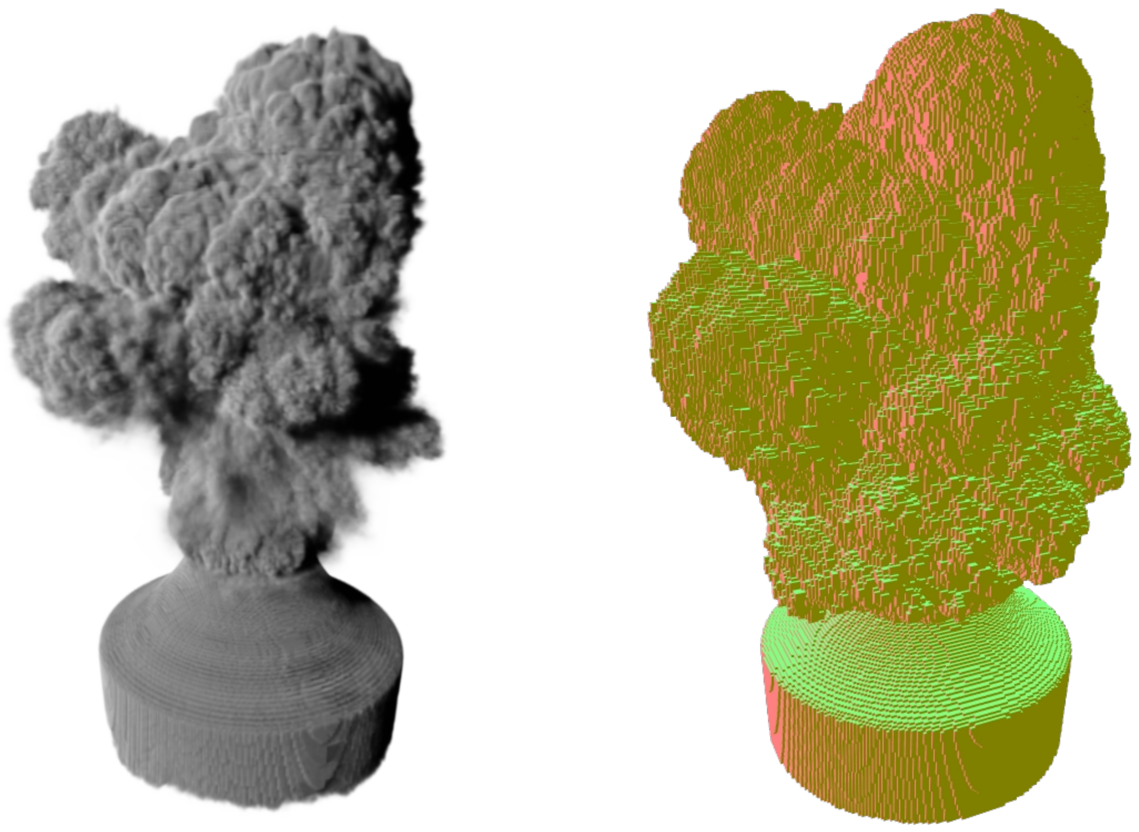


Figure 7: Volume with an axis-aligned isosurface

Another way to interpret the KD-Tree used for volumes is to think of it as defining an implicit isosurface over the volume, with axis-aligned boundaries. The fact that only axis-aligned boxes are stored in the tree makes it very fast for interval queries, since detailed intersection points with primitives such as triangles or quads are not necessary.

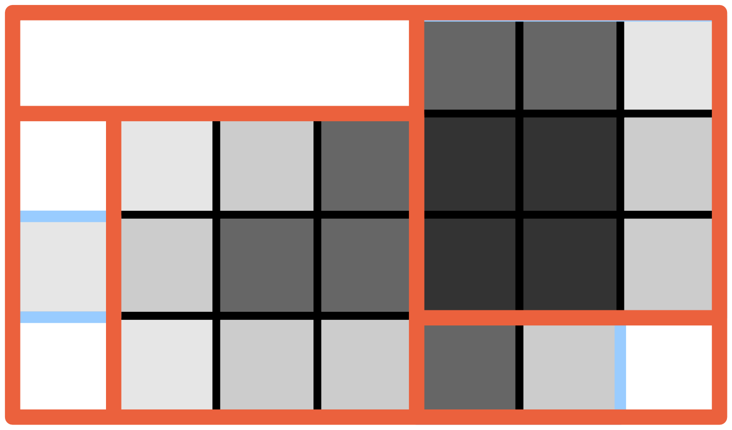


Figure 8: KD-Tree partitions (red) constructed on representative boxes (blue)

3.2 Ray Marching

Ray tracing of volume primitives is implemented with ray marching. When a volume primitive is hit by a ray, Mantra first calculates the intervals containing non-zero data using the volume's acceleration data structure. Then sample positions are generated in the interval using a user-defined volume step size and a sampling offset – determined from the sub-pixel sample or sampling identifier for secondary rays.

Often a volume primitive is large enough that if the renderer were to march through the entire volume, several thousand sample positions would need to be stored in memory. Mantra prevents this situation by limiting the maximum number of unshaded transparent samples (usually 32) before it requires that these samples are shaded. So for volumes that extend to infinity, Mantra will switch between sample generation (ray marching) and shading until it detects that the ray has become sufficiently opaque that new samples contribute little to the image. The figure below shows this effect – samples near the center of the image stop marching early since the ray march reaches full opacity. The red "halo" shows parts of the volume that don't quite reach the opacity threshold, causing the ray marcher to march all the way through the transparent back portions of the volume.

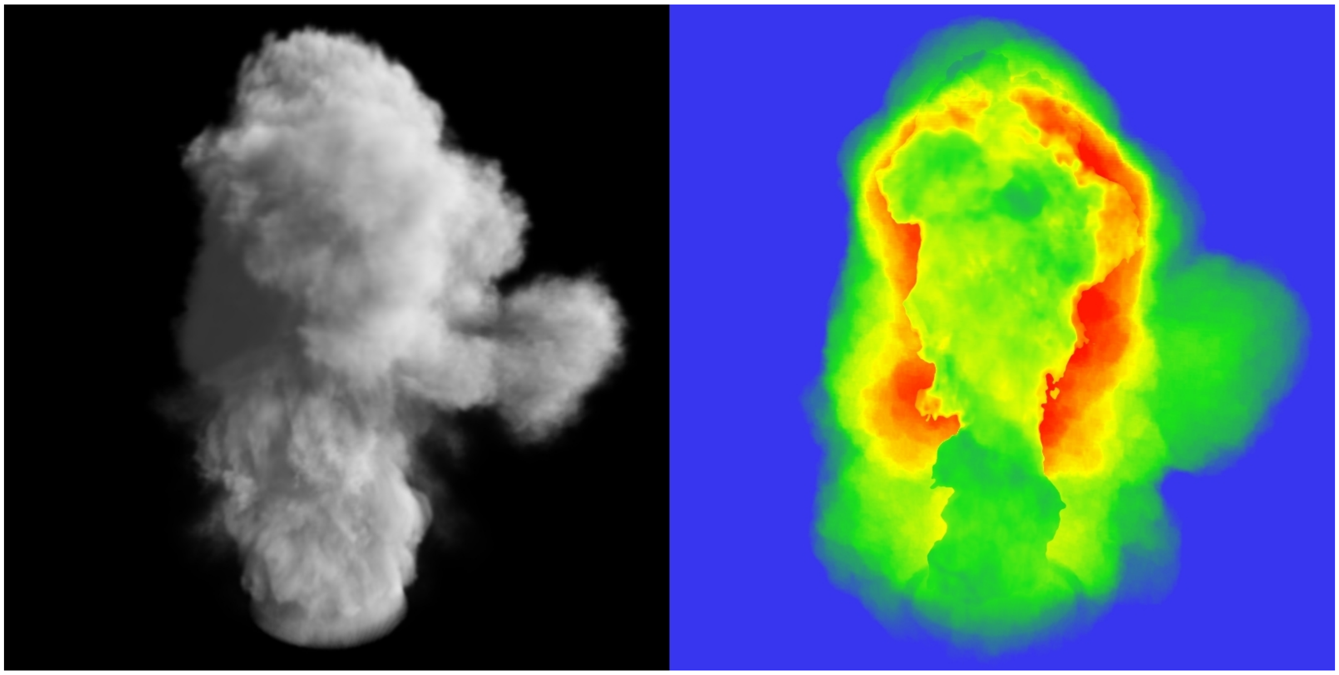


Figure 9: Volume primitive and false color image showing the relative number of steps (blue = 0 steps, red = 50 steps)

For volumes that are known to have a uniform density, samples can be distributed ideally based on the known density. Volume extinction follows an exponential falloff curve, meaning that if samples are distributed according to the exponential function they will each have equal weight. Samples can be distributed exponentially by inverting the exponential cumulative distribution function c(x):

$$\begin{aligned} p(x) &= \sigma_e e^{-x\sigma_e} \\ c(x) &= 1-e^{-x\sigma_e} \end{aligned}$$

Solving for distance:

$$x=\frac{-\ln(1-c(x))}{\sigma_e}$$

Equation 1: Volume sample distribution

Now randomly distributing c(x) between 0 and 1 produces sample distances that contribute equally to the image. The figure below shows plots of for several different densities.

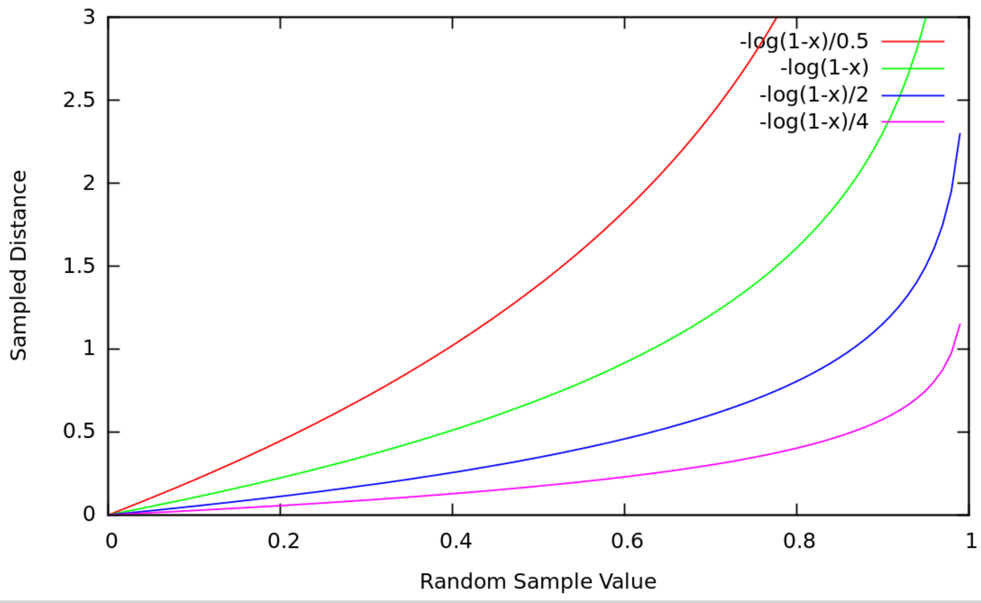


Figure 10: Uniform density sampling

An example of uniform sampling is shown below, where the sampling function above is used to place sample points in the interior of a surface. For wavelength-dependent absorption, a different sampling curve can be used for each color – with components weighted using multiple importance sampling [5].

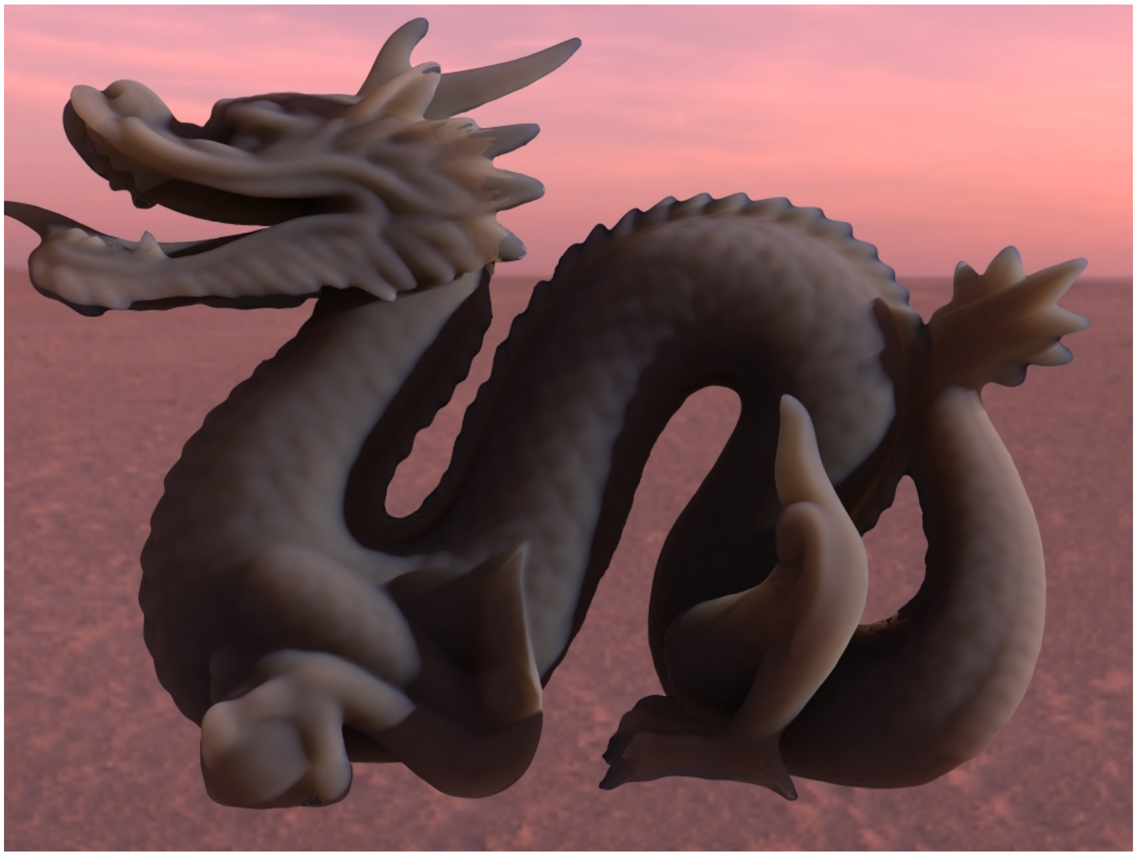


Figure 11: Single scattering in a uniform density volume can be used to simulate subsurface scattering

3.3 Microvoxels

Micropolygon rendering of volume primitives is implemented using an extension of REYES architecture [3] to handle volumetric data (originally presented in [4]). Fundamentally, a micropolygon renderer is responsible for 2 specific tasks – dicing and shading. The dicing algorithm splits up complex primitives into simpler ones, while the shading algorithm executes a programmable shader on a generated set of shading points. The same shading algorithm is used with ray tracing, but the shading points differ – in the case of micropolygon rendering, shaders are normally executed on subdivided meshes.

For volumes, subdivision must generate 3D space-filling primitives rather than flat shading meshes. Mantra uses a 3D binary subdivision scheme followed by generation of blocks of up to 256 shading points in the form of 3D grids.

3.3.1 Splitting and Measuring

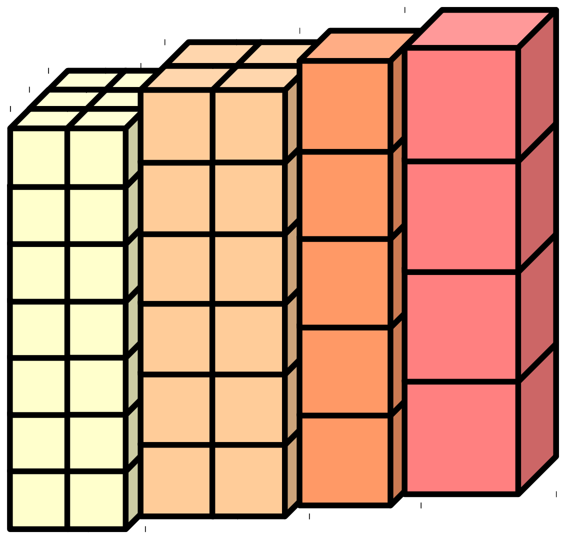


Figure 12: View-dependent dicing - viewer on left

A micropolygon renderer makes use of one or more metrics to calculate the size of primitives in order to decide how finely or coarsely the primitive must be subdivided before it is small enough to shade. This subdivision metric is used to make decisions about when a primitive should be split and also to determine the number of shading points that are required once it is known that subdivision has reached the limit. The metrics used for measuring the size of a primitive include a screen-space metric along with other constraints such as a depth (z-axis) measuring quality, and user-configurable parameter to control the overall scale of the subdivision.

For subdivision of volumes, Mantra uses a simple metric that is based on the pixel size and the distance of the voxel centroid to the viewer:

```
float
VRAY_Measure::getDotArea(const Vector3 &pt) const
{
    // pixel_xsize and pixel_ysize are precomputed based on the camera's view
    return pt.length2() * pixel_xsize * pixel_ysize;
}
```

3.3.2 Shading Grid Generation

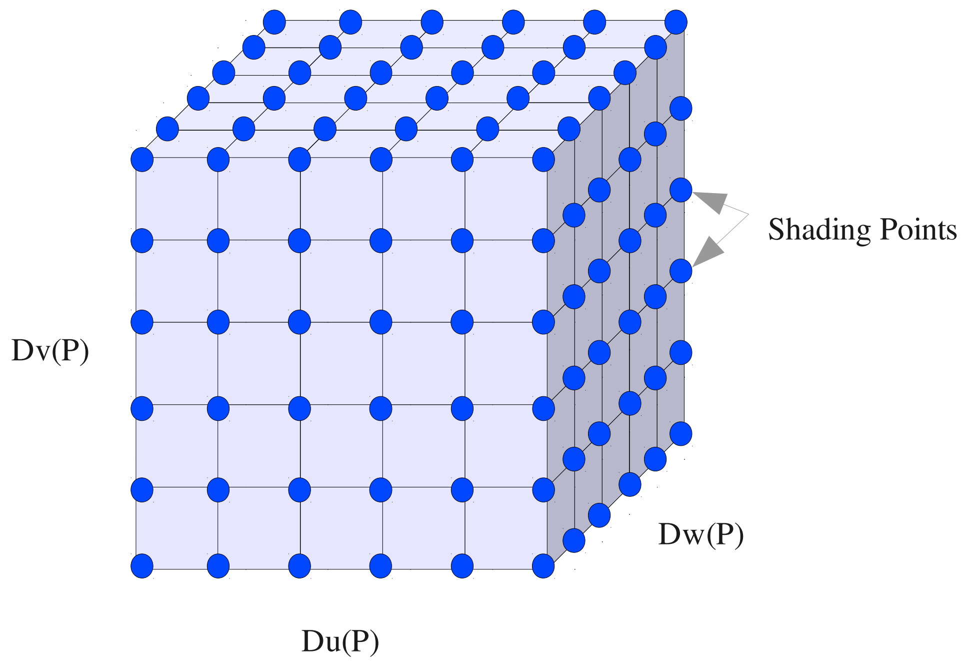


Figure 13: Cube of 216 shading points

Shading cubes are created when the splitting/measuring process determines that the volume is small enough (based on the metric used for measuring) that it can be shaded. Mantra uses a fixed maximum number shading points (usually 256) per cube to help take advantage of batch shading (shading many points with an individual shader invocation). The 3D structure used for volumes always uses inclusive shading point placement, so 2x2x2 voxel grids shade a total of 8 times. Inclusive shading point positioning allows Mantra to use linear interpolation within a shaded grid without the need to synchronize between neighboring grids.

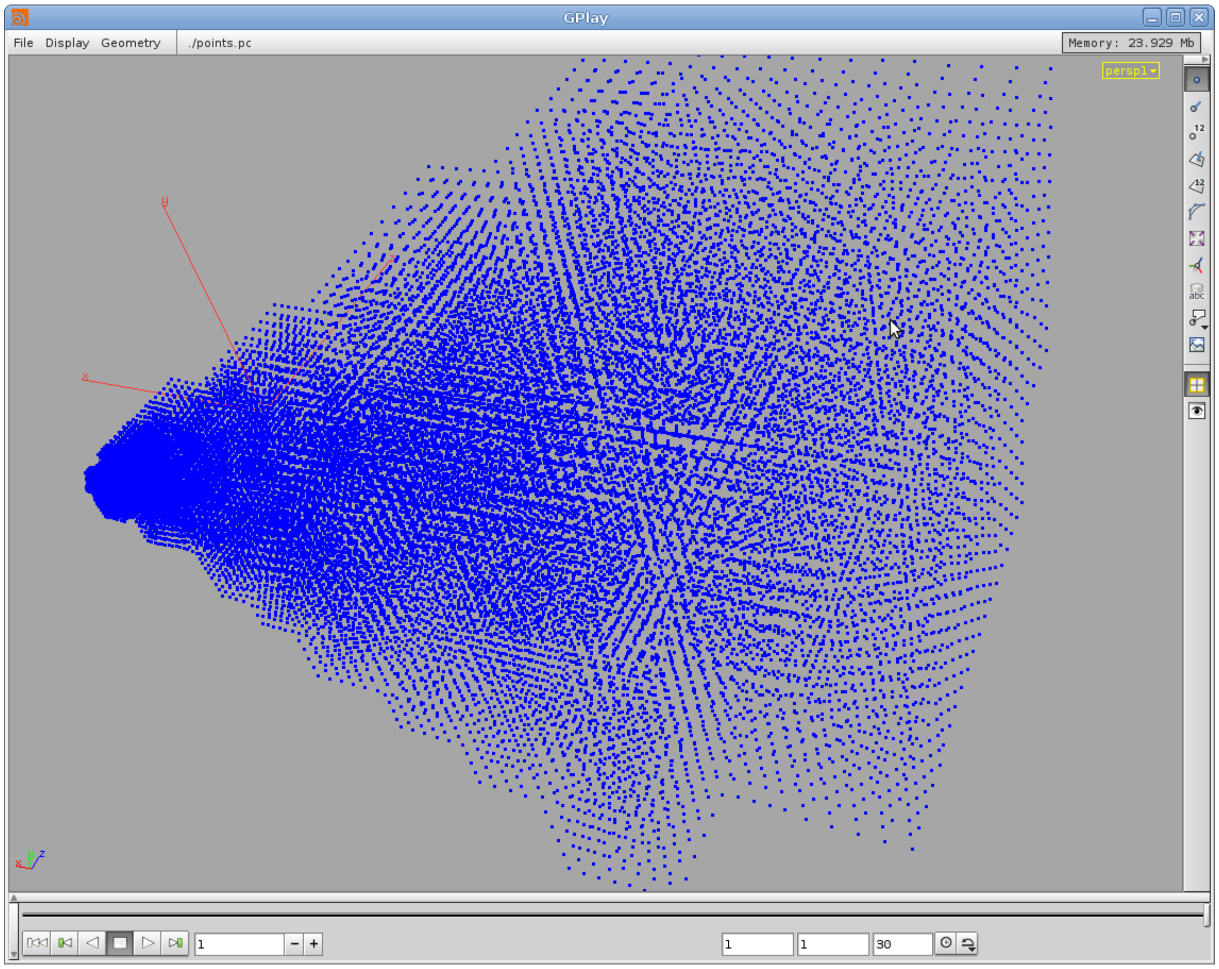


Figure 14: Actual cubes of shading points generated by a render. The viewing frustum originates on the left of the image

3.3.3 Sampling

Mantra uses a fixed pool of sampling patterns for sampling of all features in a render. Sampling patterns use a resolution that is determined by the number of pixel samples in the render – specified as the product of 2 values (eg. Pixel samples of 3x3 results in 9 pixel samples). Sampling patterns are used to distribute x/y pixel anti-aliasing samples, time, depth of field, and an arbitrary number of shader-required sampling parameters between the sub-pixel samples within a pixel.

For volume rendering, Mantra performs an independent ray march for each sub-pixel sample. One additional sampling parameter is used for volumes to control the ray marching offset.

The same random offset is used for the entire ray march. This means that all volume samples will be placed in an identical location in camera space regardless of how the volume is transformed or how the camera moves in the scene. Other approaches to sample distribution include stratified sampling (a different random offset is used for each ray march sample) and equidistant sampling (no offset is used). Equidistant sampling produces unavoidable aliasing errors and stratified sampling will usually lead to a doubling of the amount of noise in the render while only enhancing accuracy when a single ray march is performed per pixel. See [2] for details on these different approaches to sample placement.

The following images show a motion blurred volume – first with fully filtered pixels (as would normally be produced by the renderer) and then expanded to map the individual sub-pixel samples to pixels. The sub-pixel rendering shows the structure of the sampling pattern – the correlated time and ray march offsets within individual pixels show up as patterns in the image.

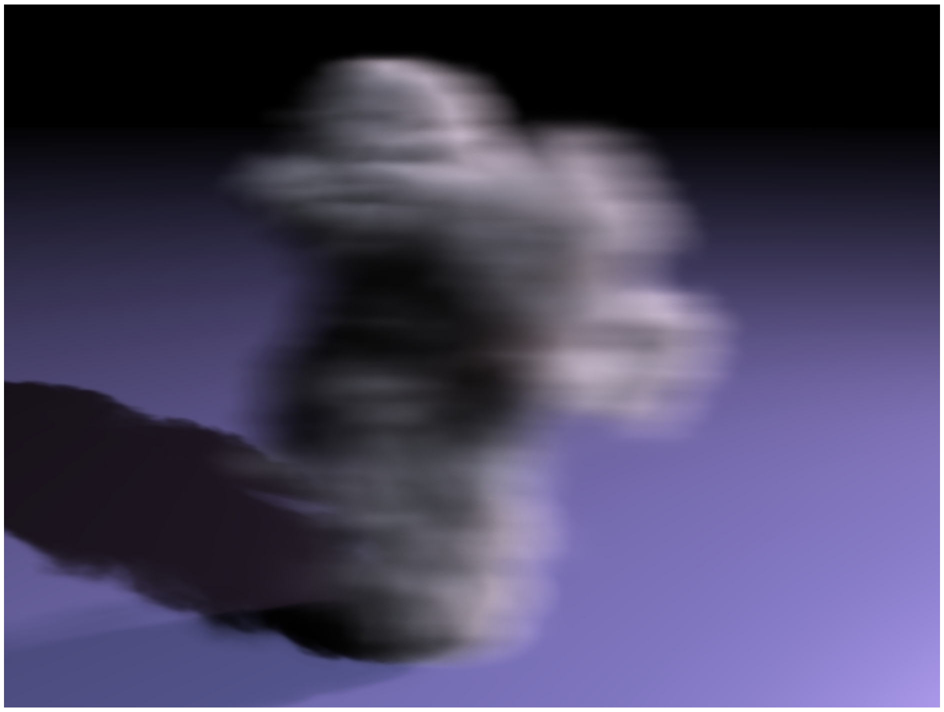


Figure 15: Render of a motion blurred volume

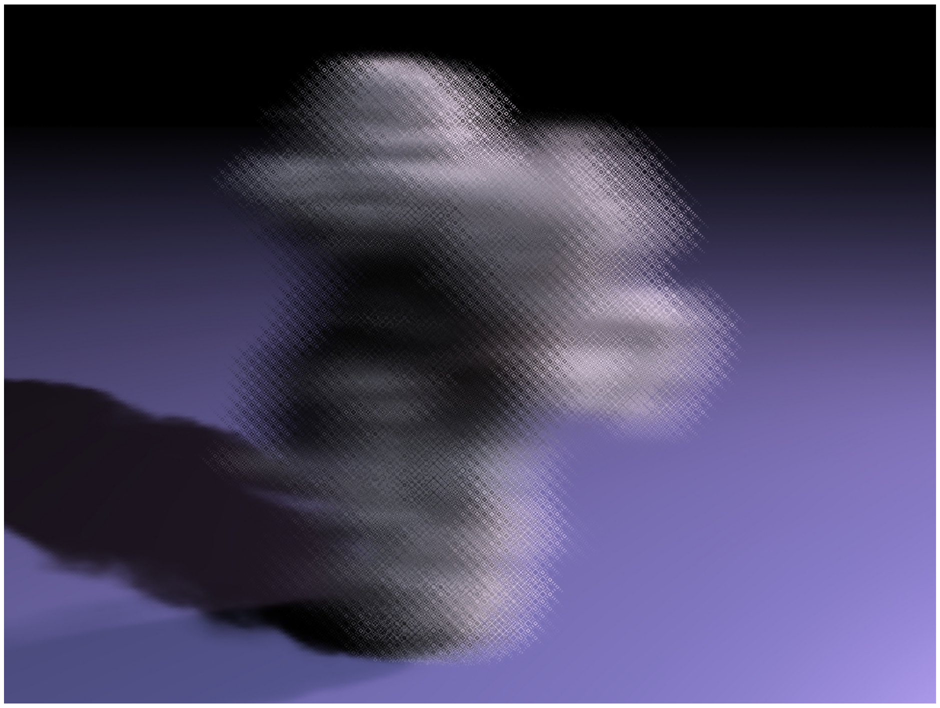


Figure 16: Sub-pixel sampling patterns for a motion blurred volume, 16x16 pixel samples

3.3.4 Motion Blur

For both ray tracing and for microvoxels, motion blur works by distributing different random sample times to individual pixel samples. In the case of ray tracing, Mantra will shade each point along each sub-pixel ray march. With microvoxels, sampling and shading are fully decoupled – allowing shaders to execute on shading grids while sampling still operates on sub-pixel ray march samples. For this reason, the sampling patterns used for rendering microvoxels are identical to those that would be used for sample placement in ray traced volumes.

Shading is often the most expensive part of the rendering pipeline - involving complex operations such as shadow map lookups, ray tracing, and texturing. Decoupling shading from sampling makes it possible to independently adjust the amount of shading, so that volumes that contain very uniform shading but exhibit a large amount of motion can focus the rendering algorithm on the component that matters most – antialiasing the motion blur. Given shaded microvoxels, Mantra only needs to perform trilinear interpolation to obtain the shaded values at each point along a ray march – significantly reducing the complexity of the ray marching algorithm.

```
void Sampler::sampleVoxelScalar(const Ray &ray, Primitive *prim)
{
    const int               map[6][4] = {
                                {0, 1, 3, 2}, // zmin
                                {4, 5, 7, 6}, // zmax
                                {0, 2, 6, 4}, // xmin
                                {1, 3, 7, 5}, // xmax
```

```
                            {0, 1, 5, 4}, // ymin
                            {2, 3, 7, 6}  // ymax
                        };
    Vector3     p[8];
    float       hit[2];
    float       zoff, stepsize;
    int         hitcount = 0;

    stepsize = prim->getStepSize();
    prim->getVoxelCorners(p);
    for (i = 0; i < 6; i++)
    {
        p0 = p[map[j][0];
        p1 = p[map[j][0];
        p2 = p[map[j][0];
        p3 = p[map[j][0];

        // Find the intersection distance and parameters
        if (Quad::intersect(t, u, v, ray, p0, p1, p2, p3))
        {
            hit[hitcount] = t;
            hitcount++;
        }
    }

    if (hitcount == 2)
    {
        zoff = getSamplingOffset(ray);
        zoff += floor((hit[0] / stepsize) - zoff) + 1;
        zoff *= stepsize;
        if (zoff >= hit[0] && zoff < hit[1])
        {
            // Iterates through the interval and adds ray march samples
            while (zoff < hit[1])
            {
                processSample(tval, prim);
                tval += stepsize;
            }
        }
    }
}
```

Ray marching against microvoxels uses a ray tracing algorithm. Given the positions of the voxel vertices, each of the 6 voxel faces are interpreted as polygons. If motion blur is used, the vertex positions are moved to the sample time. Moving voxels may undergo non-rigid transformations (as shown in the figure below), so they may not be rectangular in shape. Then, ray tracing is performed against the 6 faces, with a simple bounding optimization to avoid unnecessary intersections. Finally, the entry and exit positions are analyzed to determine the ray march interval and samples are placed within the voxel to satisfy the known ray march offset and ray position and direction. The fundamental algorithm is shown above in sampleVoxelScalar().

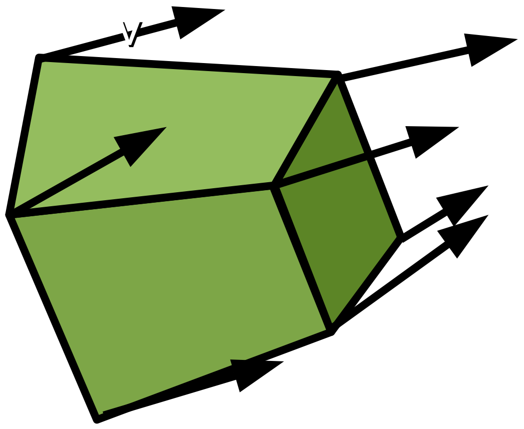


Figure 17: Velocity blurred voxel

Mantra will often have a list of rays available that require intersection against the same voxel. The algorithm above is trivially parallelized by intersecting many rays against the voxel using SIMD instructions.

3.4 Screen Door Transparency

Some rendering algorithms are unable to deal directly with the transparent lists of samples produced by ray marching. For example, path tracing and photon map generation both require fully opaque samples in combination with a phase function to determine scattering effects. To accommodate these algorithms, screen door transparency is used.

Section 3.2 provided a sampling function to distribute samples in a uniform density volume. Extending this algorithm to volumes with varying density is possible in several different ways (with varying memory and performance trade-offs). The approach used by mantra is to use the transparent samples generated by ray marching, but to discard samples until the threshold in Equation 1 is reached for a random sample value. In order to minimize wasted effort with this approach, it is important to ensure that the computation of opacity can be separated from the execution of the full shader – since several opacity evaluations are required before a single (unbiased) opaque sample is produced.

4 Shading Algorithm

Shading of volumes in Mantra operates as a simple extension to surface shading. So to shade volumes, the surface context is used with a few simple adjustments to inform the shader that it is rendering a volume and to provide information that is only meaningful when shading volumes.

Mantra uses the same shader for volume rendering regardless of whether the volume was rendered using microvoxels or with ray tracing. In general, shaders are designed to be rendering algorithm agnostic – meaning that the renderer has some flexibility in how shading can be performed. Internally, the renderer manages different shading contexts (including a ray tracing and a micropolygon shading context) which both eventually call the same shader.

Multithreaded shading is supported through a duplicate shading state for each thread. Different threads can execute different shaders in parallel, provided that the shaded data does not involve dependencies. Dependencies in microvoxel rendering can occur when independent tiles require the result of shading for the same microvoxel grid – which can commonly occur when the tiles are adjacent and a grid spans multiple tiles. The ray tracing renderer exhibits fewer data dependencies but is usually costlier to render due to the increased amount of shading.

4.1 Parameter Binding

The following shader shows a simple VEX shader interface with parameters.

```
surface
volumecloud(float density = 1;
        vector diff=1;
        vector Cd = 1;
        float clouddensity = 50;
        float shadowdensity = 50;
        float phase = 0;
        int receiveshadows = 1)
{
    …
}
```

Parameters are bound to attributes in a volume primitive by name. So if the volume contains a field named “temperature” and the shader has a parameter called “temperature”, the shader will be initialized with values from the volume field when it is executed. Bindings for vector attributes such as “vel” use the same approach, but the vector size for the attribute must match the shader parameter type. The binding of volume attributes to shader parameters involves evaluation of the volume attribute at the position of the shading point. Evaluation uses a user-specified filter (for example, a box or gaussian) that is used to reconstruct volume data from sparse representations such as voxel grids.

Shader parameters that are bound to volume primitive attributes are identified when the shader is loaded. If the volume primitive does not contain an attribute for a given shader parameter, the shader parameter is eliminated by the optimizer and replaced by a constant – leading to improved shader execution performance.

4.2 Shading Context

Volumes execute within the surface context with a few minor extensions:

- A global variable, dPdz, indicates the distance within the volume that should be composited by the shader.
- A derivative function, Dw(), allows derivative computation along the third volumetric direction. Additionally, the volume around the shading point is available with the volume() operation.
- Normals (the N and Ng global variables) are initialized with the volume gradient. If the volume

primitive does not support a gradient operation, the gradient is estimated with sampling.

Mantra's shading pipeline includes a shadow context which is executed for lights with shadows enabled. Volume renders commonly use deep shadow maps, which eliminate the need to perform a costly ray march for visibility computation.

4.3 Shader Writing

An example of a simple, general purpose volume shader (for smoke and clouds) is shown below.

```
surface
volumecloud(float density = 1;
        vector diff=1;
        vector Cd = 1;
        float clouddensity = 50;
        float shadowdensity = 50;
        float phase = 0;
        int receiveshadows = 1)
{
    vector      clr;
    float       den = density;

    if (density > 0)
    {
        clr = 0;

        // Accumulate light from all directions.
        illuminance(P, {0, 1, 0}, PI)
        {
            if (receiveshadows)
                shadow(Cl);
            clr += Cl;
        }

        // Allow for different densities for shadowed and non-shadowed
        // surfaces.
        if (isshadowray())
            den *= shadowdensity;
        else
            den *= clouddensity;

        // Clamp the density
        den = max(den, 0);
        Of = (1 - exp(-den*dPdz));

        Cf = Of * Cd * diff;
        Cf *= clr;
    }
    else
    {
        Of = 0;
        Cf = 0;
    }
    // Physically based rendering phase function
    if (phase == 0)
        F = diff * Cd * isotropic();
    else
        F = diff * Cd * henyeygreenstein(phase);
}
```

There are a few different components that comprise this shader. First, there are the parameters to the shader. The “density” parameter is a well-known attribute name, and will usually bind to the “density” field in a volume primitive. The other parameters such as “clouddensity” and “phase” will often be constant throughout a volume, and so will evaluate to constant values when the shader is executed. The values for these parameters are mapped to parameters that exist on the shader node.

Within the body of the shader, there is a lighting computation (inside the illuminance loop), which loops over the light sources in the scene and evaluates the product of the lighting and the phase function – in this case, an isotropic bsdf. At the end of the shader, Mantra also returns an explicit representation of the phase function that is available for use in physically based lighting simulations using this volume.

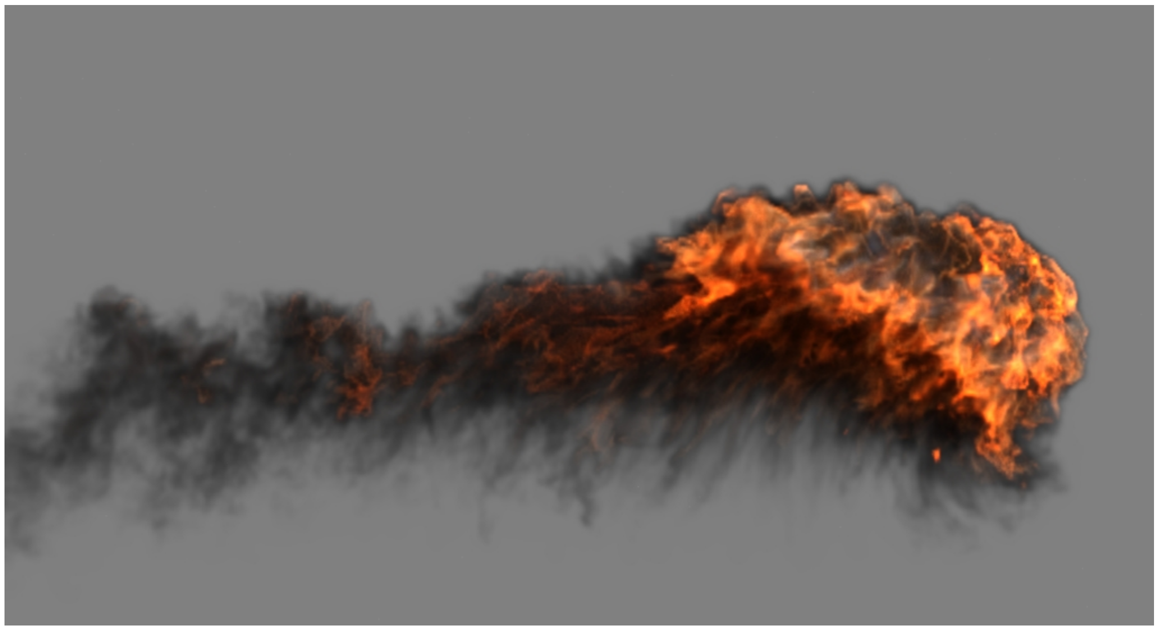
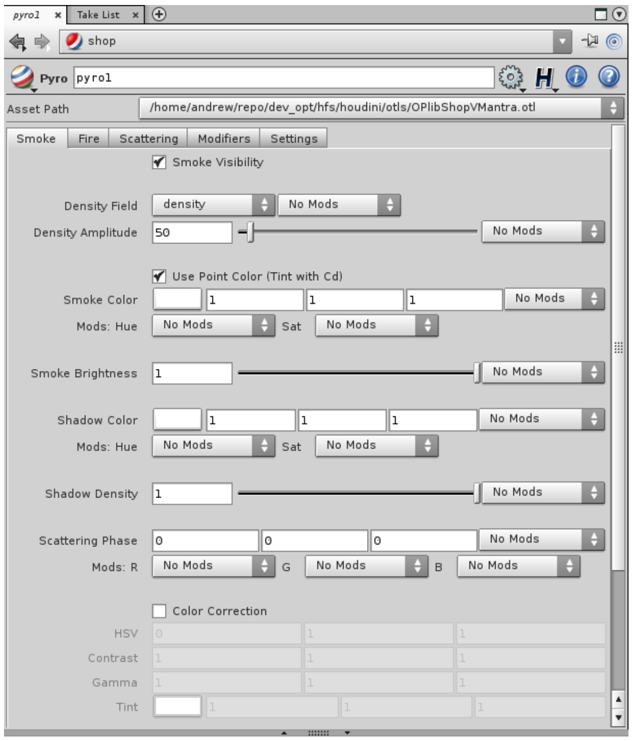


Figure 18: Fireball effect generated with “Pyro” shader

Houdini includes a more robust and complete volume rendering shader intended for use with fluid simulations. An example of the kind of effect that can be rendered using this shader is shown above. Some of the user interface features available in this shader are shown below. Parameters in the user interface map directly to shader parameters like those shown in the volumecloud shader above.



pyro1 | Take List

shop

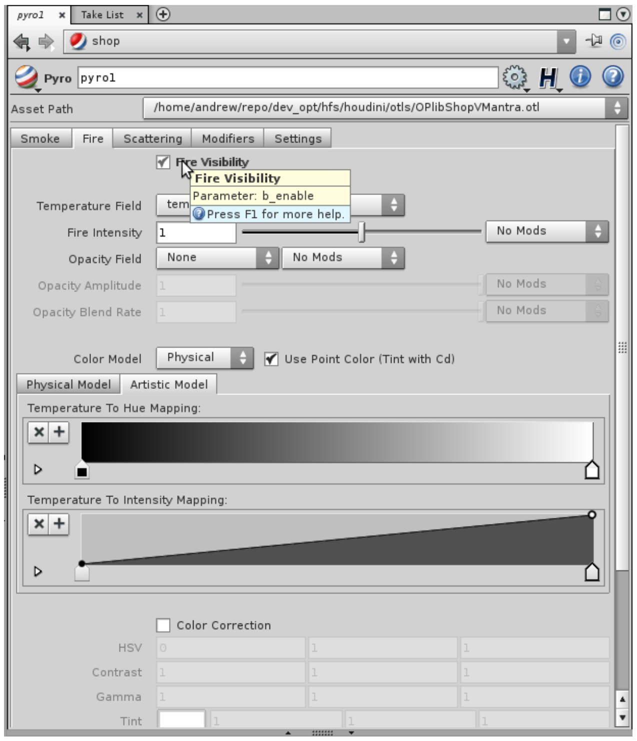
Pyro pyro1

Asset Path /home/andrew/repo/dev_opt/hfs/houdini/otls/OPlibShopVMantra.otl

Smoke | Fire | Scattering | Modifiers | Settings

- [✓] Smoke Visibility
- Density Field: density — No Mods
- Density Amplitude: 50 — No Mods
- [✓] Use Point Color (Tint with Cd)
- Smoke Color: 1 1 1 — No Mods
- Mods: Hue: No Mods — Sat: No Mods
- Smoke Brightness: 1 — No Mods
- Shadow Color: 1 1 1 — No Mods
- Mods: Hue: No Mods — Sat: No Mods
- Shadow Density: 1 — No Mods
- Scattering Phase: 0 0 0 — No Mods
- Mods: R: No Mods — G: No Mods — B: No Mods
- [ ] Color Correction
- HSV: 0 1 1
- Contrast: 1 1 1
- Gamma: 1 1 1
- Tint: 1 1 1

Figure 19: Pyro smoke tab - choose the fields to use for rendering smoke and configure modifiers



pyro1 | Take List

shop

Pyro pyro1

Asset Path /home/andrew/repo/dev_opt/hfs/houdini/otls/OPlibShopVMantra.otl

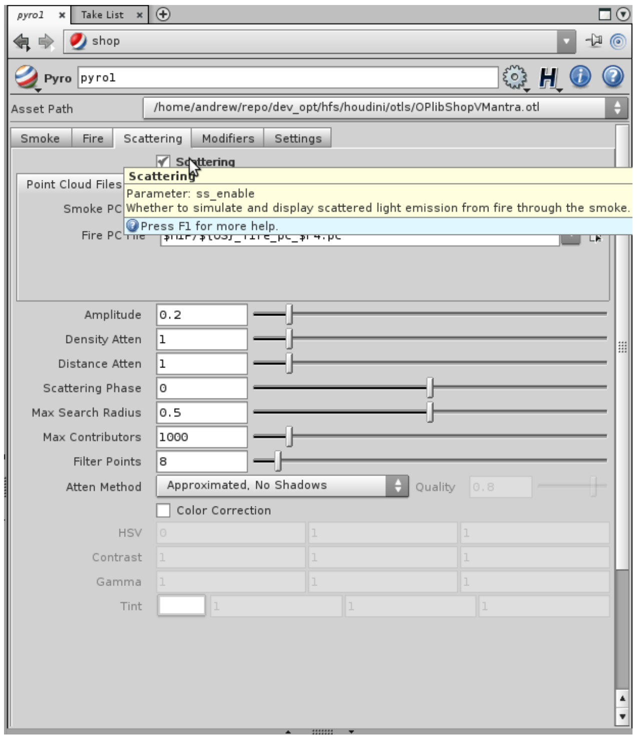
Smoke | Fire | Scattering | Modifiers | Settings

- [✓] Fire Visibility
  - Fire Visibility — Parameter: b_enable — Press F1 for more help.
- Temperature Field: tem…
- Fire Intensity: 1 — No Mods
- Opacity Field: None — No Mods
- Opacity Amplitude: 1 — No Mods
- Opacity Blend Rate: 1 — No Mods
- Color Model: Physical — [✓] Use Point Color (Tint with Cd)

Physical Model | Artistic Model

- Temperature To Hue Mapping:
- Temperature To Intensity Mapping:
- [ ] Color Correction
- HSV: 0 1 1
- Contrast: 1 1 1
- Gamma: 1 1 1
- Tint: 1 1 1

Figure 20: Pyro fire tab - choose the fields to use for rendering fire (emission) and configure modifiers



Pyro pyro1

Asset Path /home/andrew/repo/dev_opt/hfs/houdini/otls/OPlibShopVMantra.otl

✓ Scattering

Scattering
Parameter: ss_enable
Whether to simulate and display scattered light emission from fire through the smoke.
Press F1 for more help.

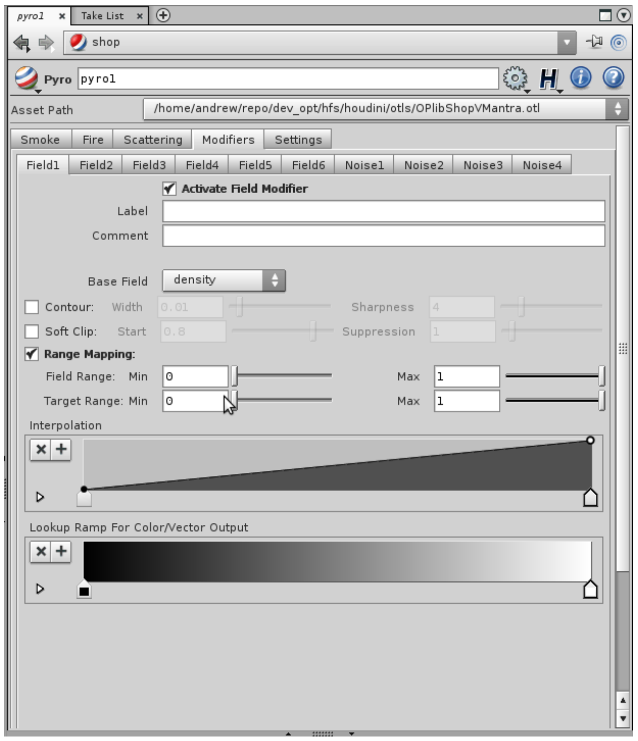
Point Cloud Files
Smoke PC
Fire PC File $HIP/${OS}_fire_pc_$F4.pc

Amplitude	0.2
Density Atten	1
Distance Atten	1
Scattering Phase	0
Max Search Radius	0.5
Max Contributors	1000
Filter Points	8
Atten Method	Approximated, No Shadows
Quality	0.8

☐ Color Correction

HSV	0	1	1	
Contrast	1	1	1	
Gamma	1	1	1	
Tint		1	1	1

Figure 21: Pyro scattering tab - configures multiple scattering calculation using point clouds



Pyro pyro1

Asset Path /home/andrew/repo/dev_opt/hfs/houdini/otls/OPlibShopVMantra.otl

✓ Activate Field Modifier

Label
Comment

Base Field density

☐ Contour: Width 0.01 Sharpness 4
☐ Soft Clip: Start 0.8 Suppression 1
✓ Range Mapping:
Field Range: Min 0 Max 1
Target Range: Min 0 Max 1

Figure 22: Pyro modifiers - configures modifiers (for example, noise) that can be used to adjust fields for rendering

5 References

[1] Vlastimil Havran. Heuristic Ray Shooting Algorithms (2000). Ph.D. Thesis

[2] Pauly, M., Kollig, T., and Keller , A. Metropolis light transport for participating media (2000). In EGRW ’02, 11–22.

[3] Cook, R. L., Carpenter, L., and Catmull, E. The reyes image rendering architecture (1987). In SIGGRAPH Comput. Graph. *21*, 4, 95-102.

[4] Clinton, A., Elendt, M. Rendering volumes with microvoxels (1999). SIGGRAPH 2009 Talks.

[5] Veach, E., Guibas, J. Optimally combining sampling techniques for monte carlo rendering (1995). In SIGGRAPH 95 proceedings, 419-428

Volume Rendering at Sony Pictures Imageworks

SIGGRAPH 2011 Course Notes: Production Volume Rendering 2

Magnus Wrenninge[1]

Updated: 6 aug 2011

[1] magnus.wrenninge@gmail.com

1. Table of contents

2. History and overview

The volume rendering pipeline at Imageworks has been developed over the last 6 years. Originally a set of Houdini plugins for modeling and rendering, it has grown into a larger tool set that integrates volumetrics-related tasks from modeling and processing to simulation and rendering.

An important part of the pipeline is the Field3D library, which was released under an open source license in 2009. Field3D is the foundation of all the volumetric tools and provides the glue that lets each tool communicate with the others – both through .F3D files and directly in-memory using the `Field3D::Field` data structures.

To provide an overview of the development of the tools, the following is a rough history:

2005-2007 : SPIDERMAN 3
: Svea was written to model and render dust and distant sand effects. Version 1 of FieldTools were developed as part of the Sandstorm tool set which was used to model, simulate and render Sandman. At this point an in-house file format based on GTO was used (called ISTOR).

2006–2007 : BEOWULF
: Fire shading and rendering capabilities added to Svea. Support for the previous fire pipeline's file format (.CACHE) was added, and was used to bring Maya Fluids caches into Svea for rendering. In order to handle the sharp features of fire, an adaptive raymarch scheme was implemented. Camera and deformation motion blur for procedural volumes was implemented using ray differentials to determine sample density.

2007–2008 : HANCOCK
: A Python interface was added to Svea in order to integrate it with the in-house lighting software (Katana). A sparse field file format and data structure was added to Svea in order to handle the high resolution voxel buffers used to model the tornadoes. An extension to the holdout algorithm allowed volumetric holdouts to be used so that each of the six tornado layers could be broken out and manipulated separately in compositing. Multiple scattering was implemented to create realistic light bleeding effects from lightning strikes inside tornadoes.

2008–2009 : FIELD3D
: The Field3D library was developed in an effort to unify the volumetric-related tools. Svea was updated to use the Field3D data structures as its native format for voxel buffers. The FieldTools were re-written (version 2) to support Field3D data structures natively in the Houdini node graph, allowing very high resolution fields to be manipulated interactively. A FLIP-based gas solver, a GPU-accelerated SPH liquid solver, and a suite of fluid-related vector field manipulation tools became the FluidTools package.

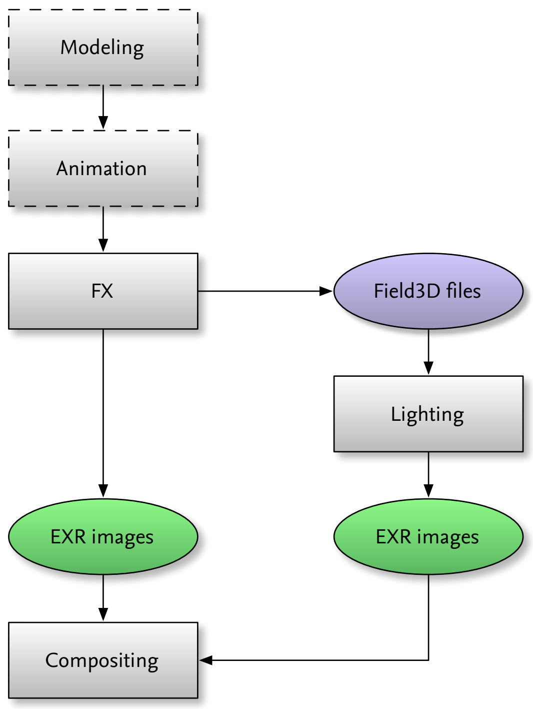
2009–2010 : ALICE IN WONDERLAND
: A new particle-based advection scheme was added to the gas solver which allowed decoupling of the density simulation field from the underlying velocity field while still remaining coupled in terms of affecting the behavior of the simulation. Svea was updated with new empty space-optimizations to render the sparse but high-resolution (often over 4000^3) voxel buffers efficiently. On the pipeline

side, the interface to Svea in the lighting package (Katana) was improved so that the full pipeline could be accessed and controlled both through Houdini and the Python API.

2010–2011 : Green Lantern
A new blackbody-based shader was introduced for rendering fire and explosions, and a new rasterization primitive type was added which uses existing voxel buffers as its base. This technique for stamping existing buffers during the modeling stage was used to build the sun at the end of the film, and also for many of the atmospheric elements at Planet Oa.

All in all, the tools have been used in: *Spiderman 3, Beowulf, Surf's Up, Hancock, Speed Racer, Valkyrie, Body of Lies, G-Force, Watchmen, Cloudy With a Chance of Meatballs, Alice In Wonderland, The Smurfs, Arthur Christmas* and *Green Lantern.*

2.1. Pipeline overview



Flow of data between departments

Seen at the broadest scale, volume modeling and rendering is accomplished in the effects and lighting departments. For certain types of elements, the effects department will handle both modeling and rendering/lighting of volumetric elements. In other cases effects only handle modeling of the volumetrics and pass off Field3D files to the lighting department for final rendering. In both cases final rendering is done using Svea, outputting one or more EXR files as the final product.

2.2. The volumetrics toolkit

The tools used to create volumetric effects at Imageworks fall into four basic categories (volume modeling, processing, simulation and rendering) and are accomplished by three different modules.

2.2.1. FieldTools & FluidTools

For interactive processing of voxelized data we use a suite of Houdini plugins called FieldTools. These offer a parallel workflow to Houdini's own volume primitives, and are used to connect and manipulate all field- and voxel-related data in the effects pipeline. The tools perform tasks such as field creation, level set conversion, calculus and compositing operations, attribute transfer from/to geometry, etc. Some of the main features are:

- **Support for multiple data structures**. The FieldTools use the data structures from Field3D as their internal representation, and any subclass of `Field3D::Field` may be passed through the node graph. This lets the user mix densely allocated and sparse fields, as well as MAC fields (used in fluid simulation) in the same processing operation.
- **Resolution, mapping, data structure and bit depth independent processing of fields**. Operations that apply to multiple fields can mix resolutions arbitrarily, and also lets the user mix fields of different mappings (i.e. different transforms). Most algorithms are optimized for certain data structure combinations (such as dense-dense, sparse-sparse), but fall back to generic code in order to support all data structures. Because Field3D's I/O classes support multiple bit depths, the user can also freely use any combination of half/float/double to save memory or provide extra precision where needed.
- **Field3D fields flow in their native form through the Houdini node graph**. Using a shared memory-manager, FieldTools, FluidTools (and Svea) can pass *live fields* (Field3D fields in the Houdini node graph) between each other with no copying or other overhead.
- **Multi-layer support**. Any number of fields may flow through a single node connection, for example when converting complex geometry into multiple separate level sets and velocity fields.
- **Procedural and boundless fields**. Procedural fields can co-exist with voxelized fields, and can both be used interchangeably. Procedural fields such as Perlin noise or Curl noise can be sampled at any point in world space (i.e. even outside their designated *bounds*), but still respond to transformation operations such as translation, scale and rotation.
- **Hardware acceleration**. Most of the voxel processing tools (resizing/resampling, blurring, distortion) are implemented in both as CUDA-optimized and multithreaded CPU code, taking advantage of hardware acceleration where available, but falling back on a CPU version that produces identical results if no compatible graphics card is available.

For liquid and gas simulation Imageworks uses a proprietary framework that connects directly to the FieldTools in order to input live Field3D fields from the Houdini node graph into the fluid simulators, and to give the rest of the system direct access to the simulator's internal simulation fields. The gas simulator is based on the FLIP algorithm and uses a hybrid particle/field-based advection algorithm with very low numerical dissipation. Two liquid simulation schemes are used – one FLIP-based for voxelized liquids, and a GPU-accelerated SPH solver for particle-based fluids.

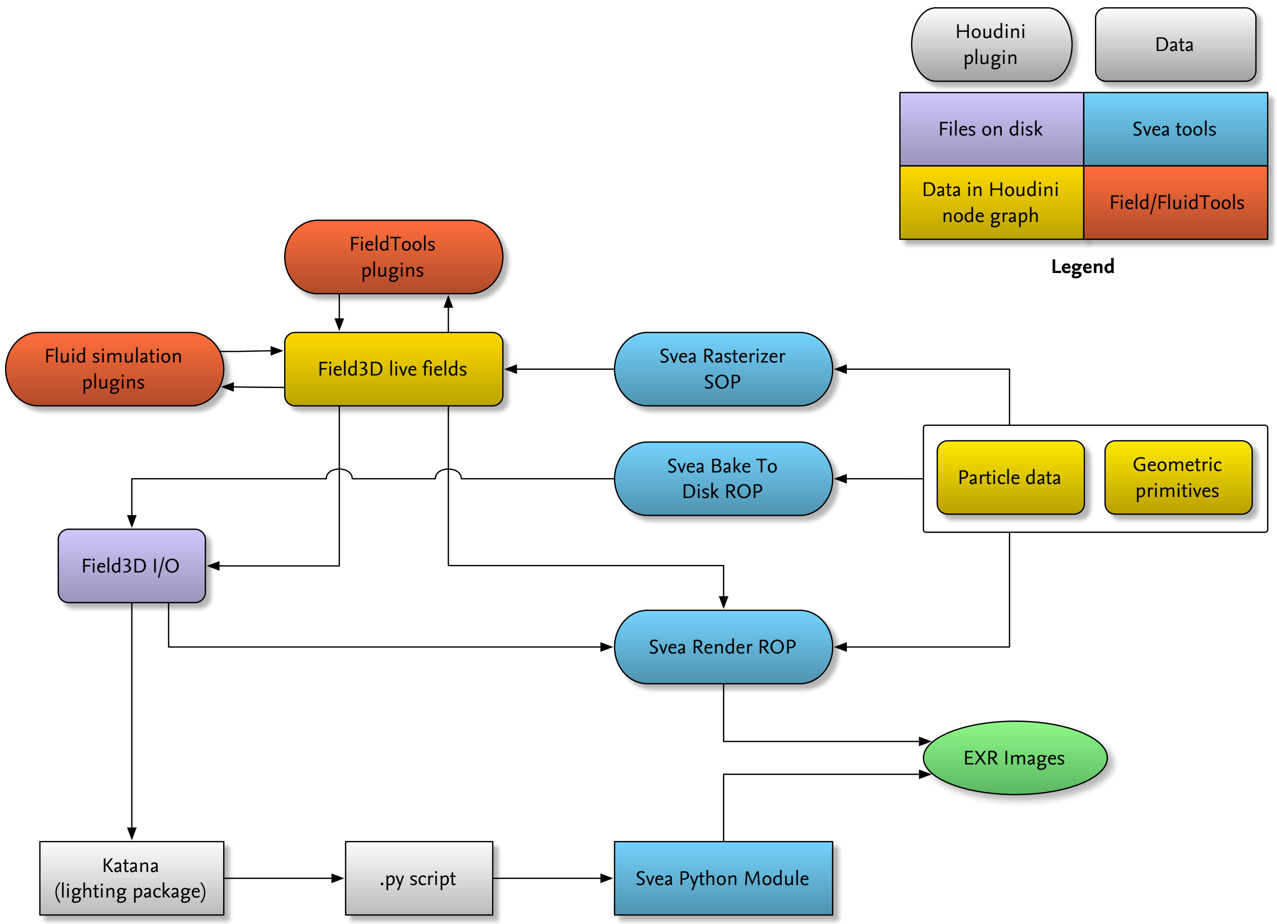
2.2.2. Svea

The volume modeling and rendering tools at Imageworks go by the name Svea (Sony Volume Engineering Application). Some of its key features are:

- **Extendable and scalable modeling pipeline**. An API gives TDs and developers the ability to extend most stages of the modeling pipeline using plugins. Rasterization and instantiation primitives can both be extended, and a secondary stage for processing of instantiated geometry (Filters) makes for a modular workflow. To ensure that large data sets can be generated, the pipeline can operate in a batch/chunk-mode where data is processed piecemeal, in order to keep memory use under control.
- **Flexible rendering pipeline**. Svea is a raymarch-based renderer, and the scene graph is implemented as a shade tree, which lets users build complex trees of volumes/shaders inside the Houdini node graph. The Svea also offers a plugin API for extending the types of volumes supported by the renderer.
- **Resolution independence**. The interaction between the raymarcher and the shade tree uses physical units which makes the renderer agnostic of aspects such as the underlying resolution of voxel buffers, or even whether the data is supplied by voxels, some procedural function, or a combination of both.
- **Effects and lighting artists both have access to the full feature set**. Both the modeling and rendering tools are exposed through the Houdini plugins and Katana (via the Python API). This allows effects artists to not only hand off voxel buffers of data to be rendered by lighting, but to create entire setups that utilize the full modeling and rendering pipeline, which can then be used and tweaked by lighting artists. In the *Production examples* section we will discuss how this was used on Alice In Wonderland to let lighting artists handle volumetric effects on over one hundred shots.
- **Integration with FieldTools pipeline**. Any *live field* present in the Houdini node graph can be rendered directly in Svea, and voxel buffers created by Svea can be passed back into the Houdini node graph and processed by other tools without having to write any data to disk. Likewise, the shade tree that is built and evaluated during the raymarch step can be rasterized and returned to Houdini as voxel data.

Although Svea and FieldTools support both voxel buffers and procedural volumes, Svea leans towards a procedural approach where voxel buffers are but one instance of generic volumes, and FieldTools leans towards discrete (i.e. voxelized) volumes where procedural volumes act as immutable, infinitely high-resolution voxel buffers.

2.2.3. Integration of components



Data flow between various parts of volumetric pipeline

The diagram above illustrates how the volumetric tools are exposed to the user and how data flows between the various parts. We can see that there are several categories of tools that need to interact:

- The FieldTools and FluidTools (in red) all work independently and communicate by passing Field3D fields directly through the Houdini node graph. We refer to these fields as *live fields*, and to distinguish them from Houdini's native primitives they are also *blind data* when see by Houdini's built-in nodes. These fields may flow into the volume rendering part of Svea (Svea Render ROP) regardless of their origin, making it possible to directly render previews of running fluid simulations and field processing operations, without having to write a file to disk.
- The Svea plugins in Houdini (blue), which are responsible for volumetric modeling operations as well as for rendering volumes into final images. Modeling operations take geometry data from the node graph (particles, curves, and surfaces) and rasterizes the data into Field3D fields. Voxel rasterization can happen in any of the four modules, and is only dependent on receiving a *scene description*, which can be supplied either directly from the Houdini node graph or through the Python API. The

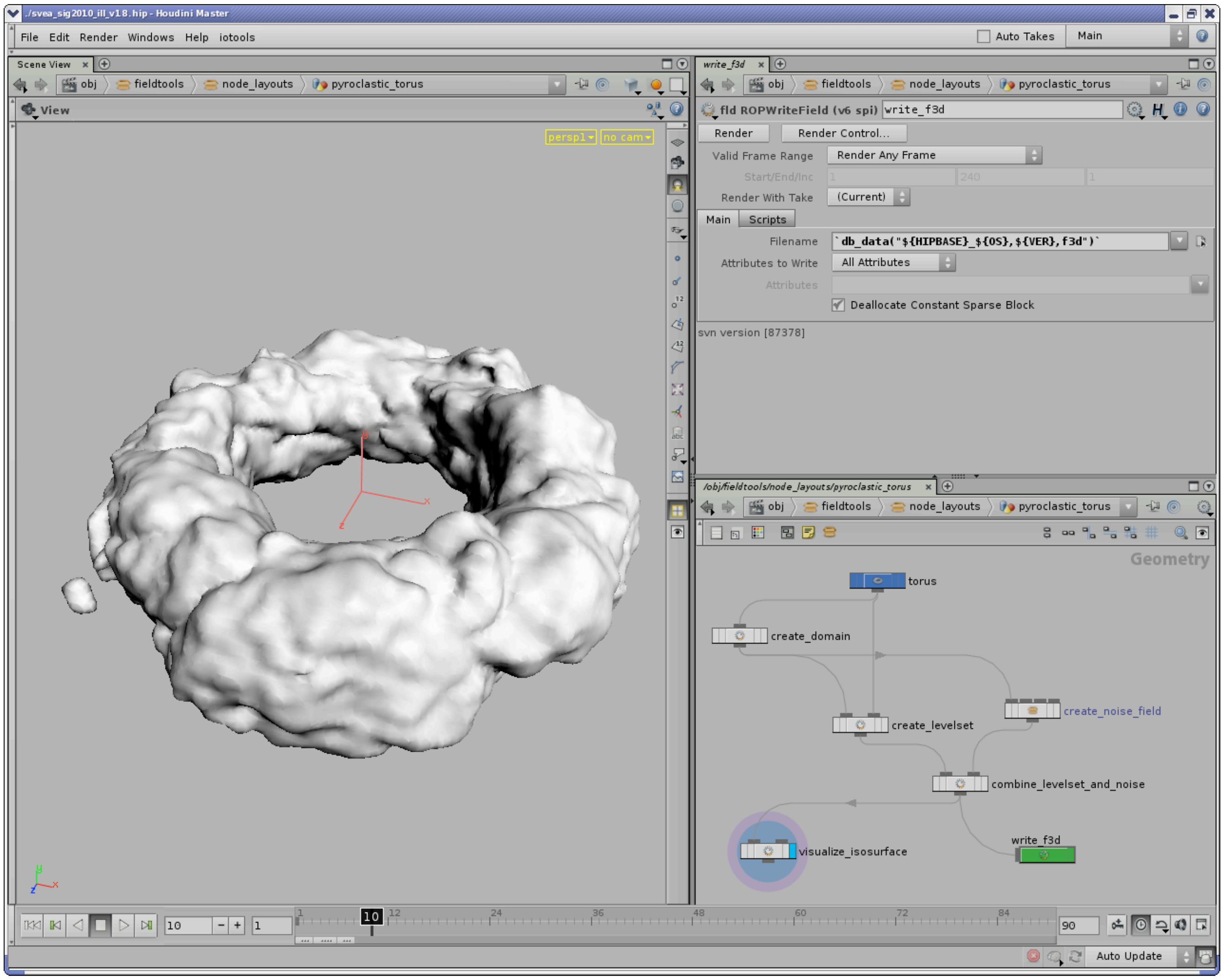
rasterized volumes can either be raymarched directly (in the Svea Render ROP), be output back into the Houdini node graph (the Svea Rasterizer SOP) or written to disk (the Svea Bake To Disk ROP).

- The lighting application (Katana) uses Svea through the Python API, and creates Svea volume primitives in the Katana scene graph, which are then translated into a generic scene description form and written to .PY files. Once translated, the Svea scene description is identical to the data passed from Houdini to the Svea OPs, and exposes the full Svea processing pipeline, including both modeling and rendering operations.

2.2.4. Examples

To illustrate how the various tool sets are integrated we will now see how some simple tasks are accomplished in the volume pipeline.

This first example shows how a piece of geometry (*torus* node) is converted into a level set representation, modulated by a noise field, and converted back to geometry for visualization. Although the tools are resolution independent, sharing the domain and voxel resolution between both the level set and the (voxelized) noise field is more efficient as the plugins detect that the operation may be performed in voxel space, and avoids unnecessary coordinate space transformations.



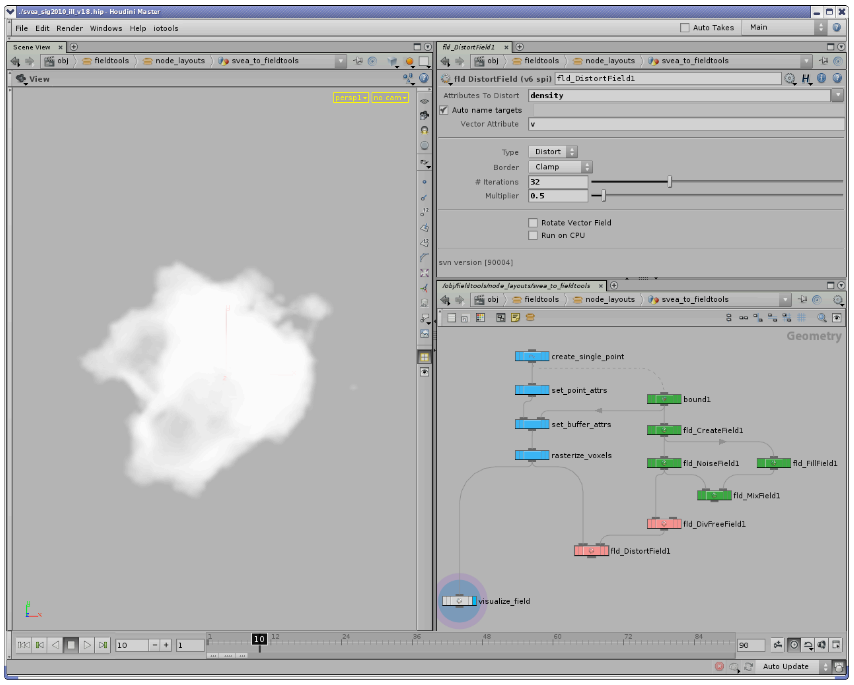
fld ROPWriteField (v6 spi) write_f3d

Valid Frame Range: Render Any Frame
Start/End/Inc: 1 240 1
Render With Take: (Current)
Filename: `db_data("${HIPBASE}_${OS},${VER},f3d")`
Attributes to Write: All Attributes
Attributes:
[✓] Deallocate Constant Sparse Block

svn version [87378]

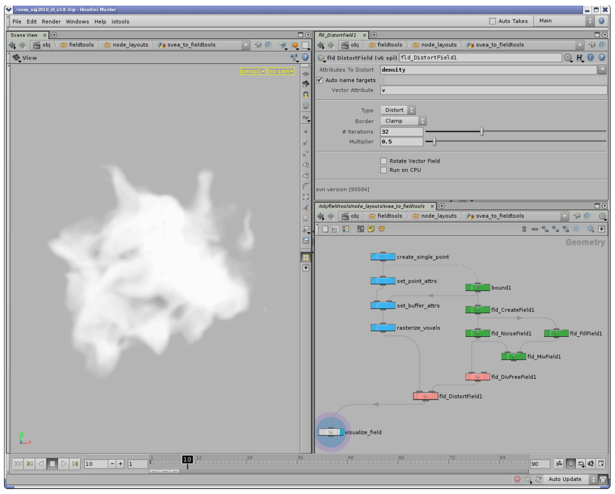
Processing level set data using FieldTools operations

The next examples illustrates how the various tools can work together across functional boundaries to accomplish a task. All data is shared between the various plugins, and the voxel buffer created inside the voxel rasterization node is available directly to the distortion node. In this case, the resulting density buffer is visualized using OpenGL, but it could also be rendered by Svea and be accessible to any part of its shade tree. The field distortion code is GPU-aware and will execute on CUDA-compatible graphics cards, or revert to an identical CPU implementation if no graphics card is available.

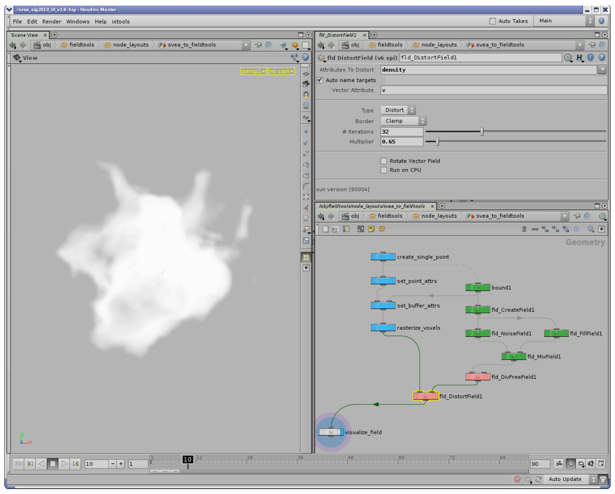


fld_DistortField1
Attributes To Distort: density
Auto name targets: ✓
Vector Attribute: v
Type: Distort
Border: Clamp
Iterations: 32
Multiplier: 0.5
Rotate Vector Field: ☐
Run on CPU: ☐
svn version [90004]

Svea nodes (blue), FieldTools nodes (green) and FluidTools (red) used in conjunction



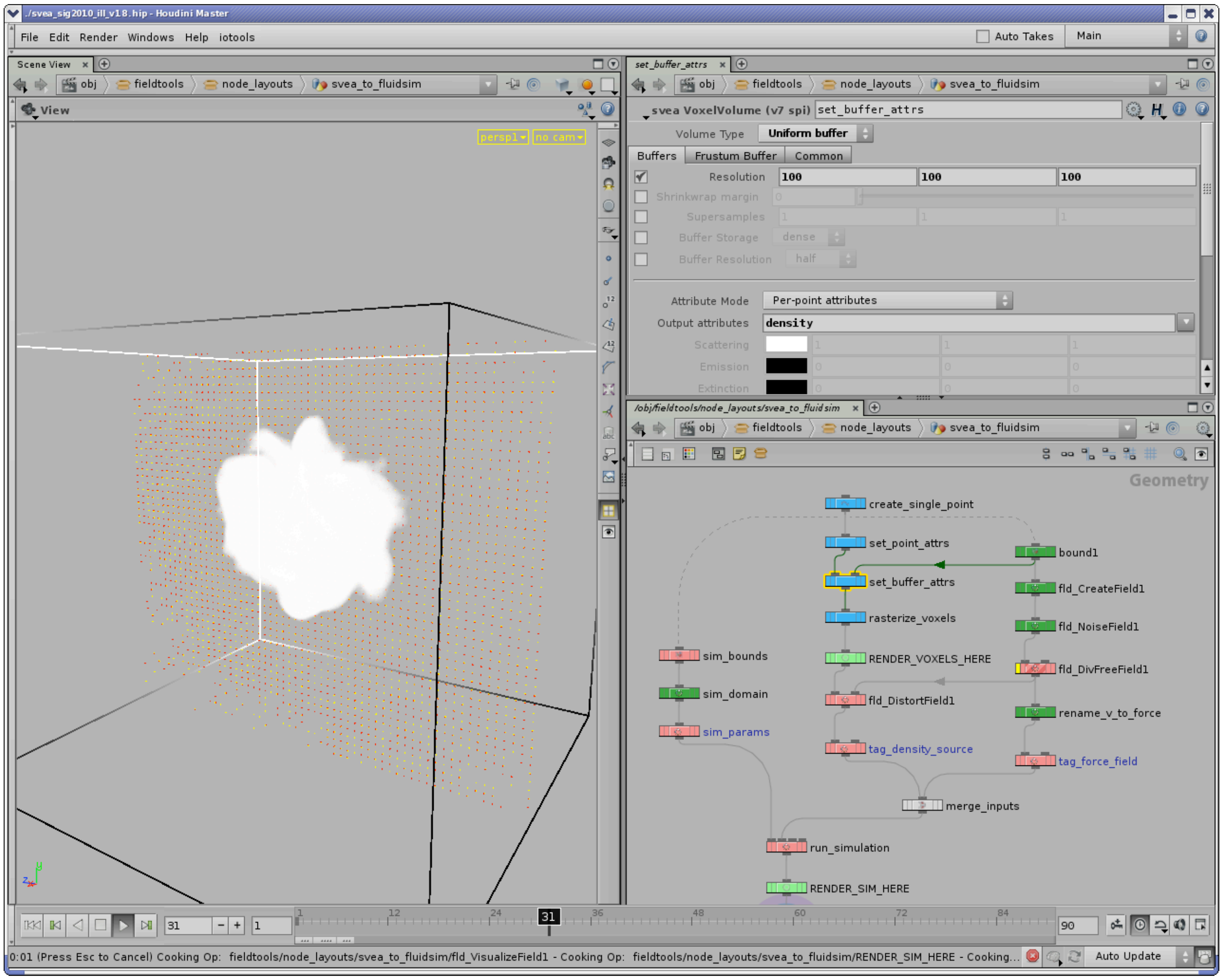
Svea point primitive distorted by noise vector field



Attributes To Distort: density
Vector Attribute: v
Type: Distort
Border: Clamp
Iterations: 32
Multiplier: 0.65

Distortion only affects top of noise point after modulating vector field by a gradient ramp

In this third example, the same noise point is animated and used as a density source in a fluid simulation. The flexibility given by this level of integration and direct interaction between components is important in production work, where each added step in a process is a potential source of errors and user mistakes.



Volume Type: Uniform buffer

Resolution: 100 100 100

Attribute Mode: Per-point attributes

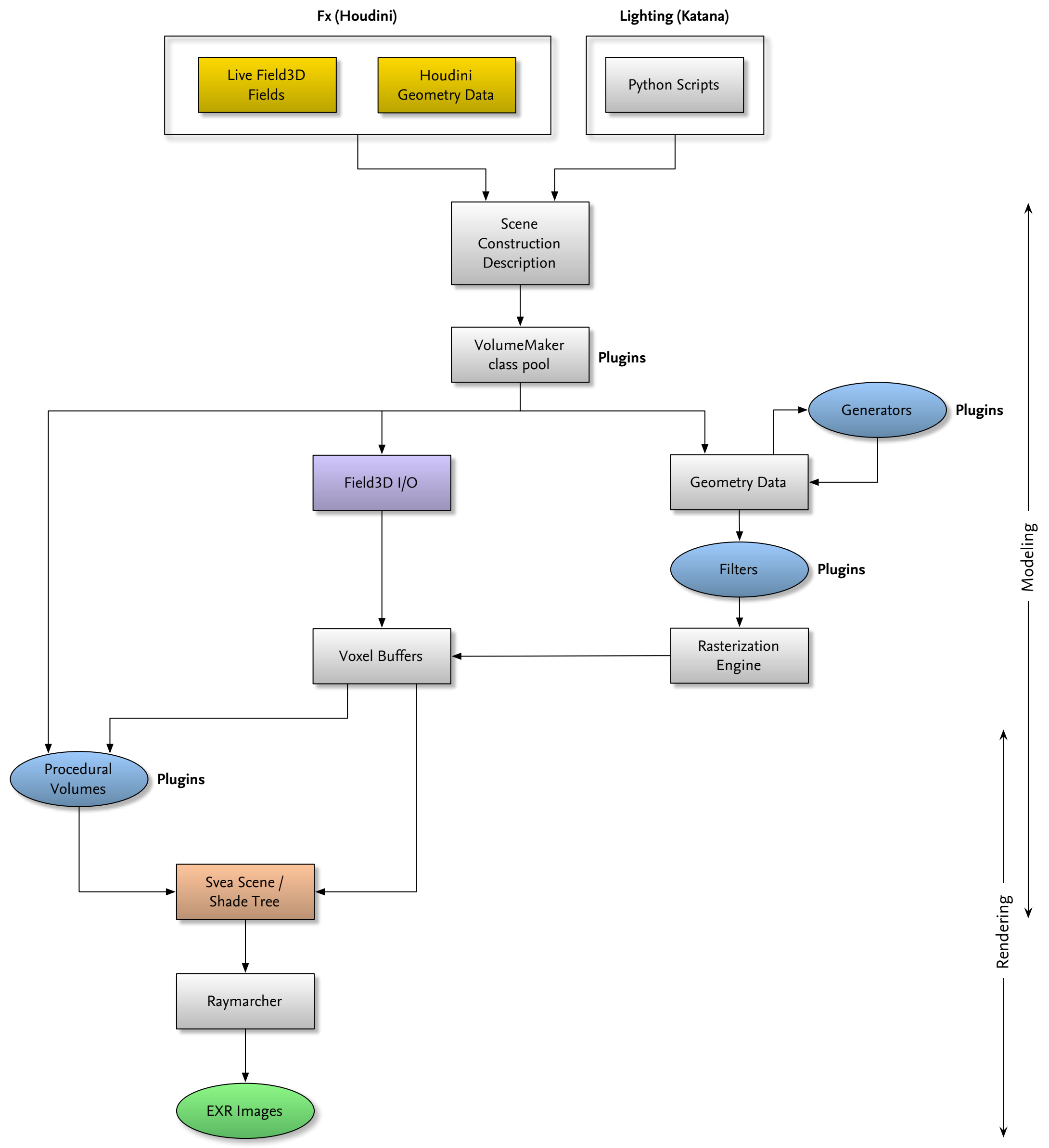
Output attributes: density

0:01 (Press Esc to Cancel) Cooking Op: fieldtools/node_layouts/svea_to_fluidsim/fld_VisualizeField1 - Cooking Op: fieldtools/node_layouts/svea_to_fluidsim/RENDER_SIM_HERE - Cooking...

Using a Svea primitive as a density source in a fluid simulation, and a voxelized vector field as a force input

A key design objective was to maximize the utility of both the in-house and the existing tools in Houdini. With this in mind, it is possible to convert Field3D fields to and from Houdini's native volume primitives, and also to a particle system representation which allows the artist to manipulate volume data in Houdini's native expression language as part of their workflow. The proprietary volume primitives also interact efficiently with Houdini's built-in transformation nodes, so that arbitrarily high resolution data sets can be moved around a scene in real-time.

3. Volume modeling



Volume modeling and rendering pipeline

The graph above shows how the Svea modeling pipeline processes and combines data in a number of ways in order to create a `Scene` (which is effectively a shade tree). Voxel rasterization is often the method used, but the pipeline is not limited to just voxels – instead it maintains a plugin system of `VolumeMakers` whose task it is to turn each item of the scene description into one or more `Volumes` that is part of the final `Scene` (the *Shade trees* section will describe this in more detail).

The *Scene Construction Description* contains a set of `GeometryData` objects, each of which contains a basic set of primitives and points along with global, primitive and point attributes. This serves as the basic description of how each `Volume` should be created.

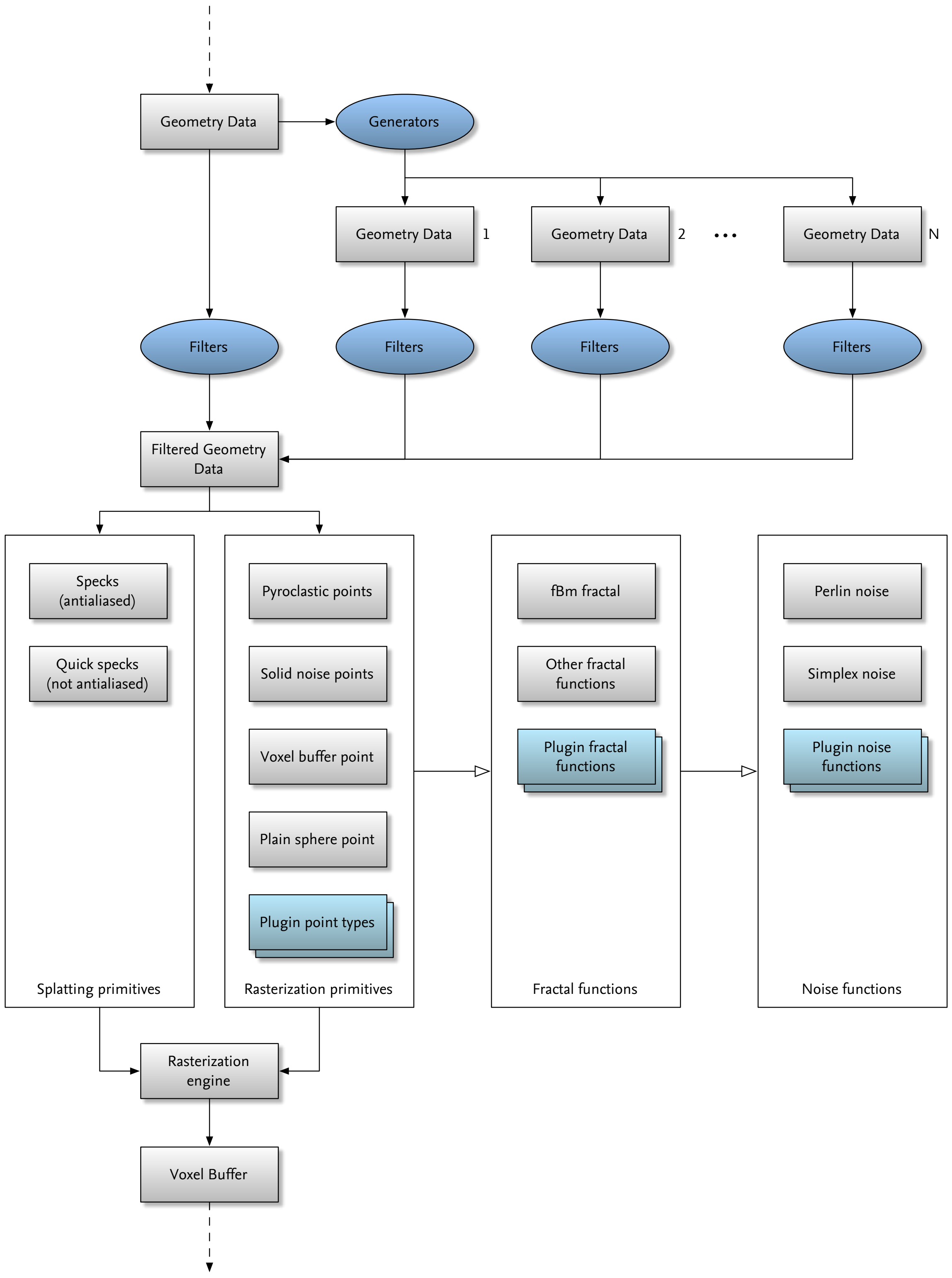
- *N* `GeometryData` objects
  - Detail (global) attributes
  - Primitives, primitive attributes and primitive groups
  - Points, point attributes and point groups

Once the `VolumeMaker` class pool is handed the scene construction description, it will dispatch each `GeometryData` object to its appropriate `VolumeMaker` (based on a global attribute), which extracts information about how to set up its particular type of `Volume`. Some examples of `VolumeMakers` are:

- **File I/O**, which reads Field3D and other supported volume formats from disk.
- **Rasterization pipeline**, which creates voxel buffers. (Described in more detail in the next section.)
- Each type of procedural volume also uses a `VolumeMaker` to configure its look, based either purely on global attributes, or on more complex input types, like geometry.

While the rasterization pipeline falls squarely in the volume modeling side of things, some procedural volumes blur the line by doing their modeling work at render-time, as part of the shade tree evaluation itself. Procedural noise banks, compositing- and attribute manipulation-operators are some examples. This is in contrast to parts of the shade tree that deals only with rendering tasks, such as marking volumes as holdout objects, nodes for render-time deformation blur, etc. The shade tree will also be described in more detail further on in these notes.

The following diagram shows the rasterization pipeline in more detail, and the next sections will describe two of its key features: Generators and Filters.

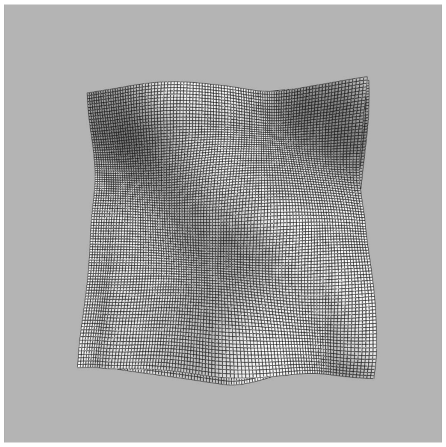


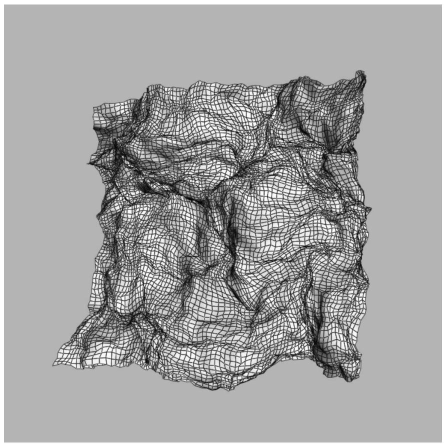
3.1. Generators

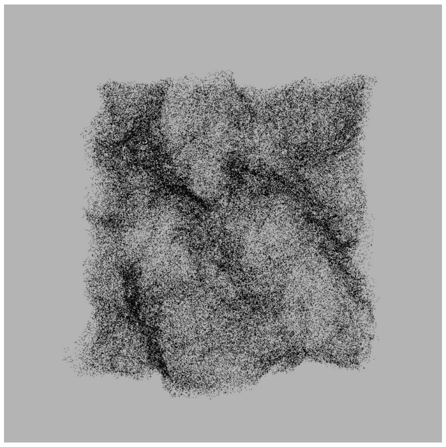
Svea's mechanism for instantiation-based primitives is called `Generators`. During the volume modeling process, each `GeometryData` instance can spawn an arbitrary number of `Generator` objects, each getting access to the `GeometryData` of its creator. This is done so that a single `Generator` object may handle instantiation for any number of primitives in its input.

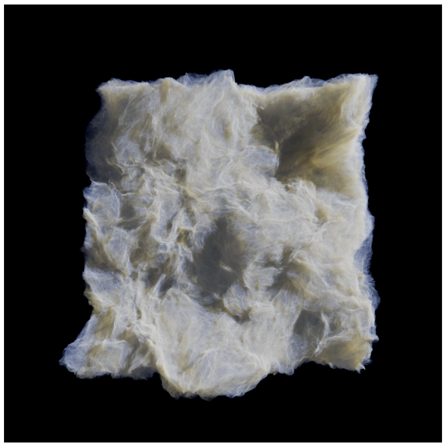
Each generator may in turn create *N* new `GeometryData` objects, which are either further recursed, or passed to the filtering stage and then rasterized. Because the amount of data generated by a single `Generator` can be arbitrarily large (production scenes sometimes involve billions of point instances), this batch-processing mode is needed in order to guarantee that rasterization will finish given a finite amount of available memory. Some examples of generators available in Svea are:

- **Speck Spline –** curve-based point instantiation.
- **Speck Surface –** surface-based point instantiation.
- **Cluster –** a space-filling algorithm that instantiates points based on the configuration of an input particle system. Used for everything from white water to mist and smoke-like effects.


Underlying primitive


Preview of distortion noise


Preview of point instantiation


Rendered result

`Generators` are only instantiated inside the volume modeling pipeline, and because of this they can be less than intuitive to use. For this reason we provide several ways of previewing the result of the instantiation. Each generator's parameters are controlled using a specialized SOP in Houdini, which is able to preview the effects of all of its parameters directly as Houdini guide geometry. We also provide a specialized SOP which replicates the generator pass in the modeling pipeline, so that instantiation can be executed directly within the Houdini node graph, passing the instantiated points as its output. These have in combination proven to be a good way to accelerate the artists' efforts when doing effects look development and production work.

In the illustrations above, readers may notice a slight discrepancy between the geometric preview of the distortion noise and the final result. Because the geometric preview only samples a given depth offset (measured along the normal from the root primitive), it only represents one *slice* of points in the final instantiated output. This slice plane may be placed at any depth by the user, allowing preview of the full volume.

3.2. Filters

`Filters` act as the final set of shaders before a primitive gets rasterized. They are instantiated and executed based on attributes set either by the user in the Houdini node graph, or by attributes created in a `Generator`. `Filters` are a type of shader that have access to the full state of the `GeometryData` instance before it gets rasterized.

Some examples of `Filters`:

- **Texture projection**. Allows images to modulate the opacity, color or any other attribute of a rasterization primitive.
- **Attribute randomizer**. Used to create per-primitive variation post-instantiation. Especially useful for randomizing velocity vectors of instantiated points.
- **Camera-based projection**. Used to project the film plate itself into a volumetric element.

The effect of filters can be previewed using the same tool that is used to preview point instantiation (as described above).

3.3. Procedural volumes

We generally distinguish between ordinary `Volumes`, which create volumetric data, and `AdapterVolumes`, which modify attributes in the shade tree (see below). `Volumes` can be both leaf and branch nodes in the shade tree (depending on whether they are driven by the properties of a secondary volume), whereas `AdapterVolumes` always reside at branch points (they required an input).

Some examples of regular `Volumes` are:

- **Voxel buffers**. The primary source of volumetric data.
- **Procedural noise banks**. Used to add detail at render-time.

Some examples of `AdapterVolumes`:

- **Compositing volumes**. Used to combine the values of one or more volume nodes.
- **Texture projection volumes**. Allows render-time projection of images into the scene. (The texture filter mentioned in the previous section is applied at rasterization time.)
- **Density fit functions**, for manipulating already-rasterized data at render-time.
- **Color ramp shaders**, which map scalar attributes to user-defined color gradients.
- **Blackbody shader**, for rendering fire and explosion-type effects.
- **Vector blur & distortion**. Used for deformation blur of fluid simulations, and for artistic effects.
- **Arbitrary shaders**, written by TDs to manipulate other attributes in the shade tree

The `Volume` classes provided with Svea serve most common volume rendering tasks, and usually the plugins that get written are `AdapterVolumes`. However, both `Volumes` and `AdapterVolumes` are written as C++ plugins and can be extended by TDs and developers as needed.

An examples of how `Volumes` are implemented is provided in the *Shade trees* section below.

3.4. Cluster

One particular point instancing algorithm called *Cluster* has survived repeated incarnations at Imageworks. Cluster was originally developed for *Surf's Up* by E.J. Lee and was implemented as a RenderMan DSO, inspired by other point instancing schemes that were in use at the time, such as ILM's work on Perfect Storm. For *Surf's Up*, Cluster was used to amplify the point count for all the whitewater rendering in the movie.

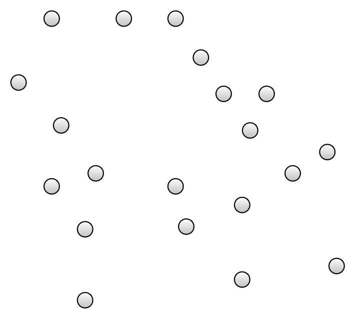
At its essence, Cluster is a neighbor-fill type of instancing scheme, which takes an unstructured point cloud and instances more points, taking into account the position of each point as well as its neighbors. After *Surf's Up*, Cluster was brought into Svea as a `Generator` plug-in, retaining the original version's approach and algorithm. On *Hancock* it was used for many of the dust and debris effects.

A third version was implemented for *Alice In Wonderland*, again as a plug-in to Svea. The first version had used an explicit approach to deciding how many points needed to be instanced around each source point. This left a lot of setup work to the user, who had to implement LOD-schemes manually in order to avoid instancing unnecessary amounts of points far away from camera. The third version altered its paradigm slightly, focusing instead on the connections and the amount of perceived density between each point pair, leaving the decision of how many points to use up to the renderer. This allowed Cluster to be used both for particulate effects, as well as smoothly filled volumes.

The algorithm can be summarized in a few simple steps:

- Loop over each point in input point cloud
    - Given a radius *R*, find all neighboring points within *R*
    - Connect each neighbor to the current point with a line
    - Instance new points on or around the line

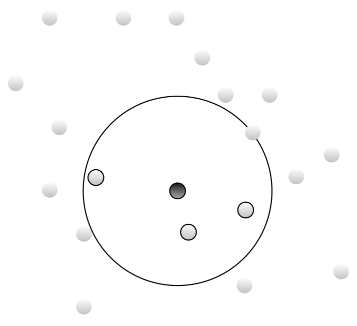
To illustrate the algorithm, we will use a simple 2D point cloud.



Example point cloud

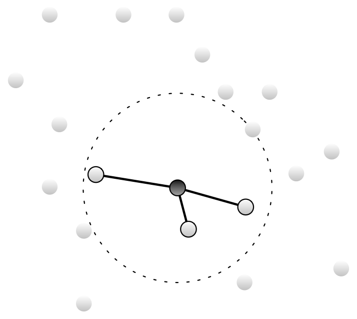
The goal of the instancing algorithm is to add extra points in a way that highlights the structure of the input point cloud. A naïve instancing scheme might simply scatter instanced points around each input point, but this approach generally results in a blurring of the underlying structure. Cluster uses a different approach, where the positions of neighboring points decide where instance points are placed.

The first step in the algorithm is, for each input point, to find all neighboring points within a given radius. Finding closest neighbors for each point in a brute-force fashion is $O(N^2)$, so to accelerate this query the input point cloud is first sorted in a KD-tree that allows fast nearest-neighbor lookups.



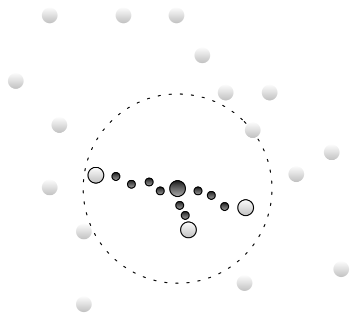
Neighbor search radius

The next step is to create a user-specified number of *N* connections from the current point to its closest neighbors. That is, if the user requests a maximum of 3 connections from each point and a total of 10 neighbors were found within a radius *R*, the closest 3 neighbor points will be used. This user parameter may vary per-point, so that each input point connects to a different number of neighbors.



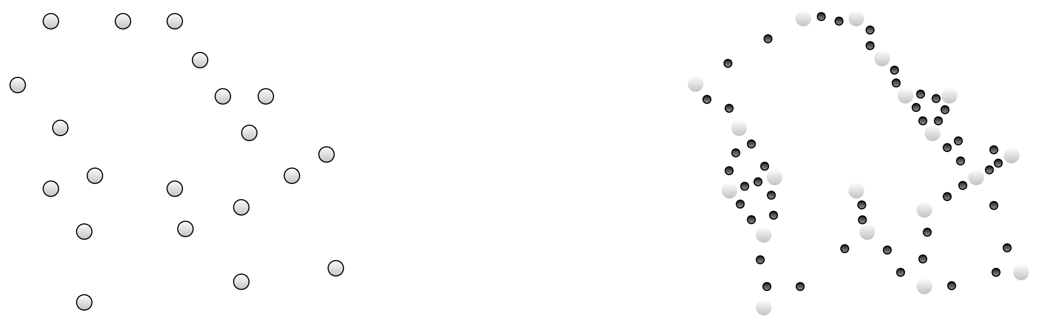
Forming connections with neighbor points

For each connection, a given number of points are instanced. The number of instanced points and their placement along the line is controlled by several parameters which let the user bias the instances towards the center or either end of the connection, and also to offset the points outwards from the line to fill in a tube. As with the number of connections, the user has control over the number of instances.



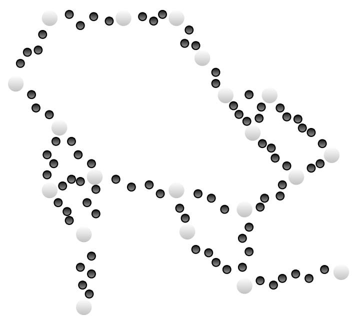
Instancing points along connection lines

The final result, with two points instanced between each input point:



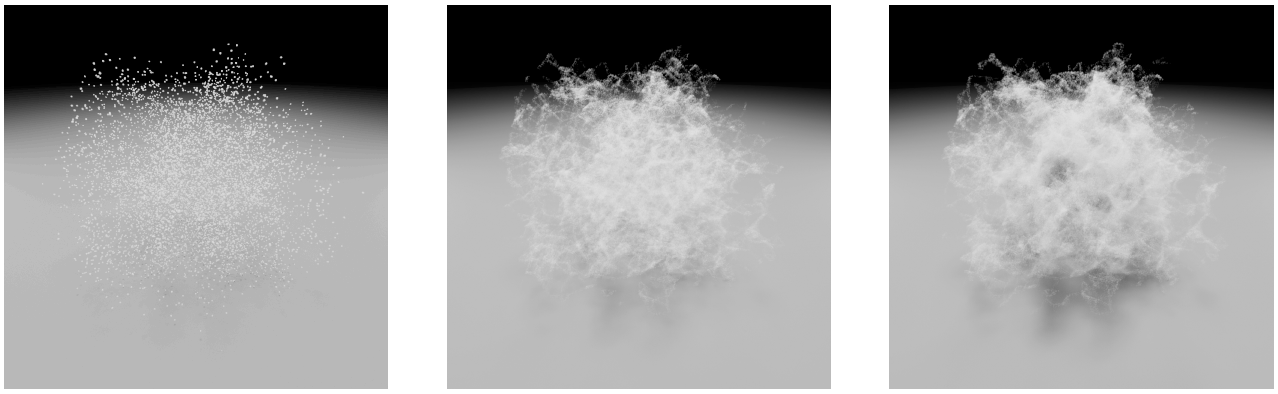
Performing instantiation for all points in point cloud

As mentioned previously, Cluster is sometimes used to instance enough points that the final result no longer appears to be made up of individual points, but rather a smooth volume. In these cases, specifying a particular number of points for each connection is no longer the best user control. Our latest incarnation of Cluster, written for *Alice In Wonderland*, instead leaves the instance count up to the renderer, and gives the user control over the radius and overall density of each connection. The result is an adaptive point count that strives to maintain a smooth appearance while using as few instance points as possible. This approach has a built-in LOD effect – points far away from camera do not need as many instances as when they are close.

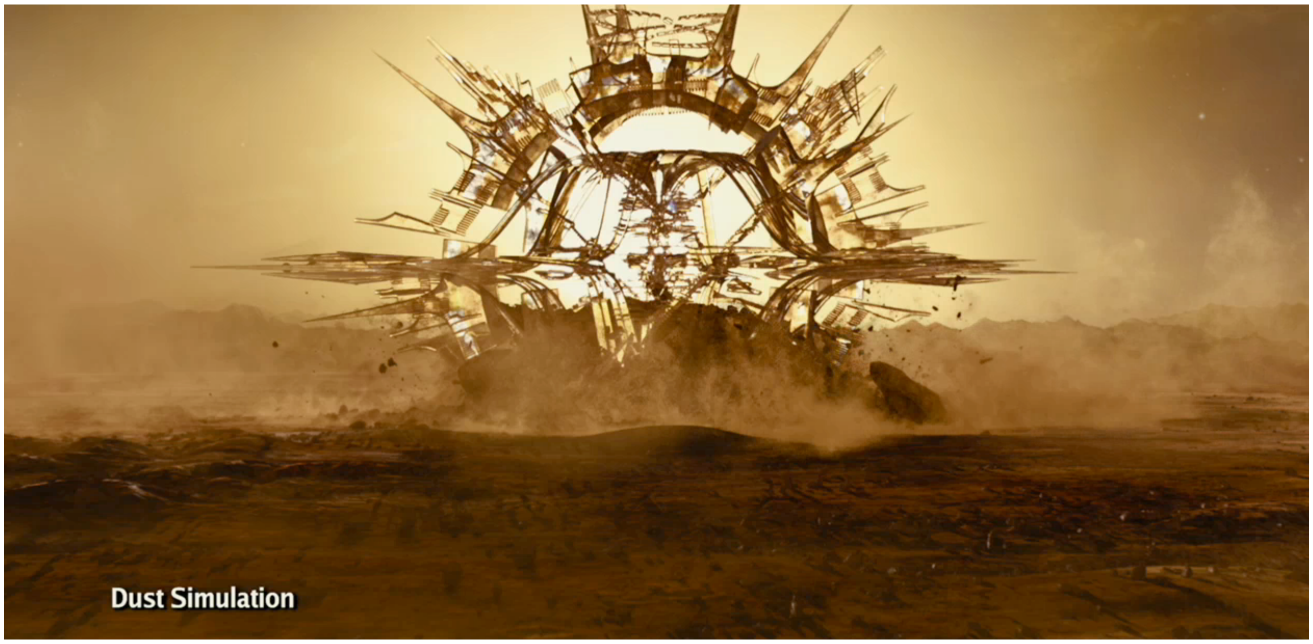


Adaptive instance point count

Although simple, Cluster can be used to create a variety of looks, which can highlight different qualities in a point cloud.



Left: Input point cloud
Middle: Limiting max # connections to 4
Right: Connecting to all possible neighbors



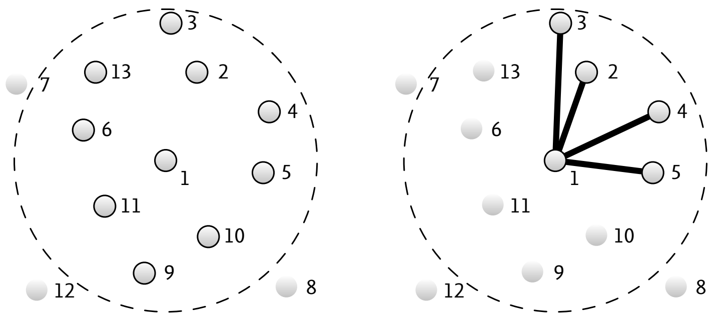
In Watchmen, the dust clouds on Mars were created by Cluster.
© 2009 Warner Brothers. All rights reserved.

The Cheshire Cat's effects were created using Cluster.
Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

3.4.1. Animating point clouds

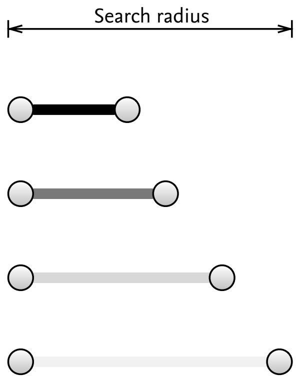
Some care needs to be taken when the input point cloud is animating. The nearest-neighbor query can change from frame-to-frame as the set of closest *N* points differ, and a connection can disappear because a neighbor point moves outside the search radius. Whenever these events occur, a connection will form out of nowhere or disappear from one frame to the next, causing a visible *pop* in the animation.

These two problems can be overcome by slightly altering the instancing algorithm. In order to prevent the changing of closest points order, we can add a sort step before connecting to the closest points. The sorting can be based off of any attribute existing on the point data, for example unique id. In this case, the exact closest neighbor is no longer picked, rather the first *N* neighbors with lowest *id* will have connections made.



Using point id for connection order

We also need to prevent connections from abruptly turning on or off for points that move into or out of another point's search radius. This is easiest to do with a ramp function: simply reduce the density of the instanced points if their connection's distance approaches the search radius. Points within a radius R_{inside} have full density, and then decrease as R approach R_{search}.



Fading the density of a connection as $\mathbf{R}$ *approaches* $\mathbf{R_{search}}$

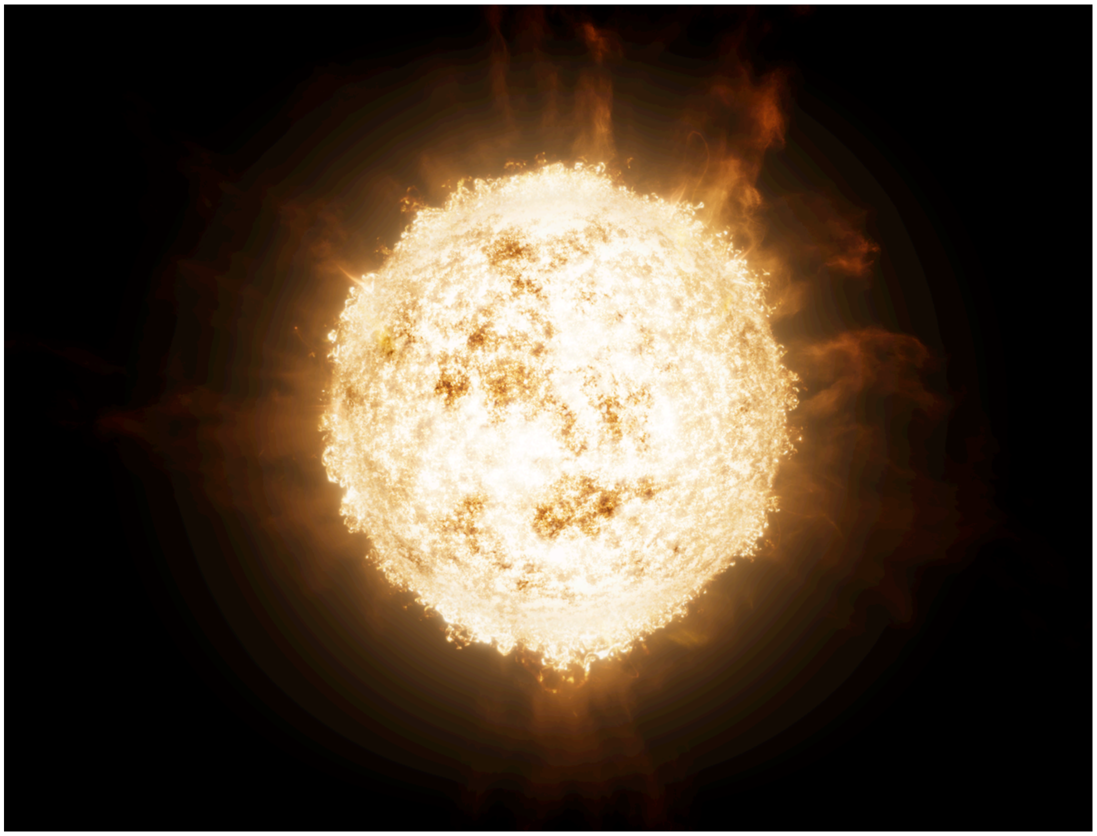
3.5. Voxel buffers as modeling primitives

At the course last year, Double Negative presented a technique used for the film *2012*, where entire fluid simulations were instanced on particle data. Inspired by their presentation, we implemented a new rasterization primitive that accomplished the same goal: stamping existing voxel buffers onto each particle in a point cloud.

The Svea implementation is very similar to what Double Negative described, and can be broken down into a few simple steps:

1. Load the Field3D file from disk
2. Start the rasterization loop for the current particle
3. For each voxel
    1. Transform from world space into the local space of the particle
    2. Interpolate the fluid data at the same local space position

For Svea's implementation we used Field3D's built-in caching of `SparseField` buffers. The cache ensured that particles that shared the same fluid data only loaded the data once, and if points were sorted by filename, we could ensure that data for subsequent particles was likely to still be available in the cache. The cache also unloads data as needed, which kept memory use under control.


The sun in Green Lantern was created using 30.000 instantiated fluid simulations of various scales. © 2011 Warner Brothers. All rights reserved.

The `StampBuffer` (as it was named) primitive was used for several effects: The sun (described in more detail in the Production Examples chapter), and for most of the clouds and atmospheric effects on planet Oa.

The atmospheric elements of Planet Oa were built using instantiated fluid simulations.
© 2011 Warner Brothers. All rights reserved.

4. Volume rendering

4.1. Svea's rendering pipeline

A fundamental part of Svea's rendering pipeline is the `Scene` object, which contains a reference to the root of the *shade tree*, as well as information about the camera and the scene's lights. The shade tree can range in complexity from a single node (in the case of rendering a single voxel buffer), to complex configurations of tens or hundreds of nodes, several levels deep. The nodes in the shade tree are all `Volume` instances (which is what was created in the volume modeling process described previously).

`Volumes` have a very simple interface:

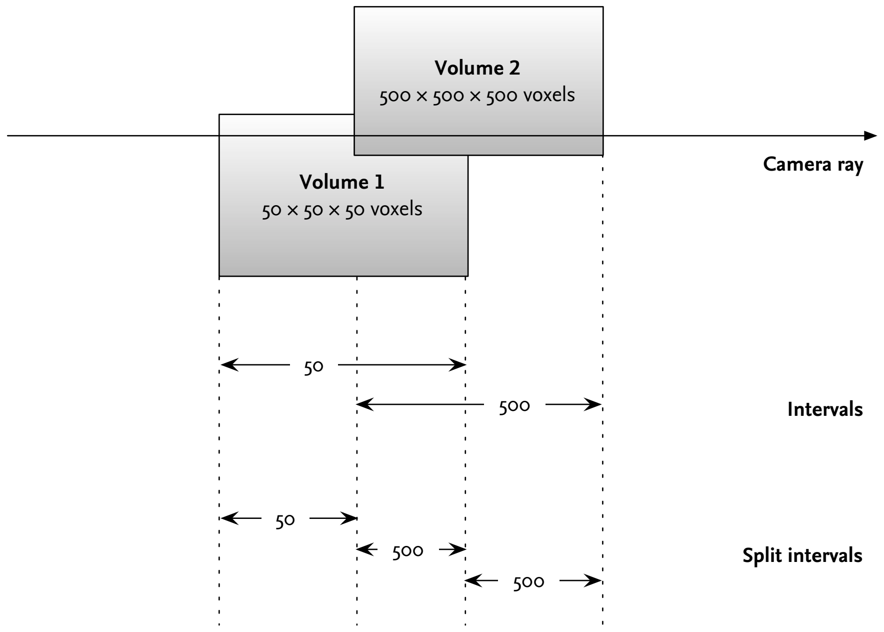
```
class Volume:
{
public:
   virtual Color value(const Attribute &attribute, const SampleState &state);
   virtual void getIntersections(const Ray &ray, std::vector<RaymarchInterval> &outIntersections);
};

struct RaymarchInterval
{
   float t0, t1;        // Start and end of interval
   float sampleDensity; // Required number of samples per length unit
};

struct SampleState
{
   V3f wsP;             // World-space sample location
   V3f dPds, dPdt, dPdu // Pixel derivatives in x/y, and raymarch step size
};
```

We note that all volumes return a spectral color from the `value` call – even scalar voxel buffers convert their results into a `Color`, and the raymarcher only deals with spectral properties. Also, in order to better support sparse volumes, each `Volume` may return an arbitrary number of intervals that should be raymarched, along with the required sample density for each. The sample density relates to the volume's frequency content and is an indication of its Nyquist limit. One alternative would be to find the most restrictive sampling density in the scene and use that throughout, but using a varying step length helps speed up rendering of areas that do not require fine sampling.

The following is a slightly simplified list of steps needed in order to render an image:

- **For each pixel** in the output image
    - **Intersect ray against scene**. This yields *N* raymarch intervals
    - **Split raymarch intervals** into non-overlapping segments
    - **For each split interval**, starting nearest camera
        - **Run raymarch loop**, updating transmittance and luminance

When multiple volumes are visible to the ray we need to make sure that they are each sampled appropriately. Different volumes may have different frequency content, and may required different sampling densities. The illustration below illustrates the problem where two volumes overlap.



During raymarching, all volumes are sampled together (the raymarcher only sees the root node of the shade tree). Because **Volume 1** requires much fewer samples than **Volume 2** we see that we only need to sample at 500 samples/length-unit for the portion of the ray that overlaps the second volume; in the first interval 50 samples/length-unit will be sufficient. We therefor split the intervals created from the intersection points in the scene into non-overlapping *segments*, and choose the most conservative sample rate in each segment.

Once we have a set of non-overlapping `RaymarchIntervals`, we can proceed to raymarch each of them, starting nearest the camera. The raymarching algorithm itself is fairly simple, with the addition of an adaptive sampling scheme (described below) and routines for deep shadow holdouts, volumetric holdouts and light shaders.

4.2. Shade trees

In the illustration of splitting raymarch intervals above, there were two volumes, but it was also mentioned that Svea only sees one volume – the root of the shade tree[2]. In reality, the structure for the example would have been:

root (`GroupVolume`)

- **Volume 1** (`VoxelVolume`)
- **Volume 2** (`VoxelVolume`)

The `GroupVolume` is a simple `Volume` class that groups *N* volumes together and composites their output using some trivial operation (i.e. sum, max, mult). Because it is a `Volume` subclass, it also needs to respond to `getIntervals()`, which it does by calling the same member function on each of its children. While this is a short example, it shows how the shade tree functions in its simplest form.

```
class GroupVolume : public Volume
{
public:
   virtual Color value(const Attribute &attribute, const SampleState &state)
   {
      Color value(0.0f);
      for (VolumeVec::iterator i = m_children.begin(); i != m_children.end(); ++i) {
         value += i->value(attribute, state);
      }
      return value;
   }
   virtual void getIntersections(const Ray &ray, std::vector<RaymarchInterval> &outIntersections)
   {
      for (VolumeVec::iterator i = m_children.begin(); i != m_children.end(); ++i) {
         i->getIntersections(ray, outIntersections);
      }
   }
   void addChild(Volume::Ptr child)
   {
      m_children.push_back(child);
   }
};
```

[2] Cook, R. L. 1984. Shade Trees. *ACM SIGGRAPH Computer Graphics Volume 18, Issue 3. Pages: 223 - 231*

We notice that the `value()` call takes an `Attribute` parameter. The shade tree in Svea supports an arbitrary number of attributes flowing through it, and makes no assumptions about what properties any of its `Volume` members choose to expose. Seen in further detail, the shade tree example above contains a few more pieces of information:

- **root** (`GroupVolume`)
  - *Attributes*:
    - emission
    - extinction
    - velocity
  - *Children*:
    - **Volume 1** (`VoxelVolume`)
      - *Attributes*:
        - emission
        - extinction
    - **Volume 2** (`VoxelVolume`)
      - *Attributes*:
        - emission
        - extinction
        - velocity

When raymarching a scene Svea always looks for a basic set of attributes (see below). The shade tree may make use of any number of other attributes in the process of evaluation, although any extraneous attributes exposed by the **root** node will be ignored by the raymarcher. Thus, velocity would be ignored by the raymarcher, but could be used by other `Volumes`, for example to apply motion blur or distortion effects. The set of basic attributes that the raymarcher samples from the root of the shade tree are:

- **scattering**, used to determine in-scattering and out-scattering.
- **extinction**, used for absorption.
- **emission**, used for self-illuminating volumes like fire.

`Volumes` can also act as a form of shader, altering the appearance of other `Volume` instances. For example, fire in Svea is shaded at render-time from simulation buffers containing temperature and density representations. In this case, it exposes a different set of attributes to the raymarcher than its inputs provide.

- **fire** (`FireShader`)
  - *Attributes*:
    - emission
    - extinction
    - velocity
  - *Children*:
    - **simulation** (`VoxelVolume`)
      - *Attributes*:
        - temperature
        - density
        - velocity

4.3. Adaptive raymarching

Sharp transitions in density are a problem for raymarchers. While voxel buffers behave nicely (i.e. vary smoothly) if their contents are rendered directly using interpolation, shaders and procedural volumes based on voxel buffers can make transitions arbitrarily sharp. Fire shaders are one notable example – they often use steep transitions to increase the perceived resolution of the simulation. For example, even if the voxel spacing of a simulation is Δx, the output of the shader may go from zero to full density/ luminance in distances as short as $0.1\Delta x$ or less. In order to prevent noise in the final image the raymarcher must reduce its step length such that these sharp transitions are captured.

One approach that is particularly easy to implement is to use each raymarch step's contribution to the final pixel value as the heuristic for whether finer sampling is needed. If the contribution is greater than some user-defined threshold (say 1/256'th of pure white color), then the current raymarch step is discarded, the step length is halved and a new calculation takes place.

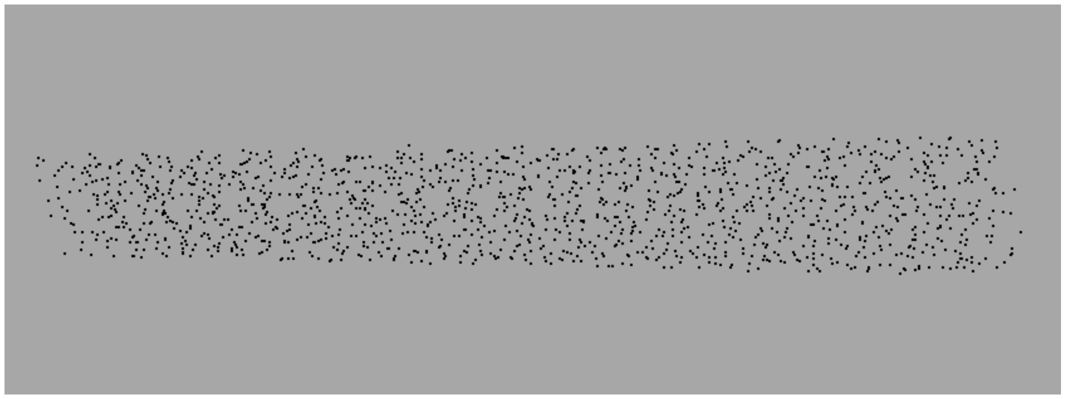
In slightly simplified code (ignoring lighting calculations, holdouts, etc.), the algorithm can be implemented as:

```
float t0, t1;
int numSamples;
scene->intersect(ray, &t0, &t1, &numSamples);
float stepLength = (t1 - t0) / static_cast<float>(numSamples);
// Luminance and transmittance
Color T = 1.0f, L = 0.0f;
// t0 and t1 for each individual raymarch step
float step_t0 = t0;
float step_t1 = t0 + stepLength;
while (step_t1 <= t1) {
   Color Lstep, tau;
```

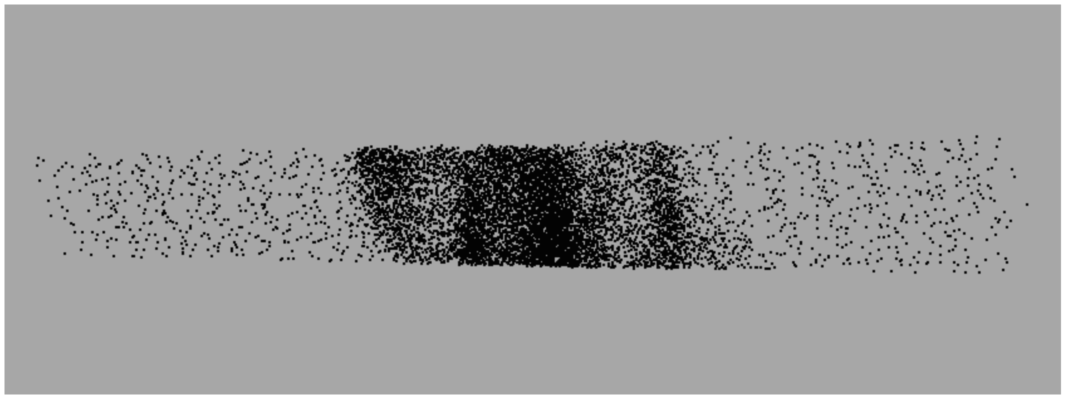
```
        // Sample point
        float t = (step_t0 + step_t1) * 0.5f;
        // Calculate incoming luminance and optical thickness
        lightingCalculation(scene->root, ray(t), &Lstep, &tau);
        // Account for current transmittance and step length
        Lstep *= T * stepLength;
        // Before applying the result, determine if we need to supersample
        if (max(Lstep) > threshold) {
           // Change step interval and recalculate step
           stepLength *= 0.5f;
           step_t1 = step_t0 + stepLength;
           // Terminate current step here
           continue;
        }
        // Apply results
        L += Lstep;
        T *= exp(-tau * stepLength);
        // If contribution is low we can increment step length
        if (max(Lstep) < threshold * 0.25) {
           stepLength *= 2.0f;
           step_t0 = stepT1;
        }
        // Increment step interval
        step_t0 = step_t1;
        step_t1 += stepLength;
     }
```

The algorithm has some notable benefits – it works without any knowledge of how the lighting calculation happens or how density translates to changes in transmittance. And because it is only dependent on the given step's contribution to the final pixel luminance it automatically takes into account both the current transmittance, which in turn incorporates holdouts, etc. This makes the oversampling less strict as the ray marches further into a volume, in effect being most sensitive at the first steps into a volume, which is where it is usually needed.

The code can be further modified to look at the change in luminance relative to current pixel luminance, and also to look at changes in transmittance. These both work well in production but are left as an exercise to the reader.



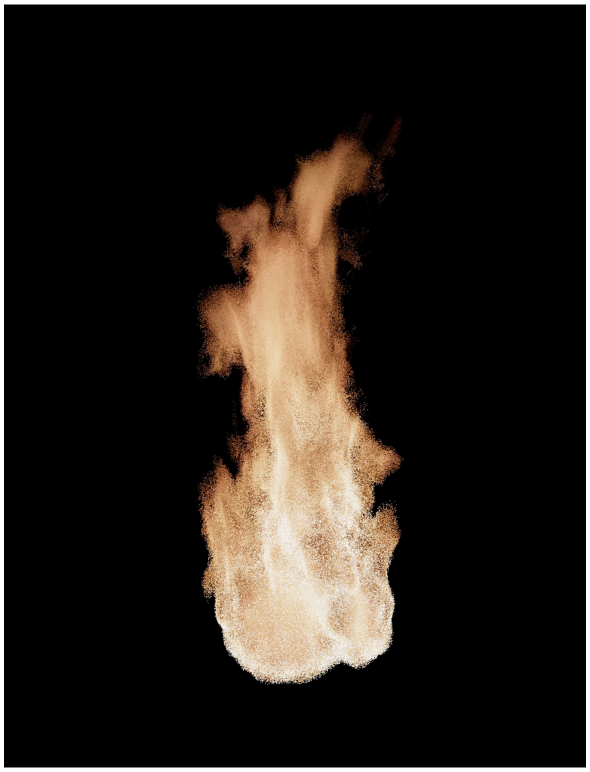
Cross section of raymarch samples through a fire volume



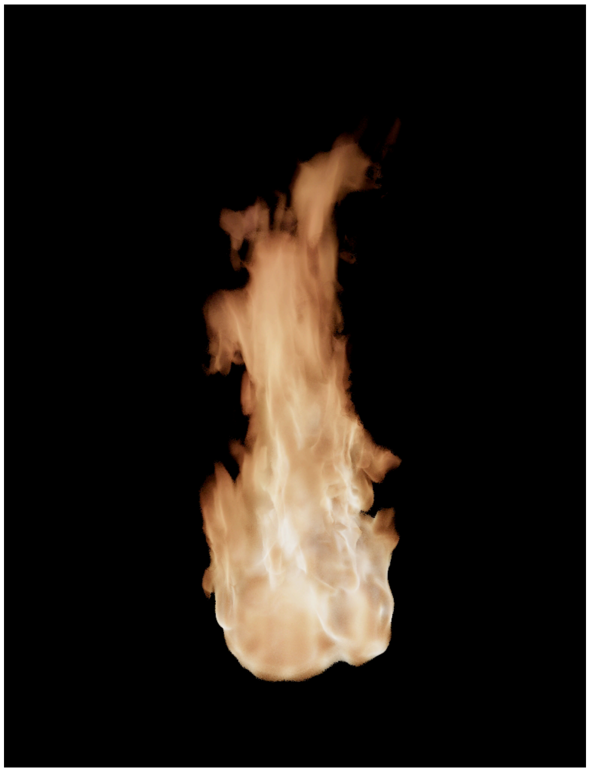
Cross section of raymarch samples with adaptive sampling enabled

In the following images, a simple fire element is raymarched at various step lengths to illustrate the level of oversampling that is necessary in order to completely remove noise from a render. The brute-force solution of globally increasing sample density is several times slower than implementing an adaptive strategy, when comparing achieved noise levels.

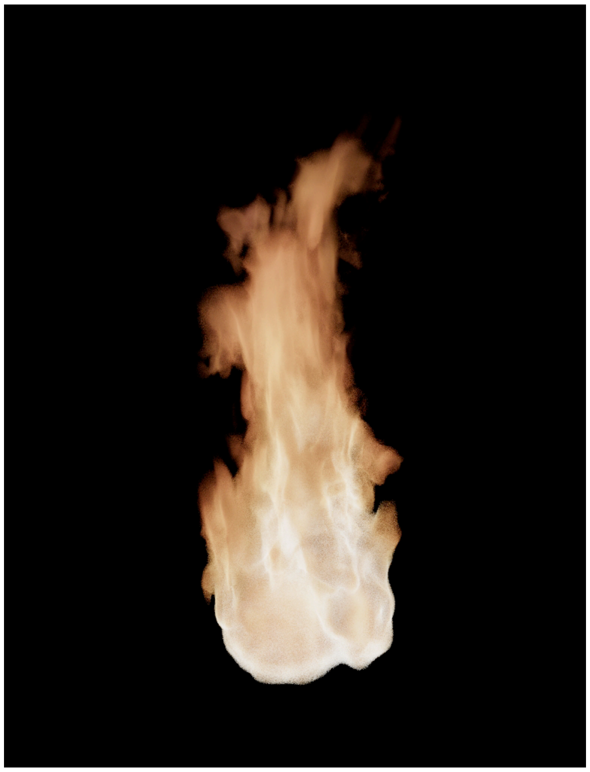
Step length	Time per frame
1 voxel	21s
1/16 voxel	5m
1/64 voxel	20m 30s
1/2 voxel, adaptive refinement	2m



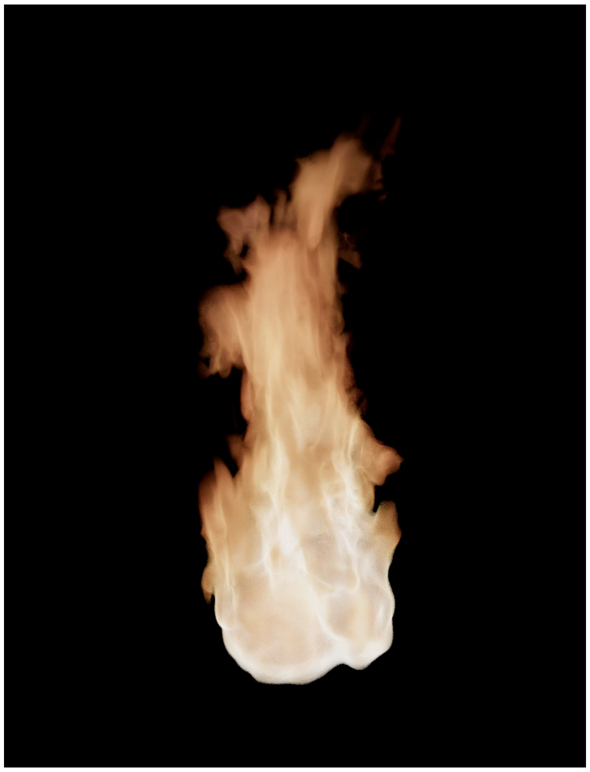
1 sample per voxel



2 samples per voxel, adaptive refinement



16 samples per voxel



64 samples per voxel

4.4. Empty space optimization

Depending on the input to the renderer, raymarching may constitute a large or small portion of the overall render time. As an example, scenes involving expensive rasterization primitives may take several times longer to compute than the final raymarch, whereas procedural volumes are next to instantaneous to create, but potentially expensive to evaluate at raymarch time. One of the easiest ways to reduce the time spent in the raymarcher is to optimize away as much as possible of the empty space in the scene.

So far, we have always intersected each ray against the domain of the voxel buffer, or whatever volume constitutes the scene to be rendered. This implies that all parts of the domain are equally important to sample, and that all parts could potentially contribute to the final image. Of course, most volumes have some number of zero-values in their domain (although some do not, for example homogeneous fog). The challenge lies in determining which areas contain data, without testing every possible sample location. Fortunately, depending on the input type, the cost of evaluation and the data structures used, there are often ways to quickly analyze which areas may excluded.

4.4.1. Level sets

Certain volume types are especially expensive to evaluate, without necessarily having very high Nyquist limits (i.e. required sampling frequencies). As mentioned in the *Adaptive raymarch quality* section, simulation buffers for fire are one example: the shaders normally used to render a simulation's temperature and density buffers contain sudden transitions which give the fire its sharp features. Deformation blur is also required when rendering fire, making the evaluation of a raymarch even more expensive. For data sets of this type, it may be less expensive to visit each voxel of the buffer once, evaluating its shader and velocity, marking it as either empty or contributing, then rendering the optimized scene, than to blindly raymarch the entire scene.

Level sets are especially useful in this context. They work both for storing the information, for providing a simple construction method, and for performing efficient ray intersection tests. We simply create a level set with the same domain as the simulation buffer, sample each voxel's shader result and velocity vector, and if the shader is non-zero we rasterize a sphere of radius abs(*v) * dt* into the level set. During rendering, for each ray fired by the raymarcher, we simple intersect it using a root-finding algorithm to find a more efficient raymarch interval than the simulation domain provides.

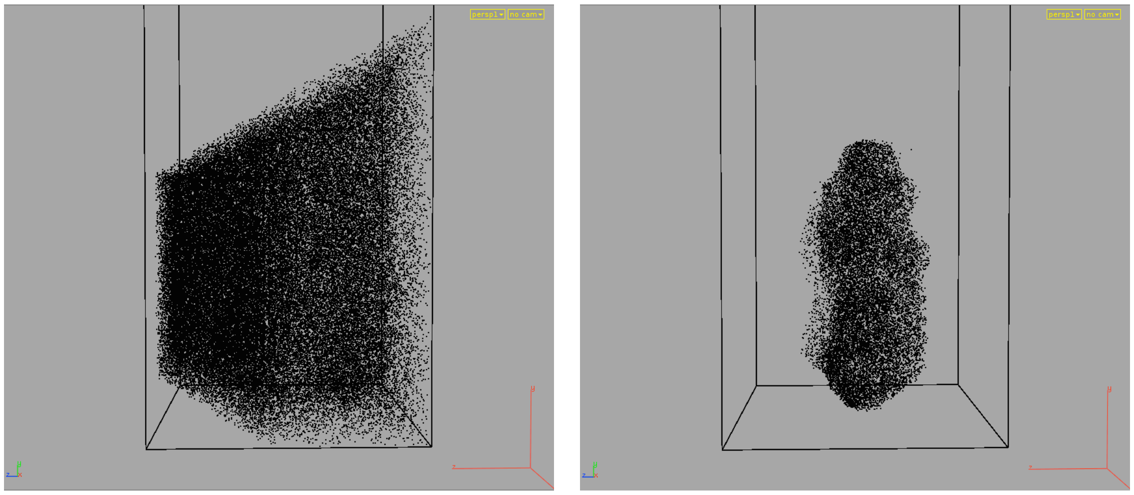
The following pseudo-code implements construction of a level set that also takes into account motion blur:

```
void calculateLevelSet(const DenseField<float> &temp, const DenseField<float> &dens,
                       const DenseField<V3f> &v, const Shader &shader, LevelSet &outputLs)
{
   outputLs.setSize(dens.size());
   outputLs.setMapping(dens.mapping());
   for (int k = 0; k < size.z; ++k) {
      for (int j = 0; j < size.y; ++j) {
         for (int i = 0; i < size.x; ++i) {
            Color emission = shader.eval(temp.value(i, j, k), dens.value(i, j, k));
            if (max(emission) > 0.0f) {
               Vector motion = v.value(i, j, k) * globals.dt();
```

```
              float radius = max(1.0, motion.length() / outputLs.voxelSize());
              outputLs.writeSphere(i, j, k, radius);
            }
          }
        }
      }
    }
```

While it may seem expensive to check the value of each voxel, it is important to remember that the raymarcher will be interpolating values at least the same number of times, and often more. Interpolations are much less cache-friendly than traversing voxels directly, and in practice the time spent generating acceleration structures using this method is at least an order of magnitude less than an unoptimized render.

The images below show how the distribution of sample points is improved during final rendering:

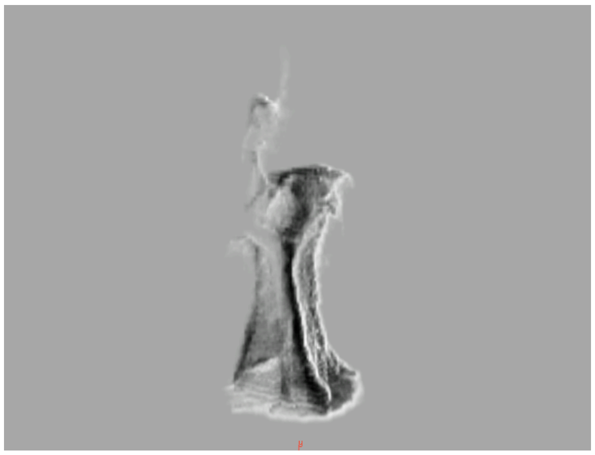


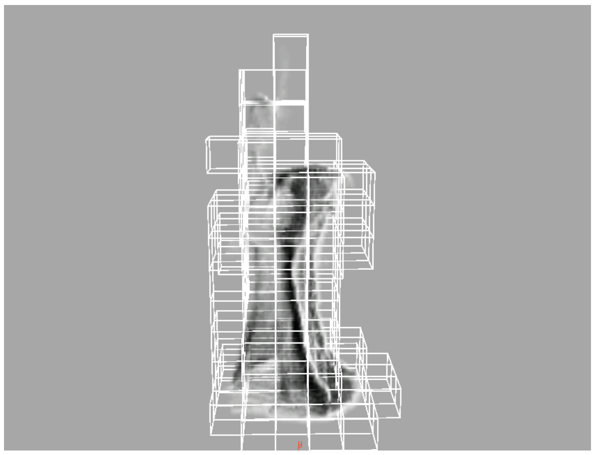
Plot of raymarch samples after intersecting primary rays against the simulation domain (left) and against a level set (right)

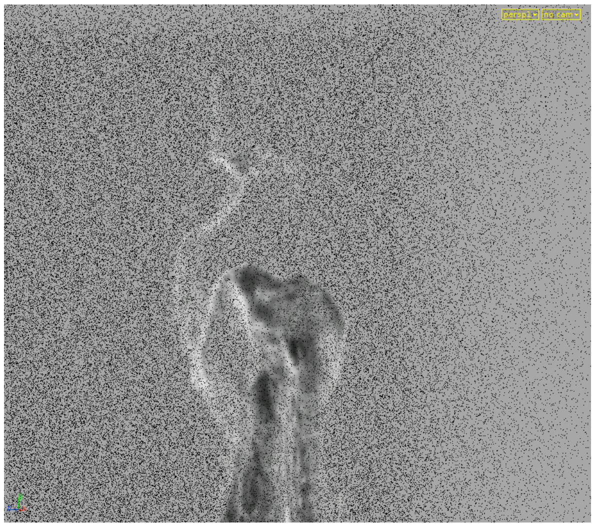
4.4.2. Using the buffer's data structure

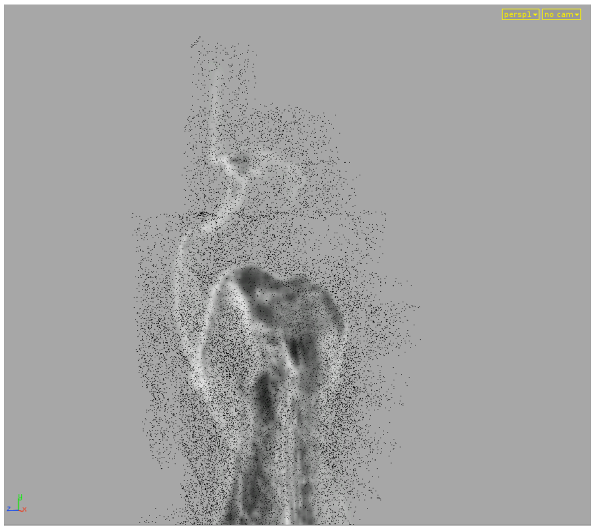
If the voxel data structure itself optimizes away unused space it is often possible to take advantage of that information during raymarching. We will examine the case of sparse blocked (tiled) arrays here, but the example extends to many other structures as well.

Intersecting an array of blocks is straightforward, and can be implemented either by testing the bounds of each block individually against the ray, or by using a 3D line drawing algorithm to find the blocks that overlap the ray's path.


Fluid simulation element


Wireframe display of sparse block bounds


Raymarch sample placement, primary ray intersected with buffer domain


Raymarch sample placement, primary ray intersected against sparse blocks

The added cost of a more complex intersection test is far outweighed by the improvement in speed gained from sampling less. Even for mostly-full data sets the cost of ray intersection is negligible compared to that of raymarching a single ray.

4.4.3. Frustum buffers

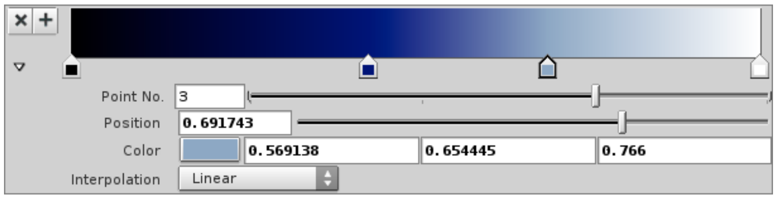
Because frustum buffers by definition have one voxel axis perpendicular to each camera ray we can perform some pre-processing that analyzes which voxels along each ray path contains non-zero values, and use that for the `RaymarchInterval` when queried by the raymarcher. As mentioned before, visiting the voxels once to build this acceleration structure is much cheaper than performing the interpolated lookups that the raymarcher does during final rendering.

For densely allocated voxel buffers there is no option but to visit each voxel along each scanline, but just like the example above sparse data structures provide valuable information about which voxels have data in them. For the SparseField data structure we simply need to check the contents of each SparseBlock, improving the generation speed of the acceleration map by several orders of magnitude.

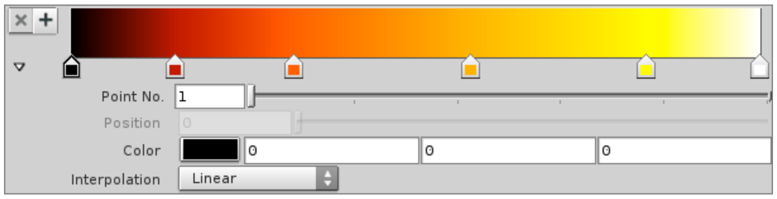
Once the acceleration map is built, it can be represented as a pair of scalar 2D images, one for the first intersection distance, and the other for the end of the interval. For a dense buffer of high resolution (2048 × 1556 × at least 100 slices), acceleration map generation can take a bit over 10 seconds, but for sparse buffers the generation time is just a fraction of a second.

4.5. Blackbody radiation

When rendering emissive phenomena such as energy, fire, etc., the look is often achieved by the artist through the use of *color ramps*. These let the user remap an attribute[3] to a color value. In the volumetric shading context this is often used to drive the emission and opacity (absorption) of the volume, which makes it a very flexible tool with lots of control for the artist.


Point No. 3
Position 0.691743
Color 0.569138 0.654445 0.766
Interpolation Linear

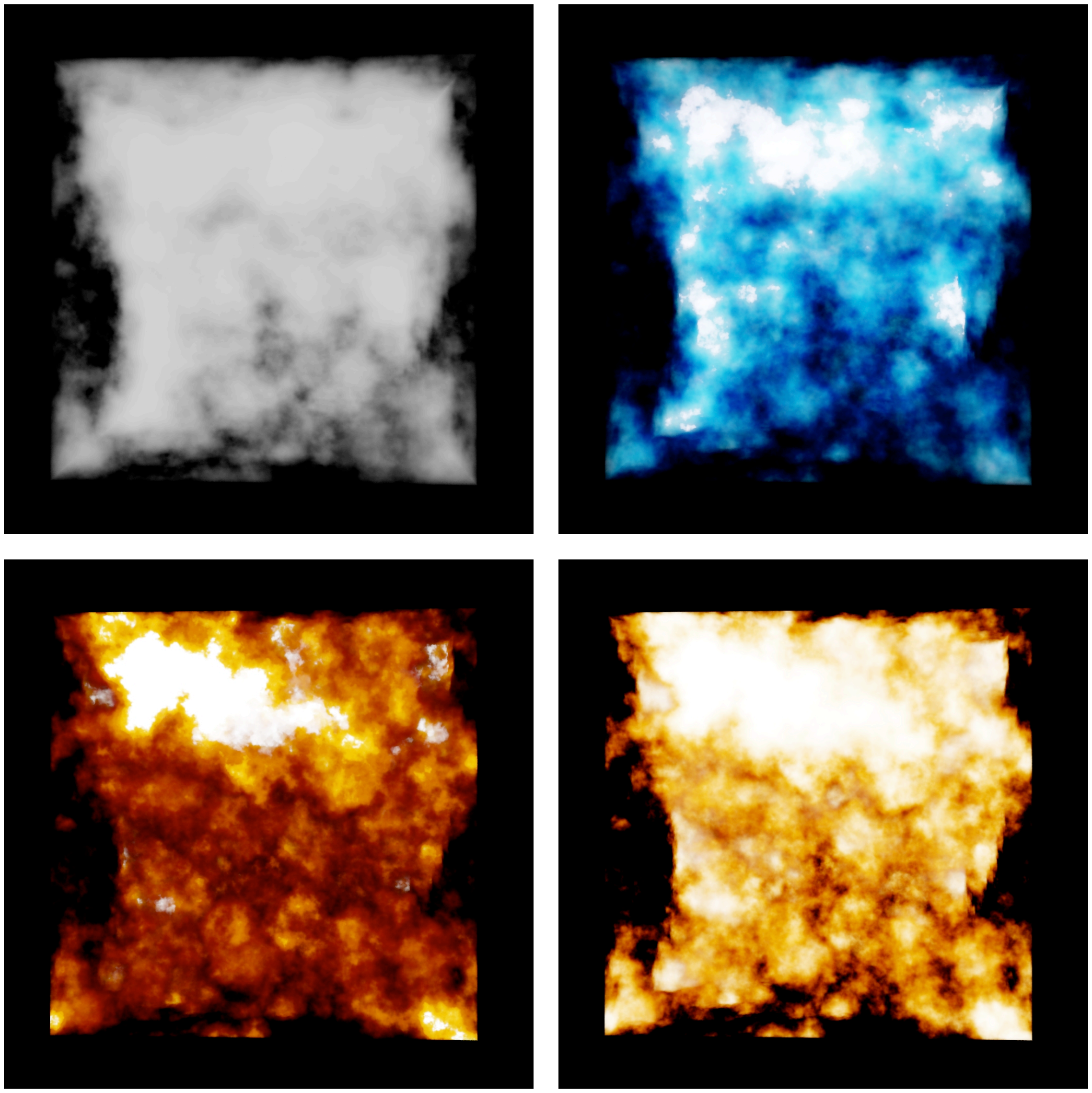
A color ramp used to achieve an energy look


Point No. 1
Position 0
Color 0 0 0
Interpolation Linear

A color ramp used to simulate a fire look

[3] e.g. density or temperature

While color ramps have the advantage of being infinitely tweakable, the amount of control can also be a drawback. Fire tends to have a very particular color range, and when trying to model its look, it can be difficult to place the color values in such a way that the rendered result looks realistic. Also, broad changes where a range of values needs to be shifted can be difficult, as each sample is individually editable by the user.



Top left: Render of density buffer
Top right & bottom left: Color ramp shader
Bottom right: Blackbody shader

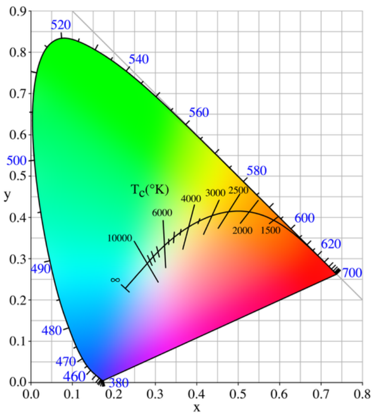
4.5.1. Physical background

A different approach to reproducing the look of fire is to use a physically based model. The law describing how a body emits light when heated is well known, having been discovered by Max Planck in 1900[4].

$$R_\lambda = \frac{C_1}{\lambda^5 \left(e^{C_2/\lambda T} - 1\right)}$$

$$C_1 = 3.7402 \times 10^{-16}\,\mathrm{W/m^2}\,,$$
$$C_2 = 1.43848 \times 10^{-2}\,\mathrm{m\,K}\,.$$

Planck's law of blackbody radiation (emitted power as a function of wavelength and temperature)



Plot of the color/chromaticity values assumed for various temperatures. The arc indicates the colors assumed by blackbody radiators.

A real-world example of blackbody radiation: The observed color of lava is directly related to its temperature

Using Planck's law, we can compute the intensity and spectrum of light emitted by a volume assuming we know its temperature. Although Planck's equation is not in a form that is suitable for plugging into a shader, with a few steps we can turn it into a function that transforms a *temperature* value into an *emission* value.

At Imageworks, Svea does not deal with specific physical units, so the example here would not be satisfactory to a physicist; however, for graphics purposes they should suffice. For those interested in more specific details wikipedia[5] is a good starting point.

[4] http://en.wikipedia.org/wiki/Planck%27s_law

[5] http://en.wikipedia.org/wiki/Black_body

4.5.2. Making the shader

The traditional way of shading fire has been to use color ramps in various combinations to control emission and opacity to achieve the appropriate look. With Svea, it has been our standard method since *Ghostrider* and *Beowulf*. During the production of *Green Lantern* we needed to render some one-off CG explosions and found ourselves spending too much time tweaking the control points of the shader curves to address specific notes, and in the process realizing that too much tweaking compromised the realism of the shader.

To help speed up the workflow, Anastasio Garcia Rodriguez (one of our effects software developers) developed a blackbody shader in Svea that quickly gave us a realistic look without having to manually design color gradients. In the process of finaling the shots, we developed some extensions to the shader that, although no longer physically based, gave the users control over very specific aspects of the look, without breaking the natural color response of the blackbody model.

The first step in writing a blackbody shader is to map a temperature value to an RGB linear color value. This is done by first finding the CIE color space value for the given temperature, and then converting from CIE space to (s)RGB.

```
Color blackbodyToRGB(float temperature) {
  // Find CIE XYZ color for the given temperature
  Color XYZ = blackbodyToXYZ(temperature);
  // Convert CIE XYZ color to RGB (sRGB color space)
  return XYZtoRGB(XYZ);
}
```

Once we know how to find the appropriate color, a simple (hypothetical) shader could be implemented as follows:

```
class BlackbodyShader : public Shader
{
public:
  Color shadeEmission(Vector P) {
    return blackbodyToRGB(m_temperature->sample(P));
  }
  Color shadeAbsorption(Vector P) {
    return m_density->sample(P);
  }
private:
  Volume *m_density;
  Volume *m_temperature;
};
```

Finding the *CIE XYZ* color is done by integrating Planck's law with the standard observer[6] values of human vision:

```
Color blackbodyToXYZ(float temp) {
  Color XYZ(0.0);
  int step = 10; // nanometers
  for (int w = startWavelength; w < endWavelength; w += step) {
    float I = planck(w);
    XYZ[0] += I * stdObserverX(w);
    XYZ[1] += I * stdObserverY(w);
    XYZ[2] += I * stdObserverX(w);
  }
  return XYZ;
}
```

Since the computation of the integral is somewhat costly and the resulting values are smooth, it is common to pre-compute the result and store the result in a 1D LUT. Once pre-computed, values are then interpolated at render-time.

```
Color blackbodyToXYZ(float temp) {
  return blackbodyLUT.interpolate(temp);
}
```

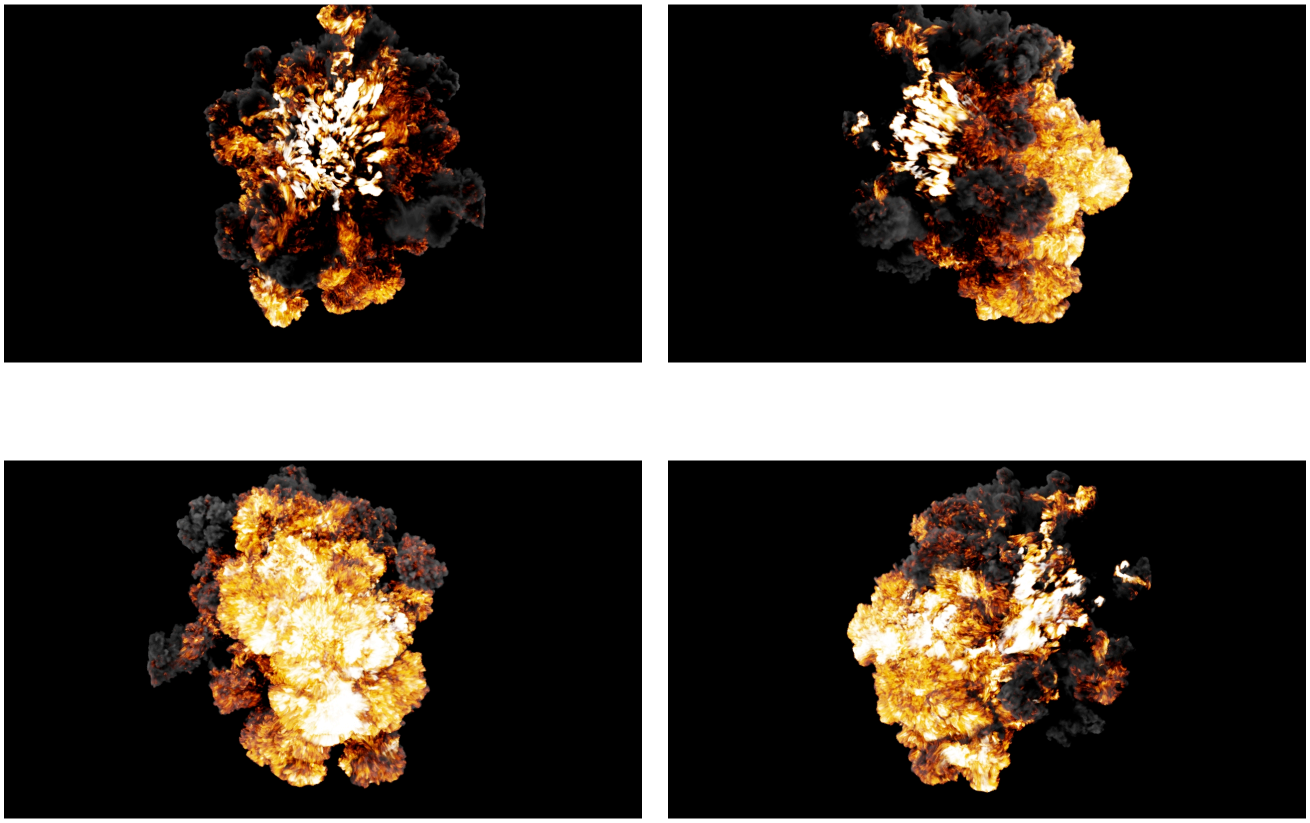
The conversion from *CIE XYZ* color to *sRGB* is a simple matrix transformation[7]:

```
M33f XYZtosRGB = {
  3.2404542, -1.5371385, -0.4985314,
 -0.9692660,  1.8760108,  0.0415560,
  0.0556434, -0.2040259,  1.0572252
}

Color XYZtoRGB(Color XYZ) {
  return XYZ * XYZtosRGB;
}
```

Using these simple operations, the resulting color spectrum accurately mimics the emission of light from hot objects and media alike.

6 http://en.wikipedia.org/wiki/CIE_1931_color_space#The_CIE_standard_observer

7 http://www.brucelindbloom.com/index.html?Eqn_RGB_to_XYZ.html



Multiple views of an explosion element rendered with a blackbody shader

The images above show an example of a fluid simulation rendered with our blackbody shader. As described so far, the shader follows physics quite closely, but in order to give the user some control over the appearance of the shader it was necessary to add parameters and in the process break some of the physicality of the shader.

The intensity (the integral of the entire power spectrum) produced by the blackbody radiation equation is described by the Stefan-Boltzmann law[8], which states that *I* is proportional to the fourth power of the temperature, scaled by a constant σ. This means that objects quickly get very bright as their temperature increases.

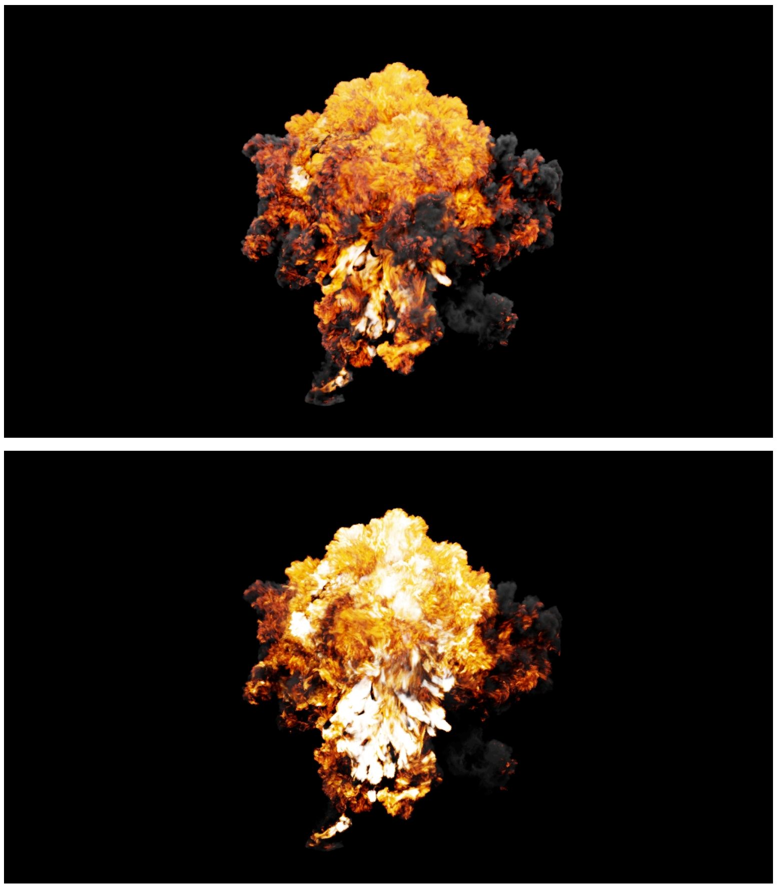
$$I = \sigma T^4$$

$$\sigma = \frac{2\pi^5 k^4}{15h^3c^2} = 5.67 \times 10^{-8} W/m^2 \cdot K^4$$

The Stefan-Boltzmann law

[8] http://en.wikipedia.org/wiki/Stefan-Boltzmann_law

This drastic increase in intensity with rising temperature looks realistic, but to the artist it may be desirable to control it. The first step in doing so is to use *CIE xyz* color (chromaticity), which is the normalized value of any *CIE XYZ* color. We can thus separate the *color* of the emission from the *intensity*. Once those are separated, we can introduce a 'physical intensity' slider in the shader which blends between the two values, letting the user dial how bright the hottest parts of the fire are in relationship to the cooler parts. Note that this is different from just multiplying the intensity since it effectively blends between $I = \sigma T^0 = \sigma$ (constant intensity) and $I = \sigma T^4$ (physically correct intensity). See the figure below for an illustration of the different appearances.

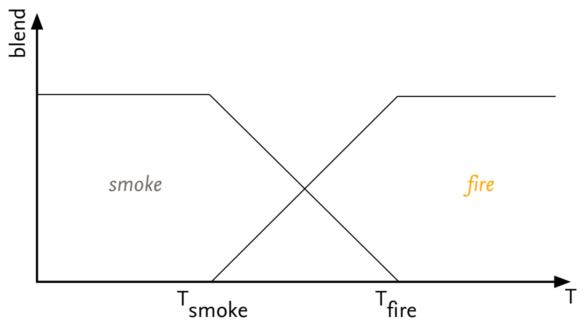


Constant intensity (chromaticity) vs. physically based intensity

Another part of the shader that needs more specific user controls is the 'cold' part of the simulation data. The cold parts, i.e. where temperature is near zero but density is non-zero, represents areas filled with smoke. Because they have very low temperature the blackbody model would say that they neither emit nor reflect light[9]. Artistically, however, we usually want the smoke to have some scattering properties, as most real smoke does reflect a small amount of light. In terms of rendering properties, we want there to be a small amount of scattering, and a large amount of absorption. This scattering behavior should only apply to the smoke, as we do not want any scattering properties in the fire portion of the shader.

[9] A blackbody, by definition, perfectly absorbs all wavelengths

Our solution to dealing with this is to split the shader logic in two parts: one that handles the *fire* part of the shader, and another one for the *smoke*. The temperature value is used to blend between the two, to make a smooth transition from smoke to fire. The user controls the blend by specifying two temperature values: T_{smoke}, below which the shader only renders the smoke appearance, and T_{fire}, above which the shader only computes the blackbody behavior. If the temperature value falls between T_{smoke} and T_{fire}, the output from the shader is blended accordingly.

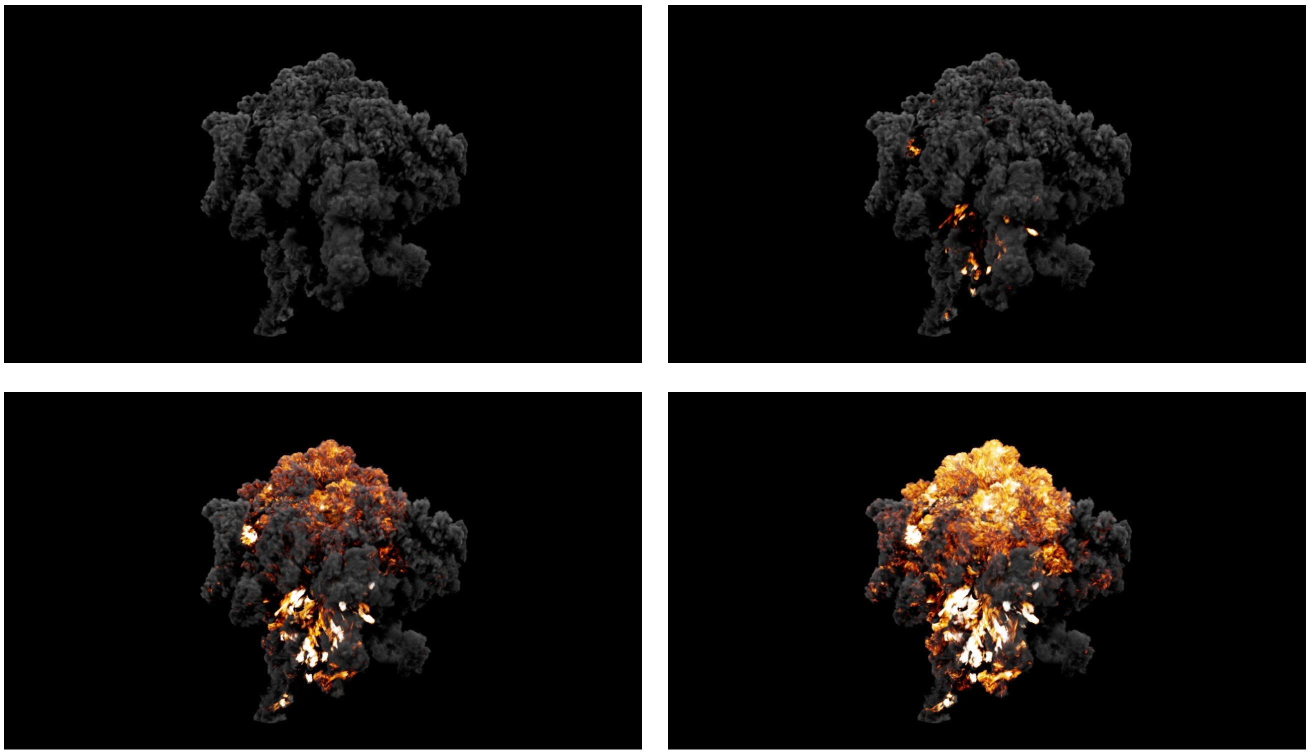


Blend functions for each shader portion

Extending the earlier example, the pseudocode is now:

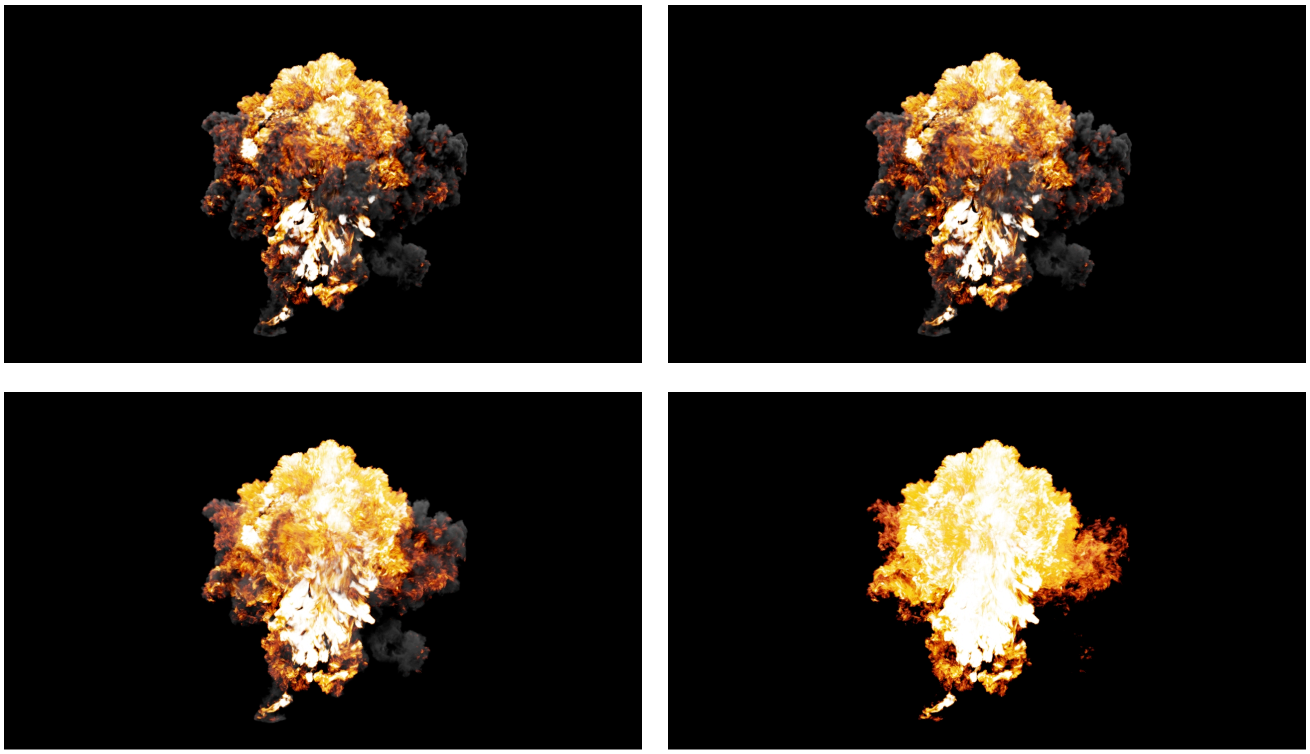
```
class BlackbodyShader : public Shader
{
public:
  Color shadeEmission(Vector P) {
    float temp = m_temperature->sample(P);
    return fit(temp, m_tempSmoke, m_tempFire, 0.0, blackbodyToRGB(temp));
  }
  Color shadeScattering(Vector P) {
    float temp = m_temperature->sample(P);
    return m_density->sample(P) * fit(temp, m_tempSmoke, m_tempFire, m_smokeScattering, 0.0);
  }
  Color shadeAbsorption(Vector P) {
    float temp = m_temperature->sample(P);
    return m_density->sample(P) * fit(temp, m_tempSmoke, m_tempFire, m_smokeAbsorption, 0.0);
  }
private:
  Volume *m_density;
  Volume *m_temperature;
  float m_smokeScattering; // Default: 0.1
  float m_smokeAbsorption; // Default: 0.9
  float m_tempSmoke;       // Default: 500K
  float m_tempFire;        // Default: 900K
};
```

The user also has control over the overall temperature of the volume. This is often used to bring temperature data that may have been simulated in a normalized range (say, [0,1]) into the Kelvin scale used by the shader. The images below illustrate how increasing the overall temperature brings more of the simulation into the fire portion of the shader, with successively brighter and red/yellow/white colors.



Successively increasing the temperature of the simulation data

The shader also provides control over the opacity of each portion of the shader. For example, the smoke may be rendered with much higher or lower opacity than the fire itself, in order to achieve particular looks. The image sequence below illustrates how decreasing the smoke density reveals more and more of the fire portion, making the final image brighter.


Decreasing the smoke density

Blackbody shaders can be a great help in creating realistic-looking fire, especially from fluid simulation data where hot/cold areas already behave appropriately. In its raw form, blackbody radiation can be difficult to control, but it can be wrangled into a semi-physical form where the user has enough control to easily change certains aspects of a render while maintaining a realistic look.

5. Production examples

The following section aims to show how all the techniques that are described in this course are used in actual shot production. Rather than describe exactly how each effect is accomplished from animation on, we will focus on which volume rendering techniques were used, why they were used, and how they helped accomplish the final result.

5.1. Hancock – Tornadoes

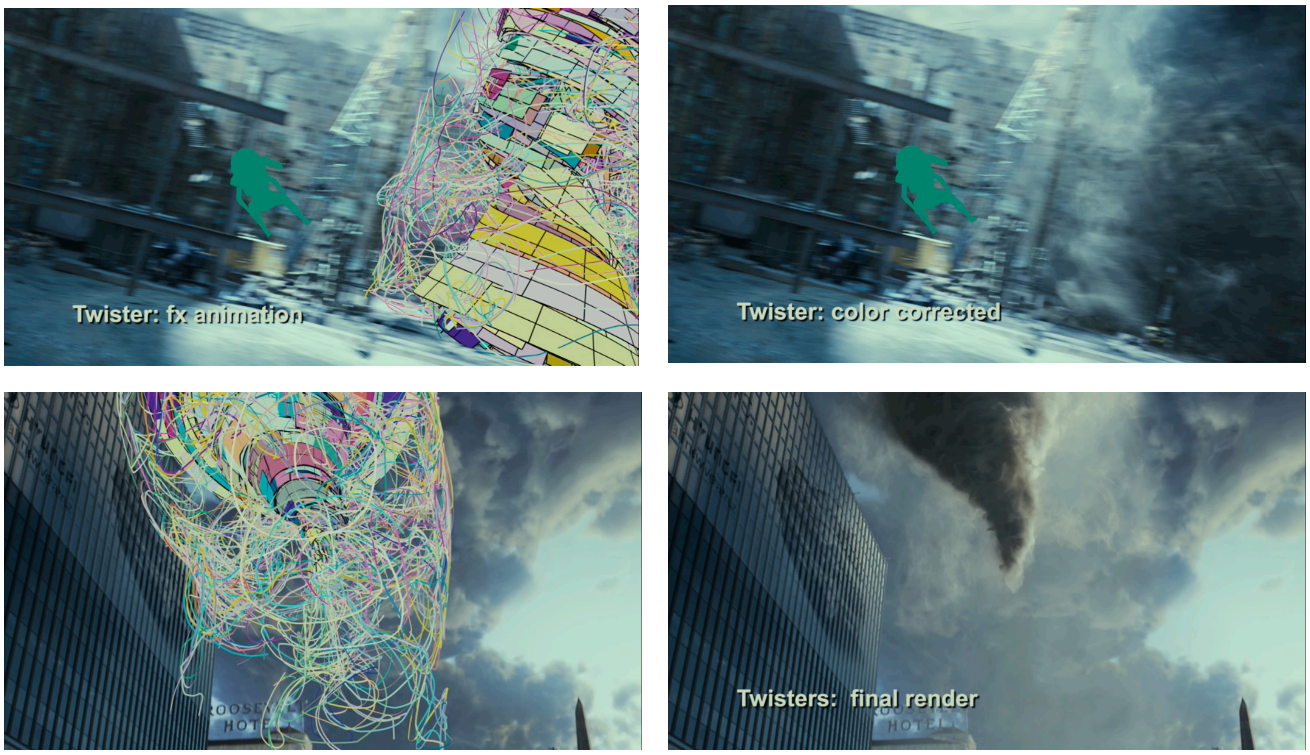
One of the major effects sequences in Hancock features tornadoes sweeping down on Hollywood Boulevard. The storm clouds in the background were modeled using existing rasterization primitives but the volume renderer had to be extended on several fronts in order to accomplish the tornado effects.

The first new development was the Generator concept, which is how Svea implements instantiation-based primitives. A surface-based primitive was used to model the core of the tornado, and a curve-based one was used to create wispy features that tore off the main funnels. The illustrations on the next page show some examples of how the look was accomplished with only a few hundred primitives per funnel.

Final shot from Hancock. © 2008 Columbia Pictures. All rights reserved.

Keeping the primitive count low helped maintain interactivity in Houdini and allowed the effects artists to fine-tune animation without having to run expensive simulations. The rasterization was, in contrast, quite expensive and shots with full-frame tornado funnels took several hours for the voxel buffers to compute.

The second big R&D task was to figure out how to store and render the multiple (often over 20) different high-resolution frustum buffers used in a single scene. A block-based sparse data structure was written (this later served as the foundation for `Field3D::SparseField`), which reduced the memory use for individual frustum buffers greatly. But even reducing the memory use of an average full-frame frustum buffer (roughly 2048 × 1556 × 400 × 16 bit) from almost 2.5GB to around 500MB meant that only around 10 could be loaded at once if enough memory was to be left for other rendering tasks. To solve this an out-of-core memory model was developed, using an LRU cache to decide which blocks would stay in-memory. Using this scheme it was possible to set a fixed cache size (often less than 1GB) which was then shared between all loaded buffers, keeping memory use under control.

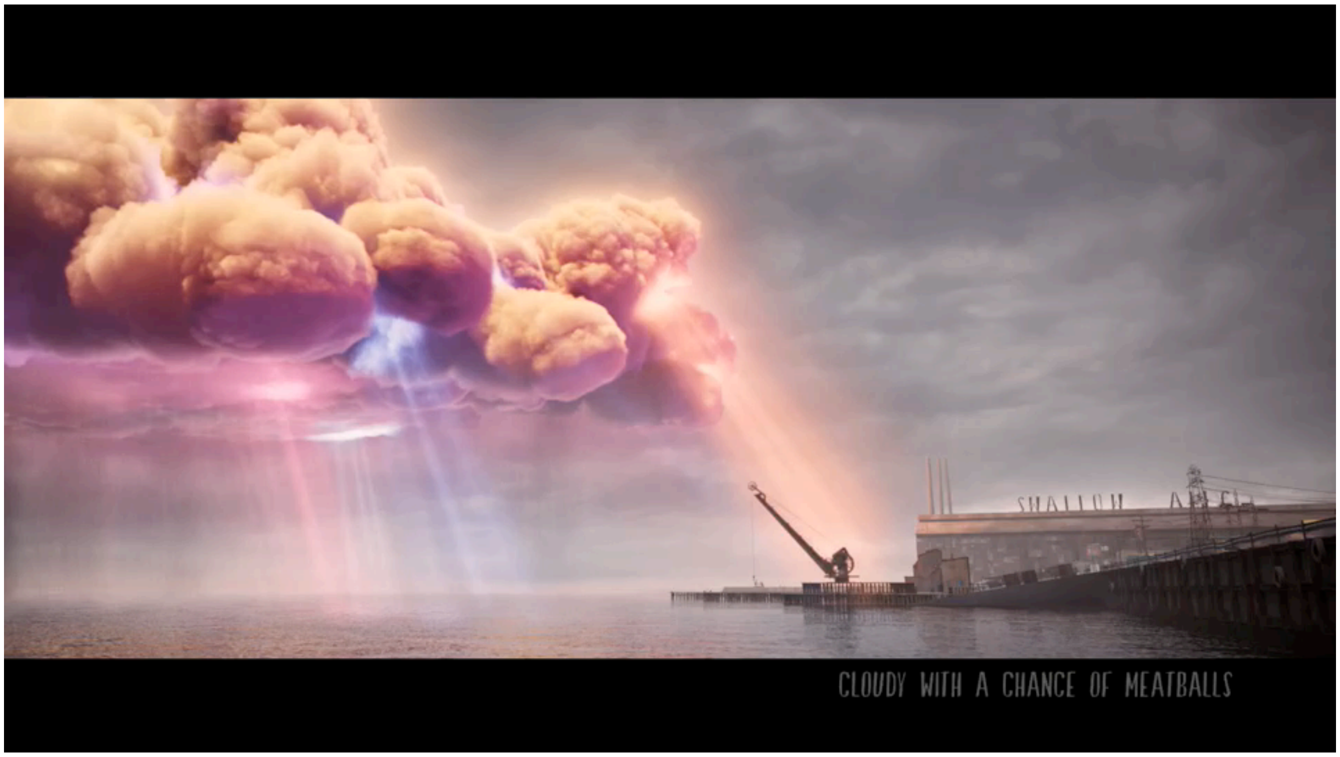


Illustrating the use of surface and curve primitives.
© 2008 Columbia Pictures. All rights reserved.

5.2. Cloudy With a Chance of Meatballs – Stylized clouds

One of the more prominent volumetric effects in Sony Animation's *Cloudy With a Chance of Meatballs* is the "Dock" sequence, where the first food-producing clouds sweep in over the city, in this case raining hamburgers. The clouds were highly stylized, literally forming hamburger shapes, but even though they weren't intended to be photorealistic, they had to be highly detailed and highly art directable both in terms of animation, modeling and lighting.

After experimenting with point-based rasterization primitives the artists found that it didn't give them enough control to create continuous features across the surface of the clouds. Level set-based approaches gave more control in shaping the clouds, but weren't intuitive and interactive enough. Instead, surface-based volumetric primitives were constructed out of the geometry handed off by the animation and modeling departments. The instantiation-based primitives were driven by attributes that could be visualized interactively in Houdini, and could be manipulated using tools already familiar to the artists.



Hamburger clouds from »Cloudy With a Chance of Meatballs«.
© 2009 Sony Pictures Animation Inc. All rights reserved.

CLOUDY WITH A CHANCE OF MEATBALLS

Hamburger clouds from »Cloudy With a Chance of Meatballs«.
© 2009 Sony Pictures Animation Inc. All rights reserved.

5.3. Alice In Wonderland – Absolem, the smoking caterpillar

Final shot of Absolem – the smoking caterpillar.
Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

Perhaps the most technically challenging part of Alice In Wonderland (for the effects department) was the simulation and rendering of Absolem's smoke. The production design and concept artwork suggested an incense-like smoke, with sharp features that faded away but did not diffuse.

The first problem regarded the representation of the smoke. In many shots the smoke covered the entire frame, which meant that resolutions would have to be at the very least 2K across, and in many cases much higher.

The second problem regarded the simulation of the smoke. Although some initial tests indicated that simulations run at low resolutions (i.e. the resolution of the simulated velocity field) gave the fluid animation a more appropriate look for the scale of the scene, there was still the problem of providing enough resolution in the visual result to maintain completely sharp features. To solve the problem a hybrid field/particle advection method was developed. Grid-based advection methods need to calculate advection in all voxels in the domain because it is non-trivial to determine which voxels will have density flowing into them. On the other hand, advection of particles is well suited to sparse volumes, because they only require advection calculation of parts that contribute to the visual result of the simulation, i.e. the locations where particles exist). Still, a hybrid approach was used because the smoke field had to be coupled to the underlying velocity field simulation (for rising smoke to behave correctly), and density sources and sinks were specified as density field inputs. The voxelized representation of the particles was created by splatting of the particle density values into a `Field3D::SparseField`, and changes to the grid representation were applied to the particles using a FLIP-like[10] algorithm by calculating per-voxel derivatives after the application of grid modifications. The voxel buffer is also what was written to disk for final rendering.

Rendering the high resolution voxel buffers required a few developments both on the file format and on the Svea side. The Least-Recently-Used cache scheme used on Hancock for reading of sparse fields was replaced with a Clock cache[11], and incorporated into the I/O routines of `SparseField` in the Field3D library. The block structure of each `SparseField` was then used to optimize the raymarch interval of each ray, so that only areas with density present had to be sampled. This reduced the render times of the highest resolution simulations from more than 10 hours to under 90 minutes. Apart from overall density adjustments, the voxel data was rendered without any shaders applied.

The highest resolution voxel buffers used were over 4000 × 4000 × 3000, but sparse enough that they could be simulated in under 5 minutes/frame using around 400 million particles, and used just over 200MB of disk space.

Final shot of Absolem – the smoking caterpillar.
Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

[10] Yongning Zhu & Robert Bridson – *Animating Sand as a Fluid*

[11] http://en.wikipedia.org/wiki/Page_replacement_algorithm

5.4. Alice In Wonderland – The Cheshire Cat

Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

After some initial tests of the Cheshire Cat's "evaporating" effect it was decided that the look had to be very subtle, only highlighting the motion of the cat as he disappears and reappears. The idea was to make the effect look as if the transformation left a trail of substance behind, where the trail would carry the same color properties as the cat from each location it streamed off of.

Some initial tests of birthing particles from the surface were successful in carrying the surface properties from the cat, but could not achieve a smooth enough look, even when using several million particles. A different approach of advecting the cat's color properties directly into a fluid simulation proved too hard to control. Instead, the final solution used a combination of both techniques, with the addition of a couple of custom Svea plugins in order to handle point instantiation and plate projection at render-time.

The first step was to convert the cat's geometry to a level set so that points could be scattered uniformly throughout the inside and along the surface. A fluid simulation was then run, using the motion of the cat as a "target field" (i.e. kinetically driving the simulation), but without any collision geometry. The cat's motion was blended in an out in various areas to accentuate the motion as desired.

Using a combination of the simulation field and procedural noise fields, a particle simulation was created for the trail streaming off the cat, recording each particle's birth frame and location. A custom `Generator` plugin called Cluster2 was then written, which filled in the space between each particle and its neighbors smoothly. (Cluster was originally developed as a RenderMan DSO for creating white water effects on Surf's Up.) This resulted in turning the 50,000 or so original particles into tens of millions of instanced particles, giving a smooth final appearance while keeping simulation times at a minimum.

Instead of carrying the color properties of the cat on each particle, which would have resulted in a very blurry image, a `Filter` was implemented that performed texture lookups on each instanced point, after the Cluster instantiation was performed.

Finally, the simulated particles were rendered together with the points scattered inside the cat's body, giving a final result that had the trail element integrated into the final lighting element without having to resort to holdouts (and their potential artifacts). The compositor could then blend between the lit element and the volume render arbitrarily.

Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

5.5. Alice In Wonderland – A mad tea party

Atmospheric mist and fog was featured heavily in some sequences in Alice In Wonderland and it was clear early on that the modeling and placement of the elements would depend greatly on the lighting in each shot. Effects artists are usually responsible for modeling volumetric effects, but in this case it would have been too time consuming to accomplish an iteration if two departments had to be involved with each change to a shot.

Instead, a library of volumetric elements was built which was used by lighting artists as a set dressing tool. The library included both fluid simulations (for ground mist) and point clouds configured as rasterization primitives (for mist and background fog). These library elements were then wrapped up into Katana primitives which lighting artists could duplicate, reposition, and vary the settings of.

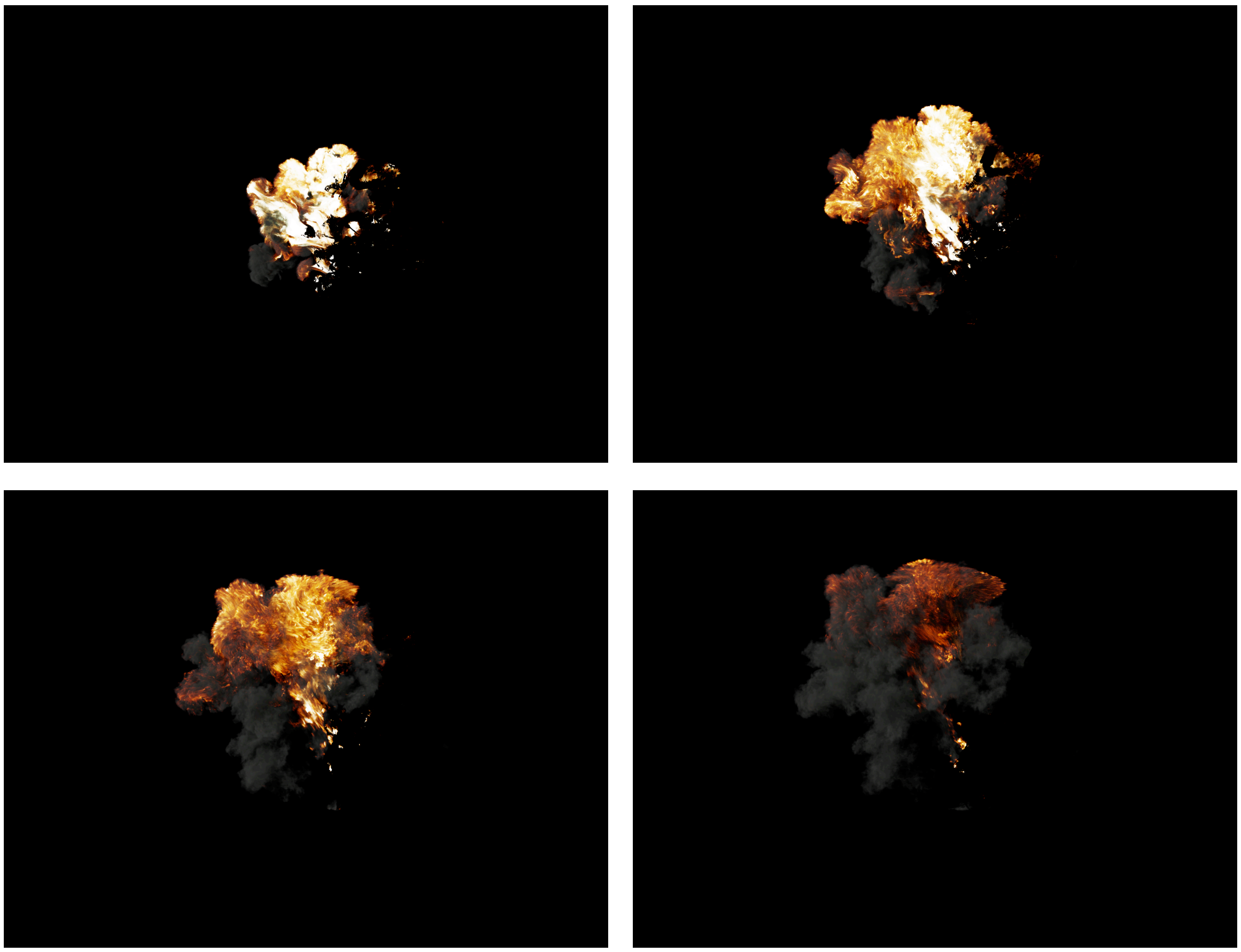
Deferring the rasterization of volumes meant that frustum buffers could be generated on-the-fly at render time, instead of being rasterized by effects artists and stored on disk. (Creating libraries of pre-baked voxel buffers usually forces the use of uniform buffers instead of frustum buffers.) This approach reduced the amount of disk space needed for the library, and because primitives were rasterized after being repositioned, there was always enough detail available – something that would be hard to achieve with baked-out voxel buffers. A separate set of "negative" voxel buffers were also provided, which let artists subtract density at render-time, for example around the table and along the path leading off into the woods.

The misty environment at the mad tea party was modeled and rendered by lighting artists using a library of volumetric elements provided by the effects department.
Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

Alice in Wonderland and artwork © 2010 Disney Enterprises, Inc. All rights reserved.

5.6. Green Lantern – Explosions

The blackbody shader described above was primarily used to render the CG explosions in *Green Lantern.* The explosions were all simulated using our in-house FLIP-based gas solver and rendered in Svea using the blackbody shader.



Raw explosion element from Green Lantern.

Some earlier explosion work (the engine core explosion in the beginning of the film) had used a color ramp-based shader, which worked well since the explosion was intended to be made up of energy. The rest of the explosions, however, needed are more physical look. To make matters more complicated, the shots were started late in the production schedule and the finals deadline was only a few weeks away.

It was quickly decided that a physically based model would get us close to a realistic result quicker than using the existing color ramp shaders, which then prompted the development of the blackbody shader.

Explosion element and final composite from Green Lantern.
© 2011 Warner Brothers. All rights reserved.

The user parameters that were discussed previously, especially the non-physical ones, were added as effects look development was underway, based on the comments that were received from the show supervisors. We often relied on the ability of the shader to adapt existing simulation data to address comments, rather than going back to re-simulate. This allowed us to quickly provide new iterations. Some typical comments were:

- *There's not enough fire.*
    - **A:** Increase the temperature multiplier to push more values into the fire portion.
- *The smoke is too dark and not thick enough.*
    - **A:** Increase the scattering coefficient of the smoke and the smoke density multiplier.
- *The fire doesn't look hot enough.*
    - **A:** Change the blend between pure chromaticity ($I = 1$) and physically based intensity ($I = T^4$).
- *The fire looks too much like lava.*
    - **A:** Reduce the opacity of the fire portion.
- *There's not enough detail.*
    - Alas, not everything can be fixed in the shader. This one usually meant going back to re-simulate.

Explosion elements and final composites from Green Lantern.
© 2011 Warner Brothers. All rights reserved.

5.7. Green Lantern – The Sun

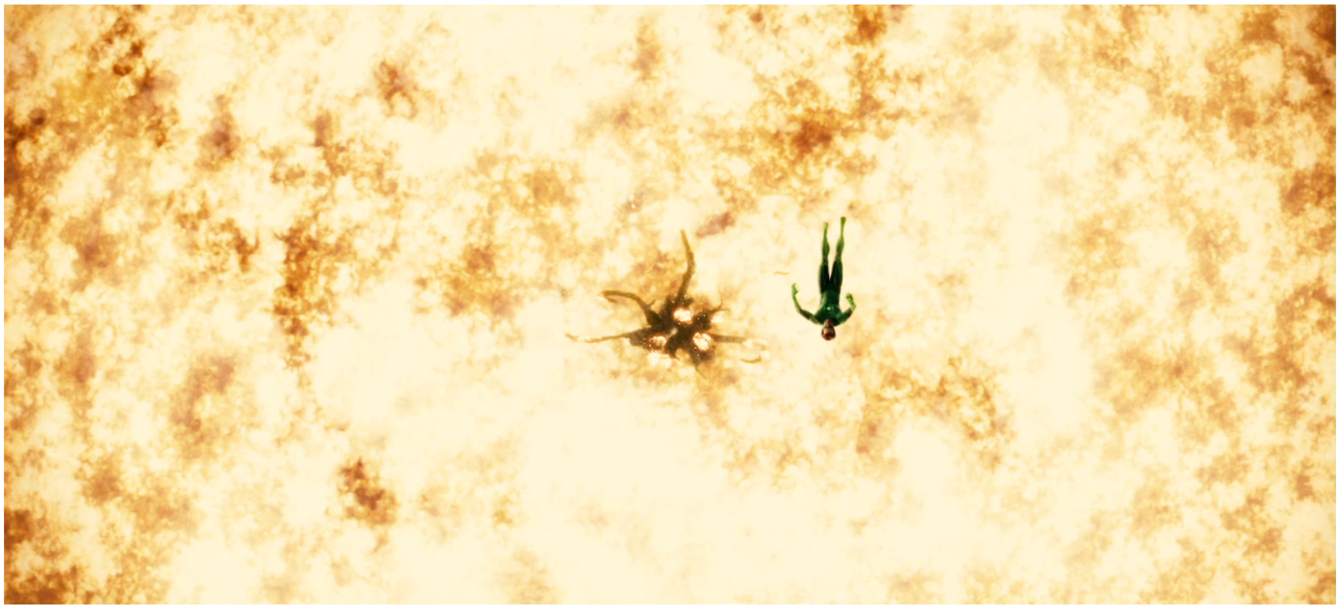
Green Lantern ends with Parallax (the intergalactic space villain) following Hal Jordan to the sun, where he is caught by its gravitational pull and falls to his demise. Just as with the explosions described in the previous section, the schedule was very short and the 40 or so shots that the sun is visible in had to be finished in just a few weeks.

In creating the effect, Joseph Pepper used the 'stamped voxel buffer' primitives extensively (see the Volume Modeling section). First, the overall layout of the sun was generated using various custom shaders in Arnold, which output a series of control maps that were imported into Houdini. The control maps were used to drive density maps and flare eruptions.

Using Houdini and our in-house fluid simulator FLU, 20 different base simulations were created in three groups (core, exterior, flare) and ~500 frames were written out and stored in a library that could be re-used across the entire sun surface.

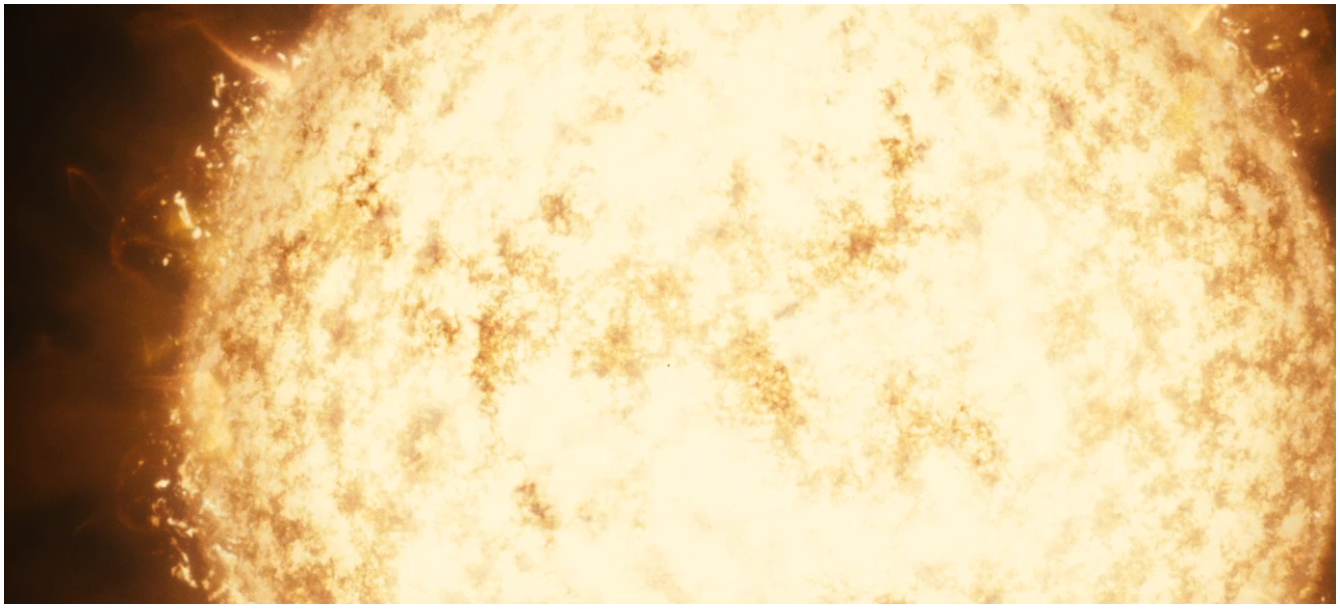
Using the control maps, between 10,000 and 30,000 particles were instanced in the three layer groups, determined by each shot's required coverage. The particles were then assigned a random simulation and offset frame, as well as a random orientation before being rasterized into the final frustum buffer.

Using Field3D's sparse data structure and caching system[12], a huge number of simulation buffers could be used in each shot. Often each of the 500 frames of each simulation was imported, which would have been impossible without the caching mechanism.



© 2011 Warner Brothers. All rights reserved.

[12] See the Field3D programmer's guide and source code for more details. http://sites.google.com/site/field3d/


© 2011 Warner Brothers. All rights reserved.

The exterior effects, such as far-reaching flares and nebula, were all modeled using Svea's basic instantiation primitives. Flares were modeled using splines and nebulae with surface primitives.

Being able to mix and match procedural geometry with simulation data turned out to be a good approach, with a single artist finishing the sun elements for all 30 shots in the sequence.

© 2011 Warner Brothers. All rights reserved.

DNB: Double Negative's Volumetric Renderer

Ollie Harding & Gavin Graham

August 2011

Abstract

DNB (***D**ouble **N**egative Renderer **B***) was developed in late 2004 for use in production on *Batman Begins*. The primary motivation was the need for a fast 3D volumetric renderer which, it was felt at the time, commercial renderers were unable to provide.

After the initial success, it was decided that DNB should be developed as a core in-house tool, available for use on all future shows. To date, DNB has been used on over twenty feature films including *2012*, *Inception* and five films in the *Harry Potter* series.

DNB was designed to fit into the 3D pipeline very much like Pixar's *RenderMan*. The renderer itself is a standalone command-line tool, with a comprehensive user interface provided as a *Maya* plugin.

This document presents the reasoning behind DNB's design and some of the key decisions which have shaped its development. It also discusses some examples of its use in production, and therein its place in DNeg's 3D FX pipeline.

Contents

1 Design History

When DNB v1 was designed, it had only a simple set of requirements:

- Rendering of Maya fluids
- Low memory footprint
- Simple design
- Fast rendering
- Controllable by way of customisable shaders
- Self-shadowing

DNB is very much a production evolved tool, continually changing to meet the needs of upcoming shows, and responding to feedback from use on those already completed. By the time the DNB v4 branch was retired in 2010 many new features had been added to its list of capabilities, including:

- Rendering of particle sets
- Motion blur (camera, fluid & particle)
- Dynamically generated isosurface normals
- Custom shading attributes
- Improved memory management
- Multi-threading
- Single & multiple scattering

Over time it has become apparent that some of the key points of the original specification have gradually become less and less relevant, in view of massive leaps in technology, client expectations, and developments in other parts of the FX pipeline. In 2009 a large scale redesign was undertaken, aiming to harness (and push to the limits) the vastly improved hardware capabilities available.

A key change to the design philosophy was driven by the fact that DNB has grown into a much larger application than the original architecture comfortably allowed for. The "simple design" requirement has been modified to a "simple interface to a complex architecture". This has allowed the reduction of what had become an extremely complex and rigid program flow into a set of simple modular components.

Along with a raft of architectural changes in v5 and v6, the major changes over v4 are:

- Improved multi-threaded program flow
- New, much more realistic 3D motion blur engine
- Generalisation of the render primitive concept and interface
- Dynamic attribute registration
- Arbitrary output drivers
- UI shader designer (*Meccano*)

The first release of DNB v7 is fast approaching, formed mainly of further development of improvements mentioned above. Each aspect of the rendering process in this newest version is discussed in more detail in the following sections.

2 Key Components

The DNB renderer, all of its plugins, and much of the Maya *functionality are written in C++. This and subsequent sections will combine discussion of conceptual, architectural and programatic topics with this in mind.*

2.1 Output Drivers

Given that the ultimate aim of a render is the output of data, the output drivers are the logical (if not the most obvious) starting point for discussion.

The `OutputDriver` class represents a black-box interface which outputs shaded and/or ray-marched data. At various stages throughout the render each output driver (there can be many) is presented in turn with a subset of the rendered data. The driver is then free to process it as appropriate, and the render continues on to the next stage.

The output drivers currently used in production include:

- **Open-EXR**: Writes an EXR[1] file of the final rendered image
- **Deep shadowmap**: Writes a deep shadowmap using the PRMan deep-texture file format
- **Blit**: Outputs to DNeg's single-frame image viewer as the render progresses
- **Point cloud**: Writes a single point for each voxel of interest, using PRMan's point-cloud file format

At each output stage the driver has access to the raw (i.e. not ray-marched) data for all voxels concerned; any required ray-march accumulation is performed internally to the driver. The Open-EXR and Blit drivers are very similar in that they perform the internal ray-march which, once complete, is rasterized into a 2D frame-buffer, and output as a simple RGBA image. The deep-shadowmap and point-cloud drivers are rather different, as they are concerned with a 3D representation of the render. The point-cloud driver typically outputs a selection of raw attributes from each voxel, whereas the deep-shadowmap driver uses the partially ray-marched opacity channel at each z-slice to output the required cumulative transmittance function.

2.2 Voxel Shader

The voxel shader is responsible for converting input attributes (provided by the render primitives) into the output channels required by the output drivers. The concept of this stage is simple, but the scope is practically unlimited; for this reason, the shader implementation is delegated to a plugin.

There is only one voxel shader assigned to any render pass, and in this sense it could be considered the "master" of the render, drawing together all the other components. Accordingly, setting up and fine tuning the voxel shader's structure and parameters is often the most involved part of a TD's task in the fluid rendering process.

No default voxel shader implementation is provided, as there are few scenarios in which it would be useful. However, a number of extremely powerful plugins are maintained centrally for widespread use – see §4.3 and §7 for more detailed discussions.

2.3 Primitives

The input attributes requested by the voxel shader are provided by render primitives. The currently supported primitives are `Fluid` and `ParticleSet` elements, which both derive from the abstract `RenderElement` interface.

[1]ILM's open-source *Open-EXR* image format is the primary image format used in DNeg's pipeline.

As discussed later on (§3.1.1), the input attribute sampling is performed in screen-space blocks (or *tiles*). For each tile, every `RenderElement` instance is called upon to add its data to the input attribute buffers. Typically, for a single tile, this would require:

1. caching some data from file
2. processing (or shading, see §4.2) that data
3. sampling the data into the tile's voxel buffer

2.3.1 Fluid Primitives

Fluid elements (voxelised data) are the most commonly rendered primitive type in DNB. As mentioned previously, the renderer was originally designed to solely render *Maya* fluids. At that early stage *Maya*'s *PDB* data format was used to store the simulated data, but the restrictive nature of the data storage and large file sizes soon led to the adoption of Tweak Film's open source *GTO*[2] format. This allowed for more flexibility, e.g. the optional reduction of floating point precision from 32-bit to 16-bit.

In 2005 DNB migrated to Double Negative's new *Zen* file format[3], which was extended to provide the *ZenVoxel* protocol and utilise custom compression techniques specifically developed for compression of voxelised fluid data. Files written using the *ZenVoxel* protocol are accessible in multiple LODs, and can be loaded in spatially coherent chunks which leads to improved memory efficiency.

Alongside developments in data storage, the sources of that data have continued to diversify. The primary source of fluids rendered through DNB is *Squirt*[4], DNeg's in-house fluid simulator. Additionally, *Maya* is still used occasionally to simulate some fluid effects, and *Houdini* is becoming ever more significant in generating and modifying voxelised data sets. Finally, a python binding of the *ZenVoxel* file interface is now available which, alongside various simple command line tools, allows TDs to quickly and easily analyse and modify their fluids.

2.3.2 Particle-Set Primitives

The second DNB primitive type is the particle-set. Although most final, high quality fluid simulations are particle based, these are more often than not output as a voxelised fluid and rendered as fluid primitives in DNB.

The great power of particle-set primitives is their instancing capabilities (figure 1). Often a simple particle simulation is used to instance either noise buffers or low resolution voxelised fluids many times into the scene, once per particle. Alternatively, particles can be used to inject attributes at carefully animated positions.

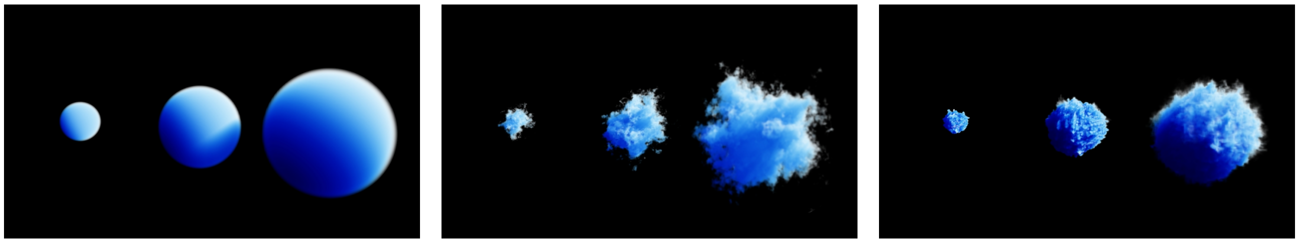

(a) **(b)** **(c)**

Figure 1: Comparison of typical particle instancing uses: (a) spherical fall-off, (b) instancing of dynamically computed noise buffers, and (c) instancing of cached fluid buffers.

[2]www.tweaksoftware.com/products/open-source-software

[3]The *Zen* file format was designed and and continues to be developed by Jonathan Stroud at Double Negative.

[4]The *Squirt* fluid simulator was originally developed by Marcus Nordenstam at Double Negative, and is now maintained by Ian Masters, Dan Bailey and Matthew Warner.

3 Program Architecture and Flow

In this section we'll follow the progress of a typical render through from start to finish, looking at some of the key architectural design aspects as they are encountered.

For the sake of example, let's say that this render will output two Open-EXR images (the *primary* and a *secondary* representing temperature). It will also output a point-cloud using the temperature attribute for use at some point later in the pipeline.

The required output attributes registered by the output drivers are therefore:

- `Oi`: the opacity channel
- `Ci`: the primary colour channel
- `Ci_temperature`: the secondary colour channel representing temperature
- `TEMPERATURE`: the raw temperature attribute

As the main primitive we'll use a voxelised fluid with `opacity`, `colour` and `temperature` channels. Additionally, we'll use a particle set with a `temperaturePP` which we'll use to control additional hotspots in the fluid.

In this case we'll use a very simple voxel shader implementation, shown in figure 2. The pink nodes on the left represent input attributes required from the primitives, and the blue nodes to the right represent outputs provided to the output drivers. From this it is clear that the render primitives must provide three input attributes: `OPACITY`, `COLOUR` and `TEMPERATURE`.

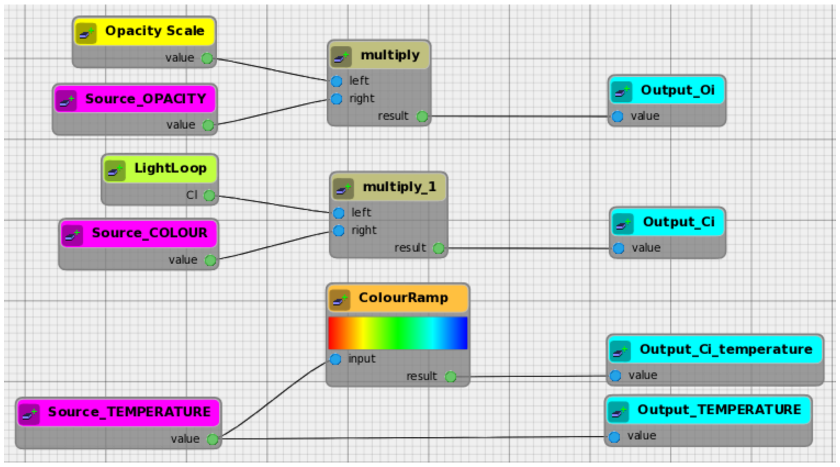


Figure 2: A simple voxel shader, providing outputs `Oi`, `Ci`, `Ci_temperature` and `TEMPERATURE` from the input attributes `OPACITY`, `COLOUR` and `TEMPERATURE`. The colour of the fluid is multiplied by the result of the light-loop, in which the incident light on a voxel is computed by combining the effects of all the lights in the pass.

3.1 Voxel Structure

After defining the attributes to use in the render, we next consider the geometry of the render and the data structures required to represent it.

DNB's voxel grid is aligned with the frustum, in such a way that the cells are indexed orthogonally in perspective space. The multi-threading and memory-management models employed require that the grid be split up into a number of screen-space tiles, implemented as instances of the `Tile` class which is responsible primarily for data ownership.

3.1.1 Tiles

A typical full-frame render resolution is $1920 \times 1080 \times 2048$ voxels, and tile sizes can vary from 8×8 to 64×64 voxels in x and y. The z resolution is always that of the complete grid. In the scenario under consideration, each voxel will require storage for 7 `float` channels (3 each for `OPACITY` and `COLOUR`, 1 for `TEMPERATURE`). Additionally, the nominal centre point is stored for each voxel. The total memory required per voxel is therefore 40 bytes, or ~158GB for the entire voxel structure.

This example uses the bare minimum of attributes, but even so it is clearly not feasible to allocate such a large amount of memory. Instead, the renderer is limited to allocating just one tile's attribute storage memory per thread of execution, and that memory is swapped between active tiles as and when required. For the standard six threads, and a relatively large tile-size of $64 \times 64 \times 2048$ this comes to a total of just 1.9GB, which is far more appropriate.

3.1.2 Camera Auto-Focusing

In some circumstances (e.g. when the voxel shader adds noise to the opacity), every voxel in the render may be considered "interesting", but for the most part this isn't the case. By default, the render's voxel structure is reduced to the minimum frustum which still encloses all the primitive elements in the pass (see figure 3). During this stage of the renderer's initialisation each primitive is given the opportunity to analyse its file headers and, if necessary, load some data from file (such as particle positions and radii) in order to accurately compute its bounding box.

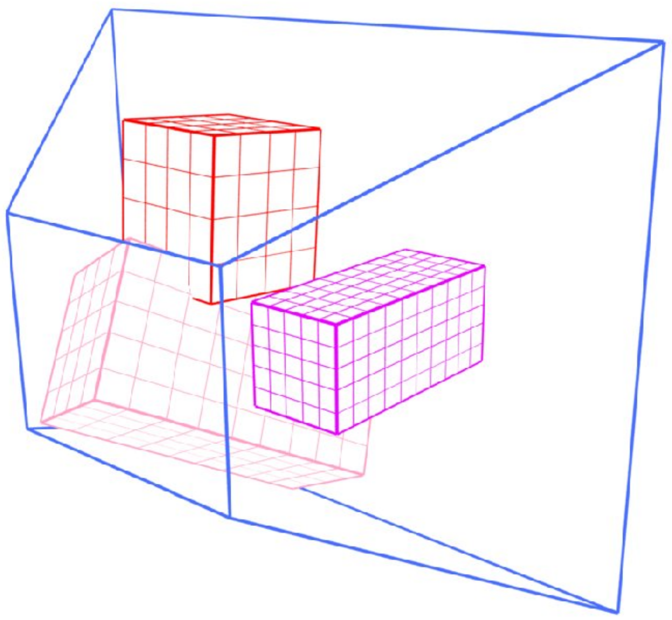


Figure 3: Autofocusing camera frustum on the fluid elements in a render pass.

Although the frustum is reduced in size, its voxels remain an exact subset of the full-frame frustum. This means that the voxel centres remain in exactly the same position in camera-space, regardless of any changes in the contents of the pass.

3.2 Task Generation

After the voxel structure, and therefore tile structure, is finalised the list of tasks needed to complete the render must be generated.

The render is split up into a set of distinct phases, each of which must be completed in order for each tile in the voxel structure. These phases are:

- **Proxy** Allows interactive output drivers (e.g. the Blit driver) to display a placeholder image showing the locations of each primitive element.
- **Fill** Sample input attribute data from the primitives.
- **Shade** The voxel shader uses the input attributes to compute the output buffers.
- **Motion-blur** The shaded output buffers are blurred according to camera motion and/or the voxelised velocity field.

Although the stages must be processed in order for each tile, they can easily be multi-threaded provided certain dependencies remain intact. This is the level at which multi-threading is implemented in DNB.

3.3 Fill Stage

3.3.1 Primitive Sampling

The first high-level tasks of any significant interest are the *Fill* tasks. These tasks are responsible for evaluating the render primitives and populating the voxel shader's input attribute buffers prior to shading.

The `ElementShader` assigned to each `RenderElement` is the object responsible for actually providing the data to populate the buffers. `FluidShader` and `ParticleShader` are derived from `ElementShader` and assigned to `FluidElement` and `ParticleSet` elements respectively. Although using the same abstract interface, the implementation of each of these specialisations varies massively. A detailed discussion of the fluid and particle shading mechanisms can be found in section §4.2.

During the render setup, each render primitive driver will have been notified of the input attributes required by the voxel shader. At that stage, the assigned `ElementShader` instances must make decisions on how the attributes will be provided. These decisions, often explicitly made by the user when setting up the shader, then define the attributes required from file.

As the data for each tile is populated, the `RenderElement` loads the required portions of data from file, and passes them to the shader to process. These portions of data might be a subset of the particles in a file, or the relevant cells of a *ZenVoxel* fluid. Examples of operations performed at this stage could be adding noise to particles' radii, or remapping voxels' opacities – there is no sampling at this point. See §4.2 for further discussion of file loading and primitive shading.

Once all the data has been prepared, the shader is instructed to fill a temporary buffer representing that primitive's addition to the accumulated data. In our example, we require the `RGB` attributes `OPACITY` & `COLOUR`, and the `float` attribute `TEMPERATURE`. It's fairly unusual for fluid shaders to do anything much more complex than re-mapping and combining attributes from the file; such a shader is shown in figure 4.

Particle shaders are often rather more involved than fluid shaders. For now we'll generate a noisy radial falloff channel for `TEMPERATURE`, but leave the other attributes un-modified (figure 5). This means that no additional opacity or colour will be added by the particles – they're simply used as a mechanism to control hot-spots in the fluid.

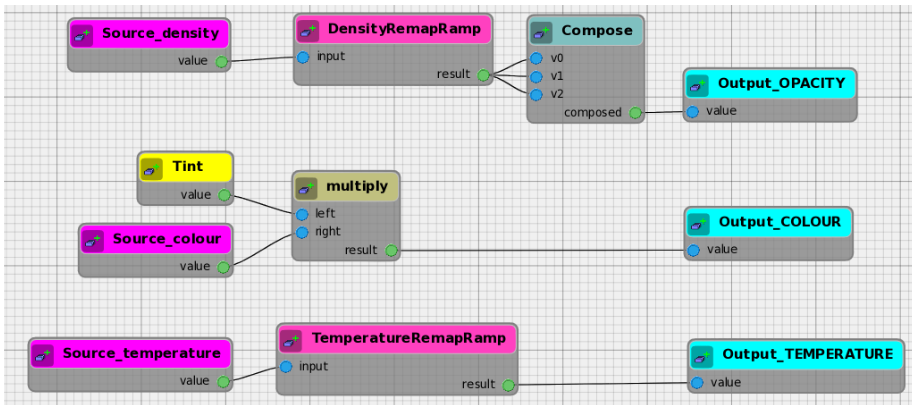


Figure 4: A very simple fluid shader to map the `density`, `colour` and `temperature` attributes from a fluid file to the voxel shader input attributes `OPACITY`, `COLOUR` and `TEMPERATURE` respectively. Fluids normally have a scalar `density` channel, so this is used to generate a greyscale `OPACITY`. The `colour` attribute is tinted to provide `COLOUR`, and `temperature` is remapped to give `TEMPERATURE`.

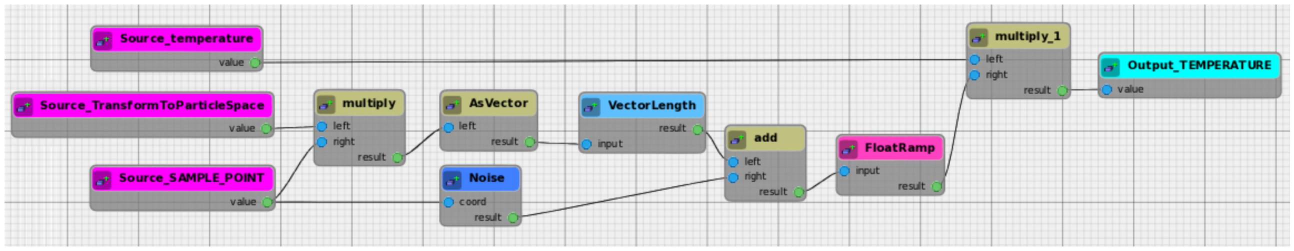


Figure 5: A representative example of a simple particle shader used to modify temperature in the vicinity of each particle. The particle is evaluated for every voxel within its sphere of influence. At each evaluation, the camera-space point `SAMPLE_POINT` is transformed to a normalised space centred on the particle's position (by multiplication with the matrix `TransformToParticleSpace`). The distance of the sample from that vector space's origin is computed, jittered, and used to drive a ramp to provide a multiplier for the particle's `temperature` attribute.

The data that has been generated for each primitive must be appropriately combined with that from the other primitives. This is often achieved using opacity weighting, but this process can be modified by shaders if another method is more appropriate.

3.3.2 Voxel Pre-Shade

Once all the data has been accumulated, the `VoxelShader::preShade()` method is called, which allows an indicative approximation of the output values to be computed.

There are two reasons for always invoking this stage. The first and most obviously apparent is that it allows interactive output drivers to display an intermediate image as feedback to the user.

The second reason is less clear: it allows the renderer and output drivers a chance to preview the data they're going to process later, allowing various optimisations to be made. Typically only one or two cheap output channels are computed at this point.

3.4 Shade Stage

Once the fill-stage has been completed for any tile it can progress on to the shade-stage. This is the stage at which the final output buffer values are computed for each interesting voxel. As with the fill stage, each tile is processed slice by slice, with all output drivers being given access to the shaded voxels. If a slice was deemed uninteresting in the previous stage it won't be shaded, and the output drivers won't have a chance to output any voxels from it.

The implementation of `VoxelShader::shade()` is normally similar to that of `VoxelShader::preShade()`, but also processes the attributes which weren't required in the earlier stage. The additional expense of providing the colours may not be large, for example if just combining and mapping input attributes (as is the case for our `Ci_temperature` and `TEMPERATURE` outputs). On the other hand, there may be some significant expense, e.g. sampling point clouds and environment maps, or evaluating light-loops and deep-shadowing (as we do for the `Ci` output). The expense here is primarily in disk IO and pre-processing the data from these external inputs. Although the costs are small in terms of an entire render, it is helpful to delay the loading of such resources until this stage in order that the earlier preview stages may be completed as quickly as possible.

3.5 Motion Blur

The end of the shade stage (§3.4) marks the completion of an un-blurred render: each output driver has been given access to every fully shaded voxel in which it expressed an interest. This is all that is needed to provide a final output image, point-cloud, deep-texture, etc.

Optionally, a final motion-blur pass can be executed after all of the tiles' shade-stage tasks are completed. The motion blur engine uses the voxels' velocities and the motion of the camera to smear opacity and other attributes in space (see §4.4).

4 Implementation

4.1 Output Drivers

As discussed previously, data provided by the voxel shader is passed to each output driver, one slice at a time. Each driver can interpret and process the data differently in order to generate its final output.

DNB currently has four output drivers in production use. Two of these (the Open-EXR and Blit drivers) are very similar, outputting RGBA images. The deep-texture driver is currently only used for rendering deep shadowmaps for lighting and holdouts, although it is by no means limited to this task. Finally, the point-cloud driver is used to bake out a point per voxel, which can later be used in other renders (e.g. in PRMan).

4.1.1 Open-EXR & Blit: 2D Image Drivers

DNB's 2D image drivers take exactly the same data and process them identically, but simply output to different destinations. These drivers can only render RGB colour channels accompanied by an RGB opacity channel. For each ray in x & y the driver accumulates a single pixel value by stepping through and accumulating each of the ray's voxels in z. For the opacity O, this process is:

$$O_{\text{ray},n} = O_{\text{ray},n-1} + O_{\text{voxel},n}(1 - O_{\text{ray},n-1})$$

For the sake of simplicity, we define

$$\begin{aligned} O_{\text{additional},n} &= O_{\text{voxel},n}(1 - O_{\text{ray},n-1}) \\ \Rightarrow \qquad O_{\text{ray},n} &= O_{\text{ray},n-1} + O_{\text{additional},n} \end{aligned}$$

Colour is accumulated similarly, but is weighted by $O_{\text{additional}}$:

$$C_{\text{ray},n} = C_{\text{ray},n-1} + C_{\text{voxel},n} O_{\text{additional},n}$$

Any number of channels can be concurrently accumulated by the EXR driver, allowing AOVs to be output in EXR format. These AOVs (or *secondaries*) are used by compositors to fine tune the composited images without having to re-render the DNB pass for every tweak, which is a great saving in rendering resources.

A key part of the ray march accumulation is *opacity normalisation*, in which the opacity value of each voxel is modified in order to maintain a consistent accumulated opacity, regardless of the spatial sampling rate in z. This is important to artists' workflow, as much development work is carried out at low resolutions, switching to high resolutions only in the later stages of production.

Partial holdouts are also incorporated into the ray march in these drivers. This is a key component required for compositing DNB renders with those from other sources, e.g. PRMan. A holdout texture is sampled directly by the driver, and used to reduce the opacity of voxels according to the sampled holdout transparency. This holdout transparency component ($T_{\text{holdout},n}$) is incorporated into the ray march by modifying the additional-opacity calculation:

$$O_{\text{additional},n} = O_{\text{voxel},n}(T_{\text{holdout},n} - O_{\text{ray},n-1})$$

4.1.2 Deep Texture Driver

The deep texture driver outputs deep-pixel data for the rendered frustum. It can potentially output any data, with any accumulation function, although the current implementation simply outputs $O_{\text{ray},n}$ as a function of n (see §4.1.1). This is the function required for deep-shadowing and partial holdouts.

4.1.3 Point Cloud Driver

It can be useful to output the 3D DNB render as a point cloud, for use as an input to another component in the pipeline. The driver currently outputs the ray-marched $O_{\text{ray},n}$ for each voxel, alongside other attributes such as isosurface normal, temperature, voxel dimensions, etc.

Point cloud files can easily become extremely large, especially considering the vast memory throughput of a very normal production render. For this reason, the resolution of these renders is normally kept to a minimum, and the driver also culls points which have no opacity in order to minimise the amount of resources wasted by redundant data.

4.2 Element Shaders

The task of the element shader has changed subtly in recent DNB versions. Historically, the shader plugin was responsible for modifying raw element data, and providing on demand samples given a point's coordinate. This worked, but it some cases it became a limitation, as the assumptions of the renderer's stock primitives didn't always tally particularly well with the intentions of the shader. For this reason, the shaders are now delegated the task of actually populating the voxel structure, tile by tile. This has allowed some great improvements in efficiency, particularly in particle shading (see §4.2.2 below).

4.2.1 Fluid Shaders

As already discussed briefly, one of DNB's most widely used inputs are Zen fluid caches. Zen fluids are compressed in *cells*, or spatially coherent blocks of voxels; typically these cells are cubes of dimension $32 \times 32 \times 32$.

Fluid shaders for this type of render primitive are initially responsible for defining the mapping of attributes required by the voxel shader to the attributes which must be read from the file. Before the renderer begins accumulating data for each tile, it first notifies each primitive that it should load any new data required by the current tile. At this point the Zen fluid primitives load the required raw data cells from file and pass them on to the fluid shader for "pre-shading". This process is normally a combination of attributes on a per-voxel basis (i.e. without reference to surrounding voxels).

After all the primitives have completed their pre-shade tasks for a given tile, the renderer moves on to filling the tile's attribute buffers with data to pass to the voxel shader. The attribute buffers to fill are passed to each successive primitive, which blends its contribution with previously existing data. In the most widely used shaders a fluid's contribution is sampled from the cellular data which was pre-shaded at the beginning of the process.

Typically the sampling is performed using hermite-spline interpolation, to give smoothest results. Local gradients are stored in order to make the interpolation quicker to calculate. This also has the benefit that the gradients of a scalar density channel also encapsulate the isosurface normals of the fluid (often required by the voxel shader), which are therefore available for free.

An example of the standard shader implementation is discussed in section §5.3, which outlines a recently developed nodal shader interface. The node graphs represent the operations to be executed on each voxel at the pre-shade stage. The results of this stage are sampled using the default hermite-spline interpolation.

Of course, fluid shaders are not *required* to populate the voxels' data in the manner outlined above (this is a key reason for the responsibility changes mentioned earlier). One commonly used shader, for instance, constructs an internal brick-map from the Zen fluid cache which is used to drive a colour bleeding solution.

4.2.2 Particle Shaders

For various reasons, the particle shader implementation is rather more complex than that of fluid shaders. Not least, particles are points with radii, their extents in space varying depending on any number of user defined or simulated attributes. This is not a problem with fluids, whose regular voxels' extents are easily calculable.

The first task of the particle shader comes before the camera is focused or any tiles have been created: it must compute the size and shape of each particle in order to provide a bounding box to the renderer. Every particle must be processed at this stage so it is also used as an opportunity to bucket particles into the tiles which they affect. The particle set's equivalent of the fluid's cell loading stage is reloading and pre-shading the particles which touch the current tile. Any per-particle attribute adjustments (e.g. bringing into range, adding noise, etc.) is performed at this stage.

The buffer filling stage of a particle shader has an identical interface to that of the fluid shaders, so no particular shading method is enforced. Over time however, one particular strategy has been accepted as the most efficient, and is shared between all of DNeg's shader variants. In this strategy, the attribute sampling task is split into two nested loops: per-particle and per-voxel.

In the per-particle loop, any transient particle constants are computed, and then that particle is sampled into each voxel. The constants are computed during the outer (per-particle) loop at this stage rather than at pre-shade as they are only needed for a very short time. Also, they are often relatively large (e.g. matrices, which are invariably required) and it makes little sense to store these for every particle.

The inner loop iterates over each voxel the particle interacts with. It is at this point that fluid caches or procedural noise buffers are sampled, radial falloff samples are computed, depth weighting applied, and so on.

Design of particle shaders is a rather involved process which cannot entirely be removed from any implementation, including the recent nodal interfaces. Even in this case the inner and outer loops are exposed to the shader designer, who must consider the efficiency of their computations. This is a cost of the freedom extended to TDs to develop their own shaders, but one which is largely circumvented by hiding much of this implementation detail from the final user's UI.

4.3 Voxel Shaders

Ironically, given their central role in a render, basic implementation of a voxel shader is extremely simple, to the extent that the C++ programming is the major stumbling block for most TDs. Bypassing this prerequisite was indeed one of the major driving forces for the development of the nodal front end to each of the shader types.

The voxel shader is simply expected to iterate over each voxel in a slice, translating its input attribute values into output buffer values. In many cases this only involves the application of a colour ramp to various scalar inputs and writing the result to the output buffers.

4.4 Motion Blur

DNB's motion blur engine is effectively a multi-segment 3D anti-aliased line rasterizer, based on a modified implementation of Xiaolin Wu's line algorithm[5]. The path of each voxel's motion during the shutter duration is split into multiple linear segments in the voxel structure's coordinate system, one line between each pair of motion samples. These lines are then rasterized into the voxel structure, maintaining an opacity weighted accumulator for each output channel in every voxel.

[5] WU, X. 1991. An efficient antialiasing technique. In *SIGGRAPH '91*, 143-152.

The path for each voxel is evaluated from a combination of three sources:

- **Camera motion**: motion due to the animation of the camera's world transform.
- **Rigid-body motion**: motion due to the animation of each primitive's world transform.
- **Internal velocity**: velocity sampled (or dynamically generated in-shader) from each primitive's data.

The effect of camera motion can be evaluated dynamically for a voxel given only its centre point and the camera's world transforms. Internal velocities and the rigid-body motion components cannot be evaluated in the same way, so they are accumulated during the fill stage, resulting in a single combined motion vector for each voxel: we call this the *compound velocity*.

For any given voxel and motion segment, the line through the voxel's centre point and along its compound velocity vector is calculated, initially in world-space (figure 6a). Each end of this line is then transformed into camera-space, taking into account the different camera matrices at the beginning and end of the motion segment (figure 6b). The line joining these two points is then used as the path along which to smear the voxel's data, as demonstrated in figure 7.

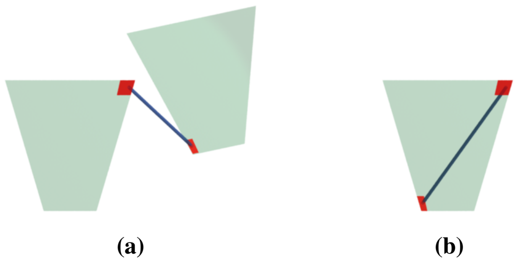


Figure 6: Computation of blur vector start and end points. (a) The camera-space coordinate at the beginning and end of the blur vector is computed, taking into account the movement of the camera's frustum. (b) The line joining these coordinates is rasterized into the frustum of the final image so as to smear the data along the path.

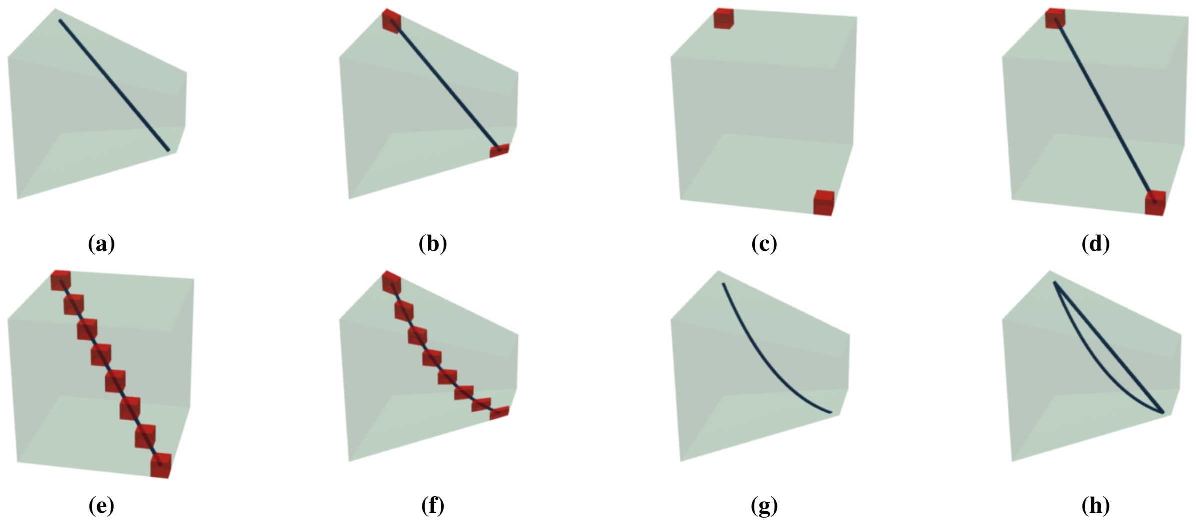


Figure 7: The rasterization of the complete motion blur path is performed in the orthogonal voxel-coordinate space, in order to reduce computation cost. The motion's path (a) is computed in camera-space, but the end points (b) are transformed to the orthogonal space (c). The rasterization is then performed linearly in this space (d), affecting all voxels along this modified path (e). These voxels, although "linear" in terms of pixels in the final image, do not necessarily follow a linear path in world-space (f) & (g). Although not noticeable in production renders, this does result in a difference between the desired and actually rasterized motion paths (h).

The smeared data is accumulated in a separate 3D buffer for each tile, which is then sent slice by slice to each output driver, as with all the other stages. Limits on physical memory mean that only a subset of tiles can be assigned accumulation buffers at any one time, so each unblurred tile may have to be evaluated more than once if its effects are far reaching – see figure 8.

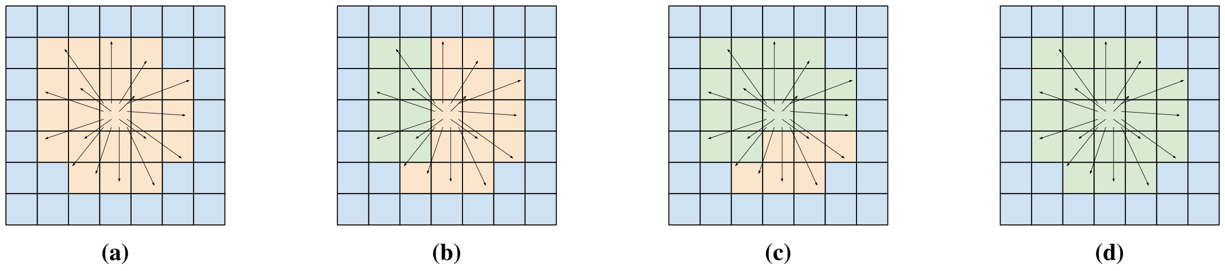


Figure 8: In many cases there is insufficient memory available to simultaneously process all tiles affected by a given unblurred tile. In this situation, the original tile must be processed a number of times, blurring into a subset of its targets at each iteration.

5 Shader Discussion

5.1 Shader Philosophy

Having covered the mechanisms by which the renderer works, it is now appropriate to discuss more specific examples of shader use and the philosophy behind some of the design decisions.

DNB has always been a very open platform which encourages TDs to get involved, empowering them to create setups best suited to their particular needs. The ultimate aim is to make it as easy as possible to get the best looking beauty image out of the renderer, but also the flexibility to feed whatever they're asked for to their compositor, so tweaks to the look can happen without a re-render.

We also encourage new thinking and artists know the door is always open to put hooks in place that would facilitate a better render pipeline. Many shaders have been written during the renderer's life with these goals in mind; it is our intention to set out the main features of some of the more useful ones as it is believed that this aspect of the renderer is just as important as some of the more intricate technical aspects of the renderer internals.

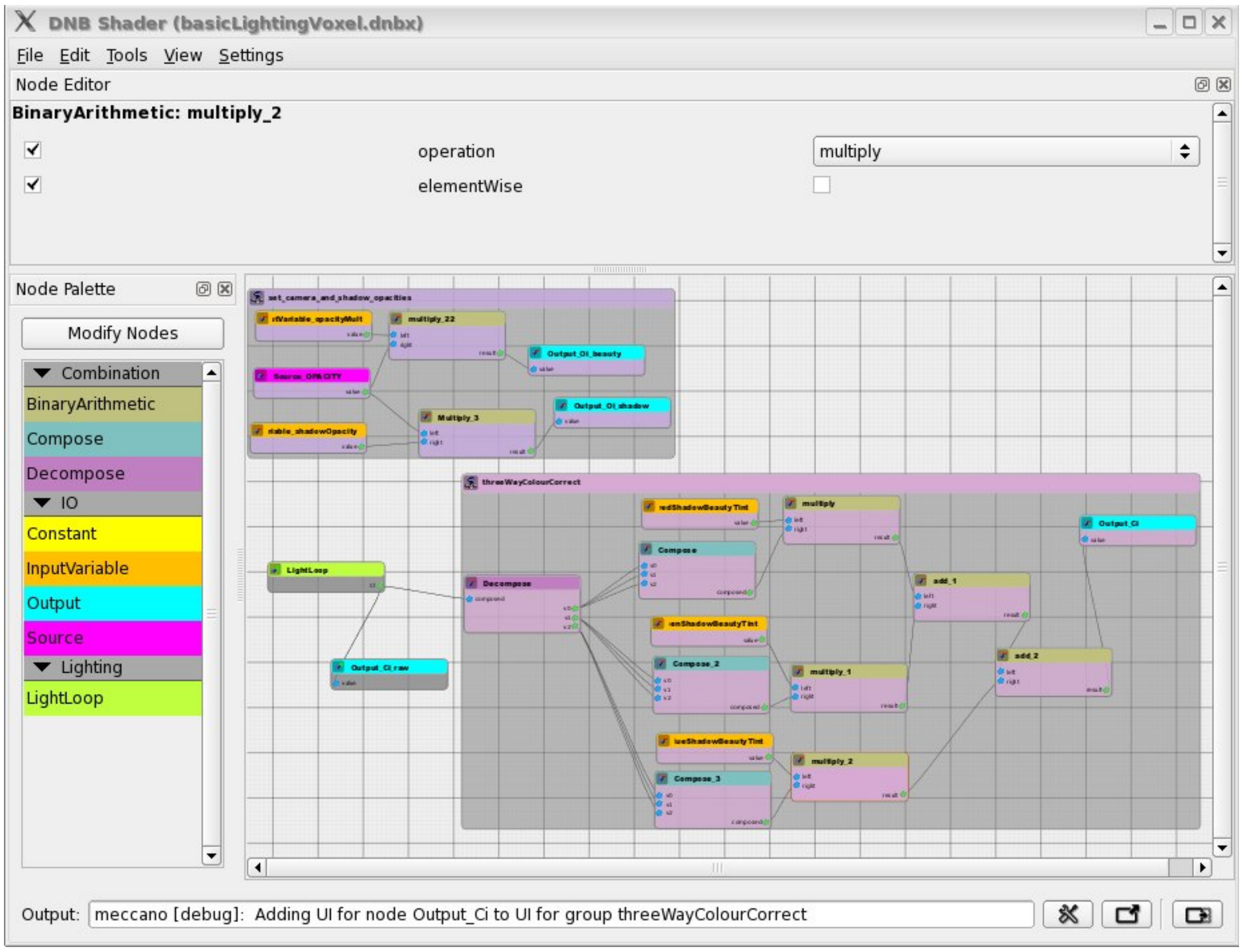
5.2 Hard-Coded Shaders

Traditionally, shaders in DNB were written in C++ and derived from a base class that provided many base class functions and parameters. They were then compiled into dynamic libraries that were loaded and executed by DNB during a render. Pre-2010, a small group of more technical TDs developed these shaders for DNB, and trained those who showed an interest to make their own additions to the core shaders. The perceived needs gradually evolved, then eventually stabilised somewhat. Since roughly 2008, there have been no significant changes to the most popular fluid and voxel shaders. With the renderer backend restructuring, it was decided that these core shaders could go into the stewardship of R&D to practice the new DNB philosophy of a simple interface to complex architecture - regardless of the guts of the renderer changing, the shader interfaces have remained constant and delivered the expected images. These core shaders remain popular for day to day use, providing good flexibility for a variety of common tasks, but are restrictive for trying new things.

5.3 Node Based Shader Interface

Recent internal developments on DNB, while they have greatly increased the functionality and power of the renderer, have also introduced new challenges to those attempting to write shaders for it. In short, the barrier to coding shaders for DNB is now far higher, taking it even further out of the hands of non-technical TDs who could have been somewhat intimidated to start with. In order to free up the TDs again, a node based shader architecture was implemented. Each node-type is a precompiled object defined in an additional plugin, and shader descriptions are now abstracted into xml files that describe node dependencies and parameter values. The shader itself is no different to any other shader, but instead of processing the data itself it delegates the task to the plugins for each node. All it has to deal with is loading and creating the nodes, and executing them in the correct order. A user friendly front end has been created so that the user experience is akin to some other node based UIs in our industry, and TDs are once again free to create images through whatever methods they can come up with.

The usual working practice is for a sequence lead to design the shader graph, set appropriate defaults, and publish this out to the show. Any subsequent improvements to the graph can be pushed out using DNeg's publishing mechanism, with users free to pick up improvements as their shot needs dictate. If a user needs to alter the graph itself, they can check out a copy of the shader, alter it and branch off to a custom shot/sequence publish so as not to damage the rest of the pipeline.



BinaryArithmetic: multiply_2

✓ operation multiply
✓ elementWise ☐

Output: meccano [debug]: Adding UI for node Output_Ci to UI for group threeWayColourCorrect

Figure 9: Our custom PyQt interface, Meccano, for designing shader graphs. Features such as only displaying active nodes in the Palette, Grouping nodes used to achieve a particular task, and the ability to view the current graph from within Maya all help the user experience.

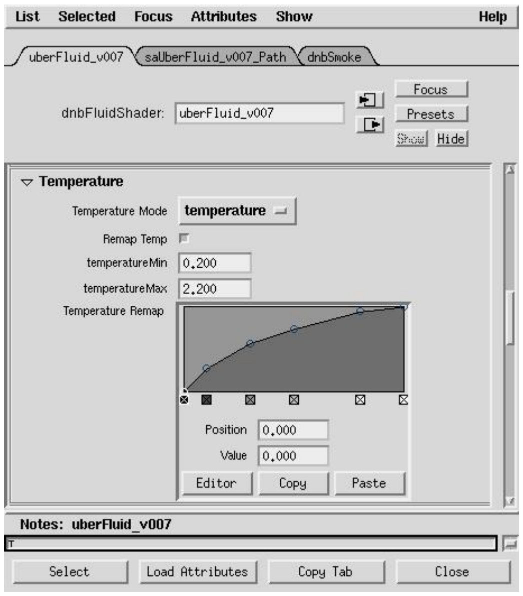
5.4 Maya Parameters and UI

Shader parameters for hard coded shaders are listed in an associated xml file for each shader written:

```
<dnbFluidShader>
<category name="general" label="General" collapsed="false">
    <parameter>
        <name>useInBeauty</name>
        <label>useInBeauty</label>
        <keyable>true</keyable>
        <type>bool</type>
        <value>1</value>
        <annotation>
            Switch off to disable fluid's evaluation in camera pass (you could
            keep it on in shadow so it casts shadows but isn't visible!)
        </annotation>
    </parameter>
    <!-- more parameters... -->
</category>
<!-- more categories... -->
</dnbFluidShader>
```

The DNB Maya plugin can read and display these shader parameters, automatically creating the correct widgets. A shader node within maya keeps track of the user's shader parameter choices and these get written to the DNB script file for the shader to use at render launch.

What the user sees in a custom AE template in Maya will be something like this:



dnbFluidShader: uberFluid_v007

Temperature Mode: temperature

temperatureMin: 0.200

temperatureMax: 2.200

Position: 0.000

Value: 0.000

Notes: uberFluid_v007

Figure 10: A fluid shader snapshot with temperature remapping options.

In the case of node-based shaders, shader designers only expose the input parameters they want users to tweak, allowing more control over how much a user can diverge from an approved look. The Maya integration is still incomplete - only simple data types can be linked from within Maya - as yet nodes driven by ramps and complex sets of parameters must be manually tweaked inside the shader graph, followed by a re-publish of the shader instance.

5.5 Comparison of Old and New Style Shaders

While the particulars of DNB are mostly just of interest to those working at DNeg, anyone designing a volume rendering system should probably think about what they will want to expose to their users, and where best to plan hooks into the renderer for users.

Hard coded precompiled R&D controlled shaders have the following benefits:

- Stability - once a shader has evolved over many productions, people know what to do with it, and the lack of "moving parts" means problems are unlikely to crop up on new shows.
- Efficiency - knowing exactly what the shader needs to do makes it possible to optimise performance as brutally as possible.
- Familiarity - with a known set of params, training becomes easier, simulations can standardise their expected attribute outputs, naming conventions can be put in place for passing elements across to compositing.

But there are also drawbacks to relying on this approach:

- Redundancies - With shaders that are designed to be one size fits all, it can end up that defaults need to be provided to account for attributes that won't always be there, or UIs with many params that don't apply to the majority of rendering cases

- Compiling Workflow - the barrier to entry for TDs is higher, they have to think about code maintenance, compilers, libraries,
- Predefined Attributes - tied to renderer version
- Pass limitations - if different shaders are required to do different things, treat attributes differently etc, multiple passes may be needed to achieve a greater variety in the outputs

Looking at our newer Node Based Shaders, there are different benefits:

- Flexible - On a given shot it can be easier for users to add their own AOVs, to change the intended behaviour with a small graph tweak, to increase complexity by merging multiple graphs
- Open To All - the workflows known to many FX TDs through tools like Softimage's Ice, Houdini's VOPs etc mean people can sit in front of the UI and instantly start plugging things in
- No Limits - there are more Creative/Artistic possibilities and possibilities for happy accidents available when users have the freedom to hack away at a graph, there are less restrictions on what "makes sense" than what would normally be engineered in a shader written in code.

But, again, there can be negatives involved:

- Thinking - shifting the onus to more users working with graphs flexibly means knowledge of functionality and individual nodes is needed, the underlying architecture is exposed
- Shader Designer Workflow - it can become less straightforward to roll out updates if people have branched off to their own graphs locally, and we currently have UI issues related to using a standalone GUI for shader creation but off-the-shelf software for the interface to the render
- Error Prone - with more moving parts, it can be harder to track down where errors have crept into scenes, it's a bit harder to get a straight printout of the line of code
- Nothing Comes Free - if you need more than default functionality, you have to wire up or create the attributes manually for every render element

6 General Shader Examples

6.1 Fluid Shaders

As previously discussed fluid shaders act directly on the data as provided by the input files. This allows users to change how different fluids appear on an individual basis. For example, the same data file could be used with different fluid shaders and this could cause many different looking fluids to be rendered in the same image.

Fluid shaders can also been used to create additional attribute data at render time. They can even procedurally create all data required thereby removing the need for a data file.

6.1.1 Basic Density Generation from Levelset

The example graph in figure 11 shows one of the most useful features of fluid shaders - describing the opaqueness of a fluid - i.e. the user is basically saying a certain density value or point in an object is equivalent to an opacity value of 1.0. Parameters get exposed for remapping this density differently based on whether we're in a shadow or beauty pass.

This can be utilized to bring differently simulated fluids into the same range so they all sit together, or to emphasise particular differences between sims, like for temperature remapping, to bring up an area of hotness in the core of a large explosion. We also provide the ability to remap attributes for example using temperature from a fluid sim as the density of a render. With the latest versions of DNB, the user is free to create as many arbitrary attributes as they need to pass across to the voxel shader, so long as the voxel shader is set up to expect compatible inputs.

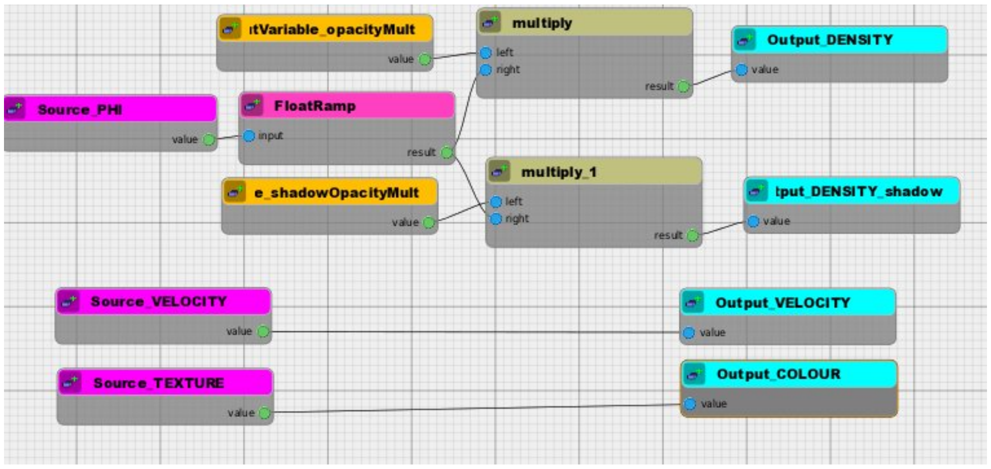


Figure 11: A levelset fluid shader, described by a Meccano graph.

6.2 Voxel Shader Practical Examples

Voxel shaders are at the heart of the DNB rendering process and this is where all the hard work gets done. It is also the place where major trickery is best to occur - input data has been munged into a high res voxel structure and wants to be made pretty. The dnbVoxelShader base class provides a lot of functionality and a very simple shader can be written in just a few lines - or with just a few nodes. Examples of typical things follow. . .

6.2.1 Beauty vs. AOVs

A very common technique used in DNB renders is to multiply the opacity channels down by different amounts in the shadow pass. So for example the red channel will have high opacity, the green channel might have similar opacity multiplication as the beauty pass, and the blue channel might have a multiplier of some very small amount. By doing this, the shadowing appears stronger in the first channels when viewing the deep shadow map.

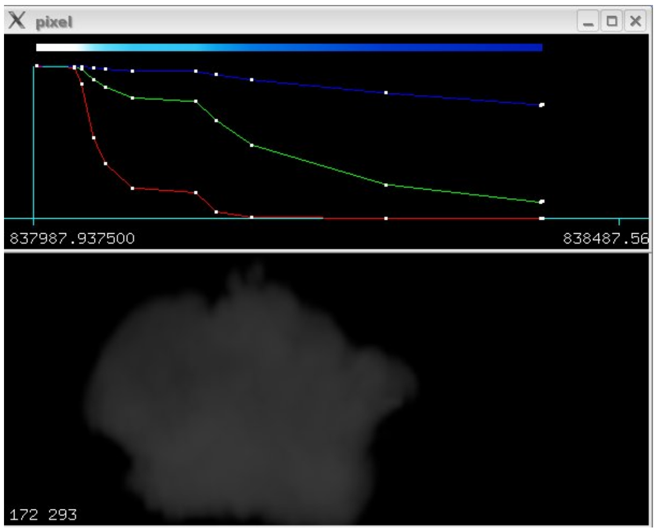


Figure 12: dsview (PRMan tool) screengrab of typical RGB opacity curve for a pixel in a deep shadow map

So when these shadowmaps are used in the beauty, for a single light it would appear that the red channel has very harsh rim lighting effects, the green channel has the expected "normal" extinction, and the blue channel allows a lot of light to penetrate the density medium before reaching a fully shadowed state. This is very useful for compositors, as they can then instantly choose between various lighting looks without messing about. It's less useful for TDs though, who are now unable to judge what their beauty looks like. To combat this, we can give controls that let the lighting TD dial in some of what the comper would be doing - but in the render - and feeding that into a beauty output. The raw useful shadowing is still getting dumped, so everybody wins. Figure 13 shows a simple implementation of that concept in a sample node graph.

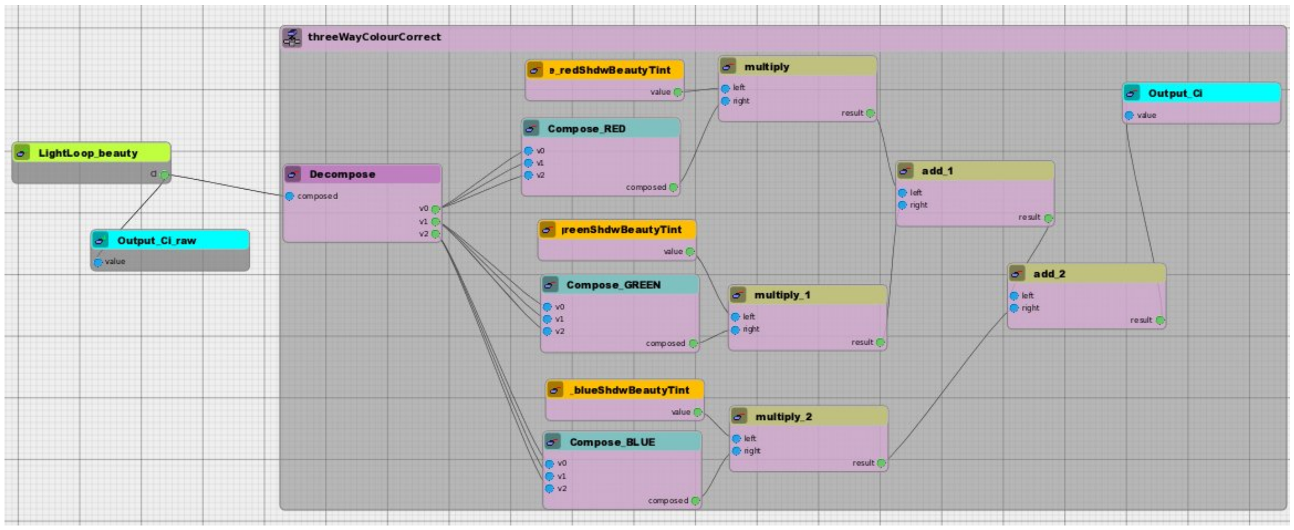


Figure 13: Operator node graph showing simple colour correction operations used to produce a desirable beauty colour from shadowmaps which have been tinted to provide different amounts of shadow in each channel.

6.2.2 Standard Lighting Effects

The IsosurfaceVoxelShader takes advantage of the base class's facility to generate isosurface normals from a density gradient. This normal can then be used to look up environment texture maps, giving surface lighting effects normally associated with proper surfaces.

Using a random lookup of 100 nearby voxels produces a blurred look, while doing a regular grid lookup with the nearest surrounding voxels used to generate a meaned isosurface normal gives the image a less blurred or dithered look. It is also possible to vary the normals arbitrarily, in this instance bump mapping the fluid!

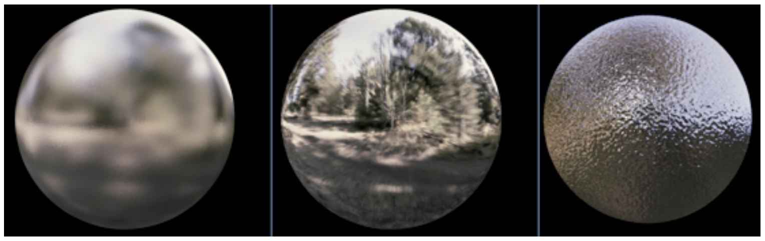


Figure 14: Comparison of lighting effects created using isosurface normal data to sample an environment texture map.

Most voxel shaders would also give controls to use these normals for doing diffuse or specular calculations, so all volume lighting doesn't just have to depend on shadowing effects.

6.2.3 Typical Secondary Output Ramp

For a utility output, it is best to try and provide the greatest possible range of information to the compositing stage, which usually means trying to normalise the input data, and map that data range into a RGB ramp which will then allow easy separation of low values through to high. In the example shown in figure 15, the top end of the density gets multiplied by an input parameter exposed in Maya, so that the density colour output can be tweaked to not max out if the input range is very high, or not sit entirely in the blue range of the colour ramp if a very wispy fluid is the input to the render.

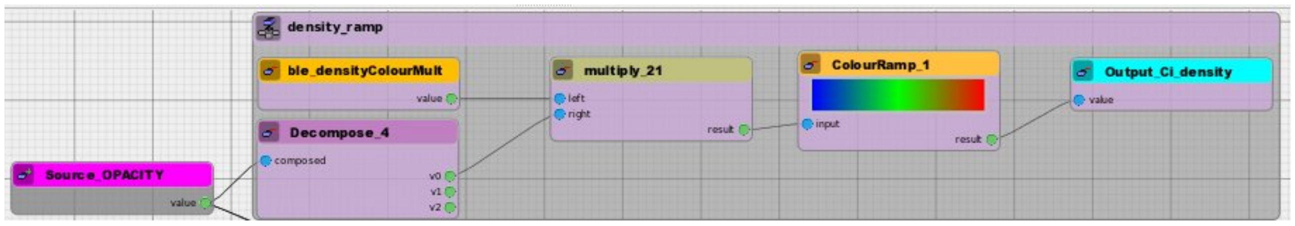


Figure 15: A typical secondary output ramp, used to map a scalar attribute to an RGB colour channel, tinted to provide maximum information for compositors.

6.2.4 Matte Objects

While DNB has support for reading in previously generated deepshadow files to use as holdouts, sometimes it is more convenient to assign a matte element shader, creating a special attribute that the voxel shader can use to knock out density in some areas of space. A voxel shader using this technique is shown in figure 16.

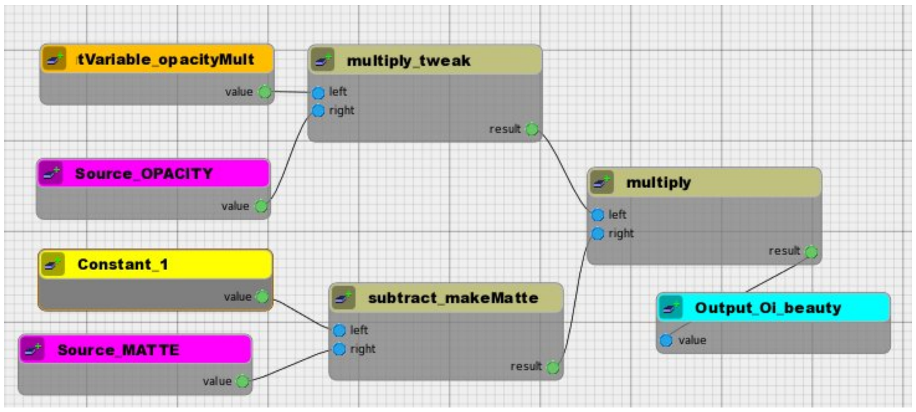


Figure 16: Using a matte channel, generated previously by a primitive element, to modify the opacity output channel.

6.2.5 Adjusting Attributes With Camera Depth

Another simple technique, given the Z position in camera space, a distance ramp can be looked up to fade up or down in order to blend things out close to camera to help disguise artifacts during fly-throughs, or indeed as objects get further from camera it might be necessary to have them slowly disappear.

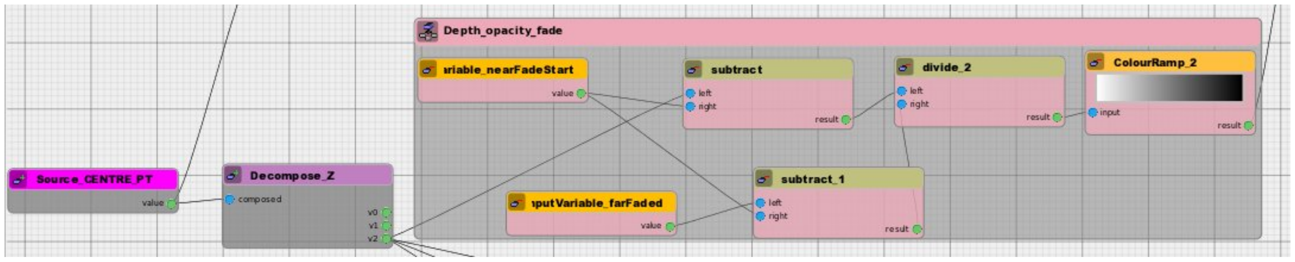


Figure 17: Modifying attributes based on the distance from the camera.

7 Shaders in Production

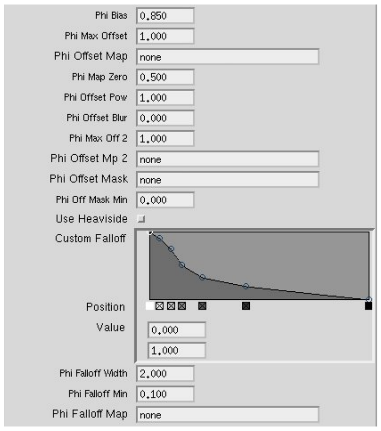
When shaders were precompiled black boxes, we found it beneficial to provide maximum levels of control over any parameters that could improve the look of the final image, or else to be able to send as many useful arbitrary outputs to the compositing stage.

7.1 Inkheart: Levelset Shader

On a project featuring a character made entirely out of smoke, it became necessary to create a soft core inside the sims. The original approach had a rigged solid guy on the inside emitting the smoke, but when he moved quickly, limbs could suddenly become visible and his geometric nature became apparent.

Using levelset code that had been added to Squirt, fluid grids were generated encapsulating the character's volume. In DNB a fluid shader was implemented to take the phi attribute per fluid and convert the levelset information to density at rendertime in the shader. In contrast to the earlier node based phi conversion graph, for this hard coded shader, extra parameters were added to:

- control softness of falloff
- implement displacement maps and mask textures for local control/detail
- add noise based on the texture coordinate on the surface closest to the current levelset position
- animate expanding noise so patterns on character's surface looked simulated not static



Phi Bias	0.850
Phi Max Offset	1.000
Phi Offset Map	none
Phi Map Zero	0.500
Phi Offset Pow	1.000
Phi Offset Blur	0.000
Phi Max Off 2	1.000
Phi Offset Mp 2	none
Phi Offset Mask	none
Phi Off Mask Min	0.000
Use Heaviside	☐
Custom Falloff	
Position	
Value	0.000
	1.000
Phi Falloff Width	2.000
Phi Falloff Min	0.100
Phi Falloff Map	none

Figure 18: Part of the levelset shader parameter UI.

Being a fluid shader made it possible to generate separate levelsets for hands, arms, body, head, and use custom texture maps for tweaking the density generation on each. By rigidly constraining the levelsets to the animation rig, resolution could be tailored for each body part based on shot requirements.

It was found to be a powerful additional tool in DNB's volume rendering arsenal, created by TDs without having to change the renderer's underlying code or architecture.

7.2 Fire Shaders Setup

Fire was an effect that first came up for DNB on Hellboy 2, then reappeared again and again on subsequent shows. A particular setup has evolved using our "uber" fluid and voxel shaders that seemed to give compositors flexibility to create various different looks for the fire without needing resims or even rerenders, so in order to make it as easy as possible to keep consistency in fire setups, specific fluid and voxel shaders were produced which utilized code from the earlier shaders but only exposed a subset of the controls.

Having the same codebase as the main multi-purpose shaders allowed for ease of code maintenance; having only a limited subset of parameters exposed made life easy for newbie TDs. It was additionally installed with good fire-looking presets per-show after initial lookdev, but always with the end target being the same secondary outputs based on the proven methodology (age, density and temperature mapped to an RGB ramp) that plugged into a standard comp template. All of the passes are compared in figure 19.

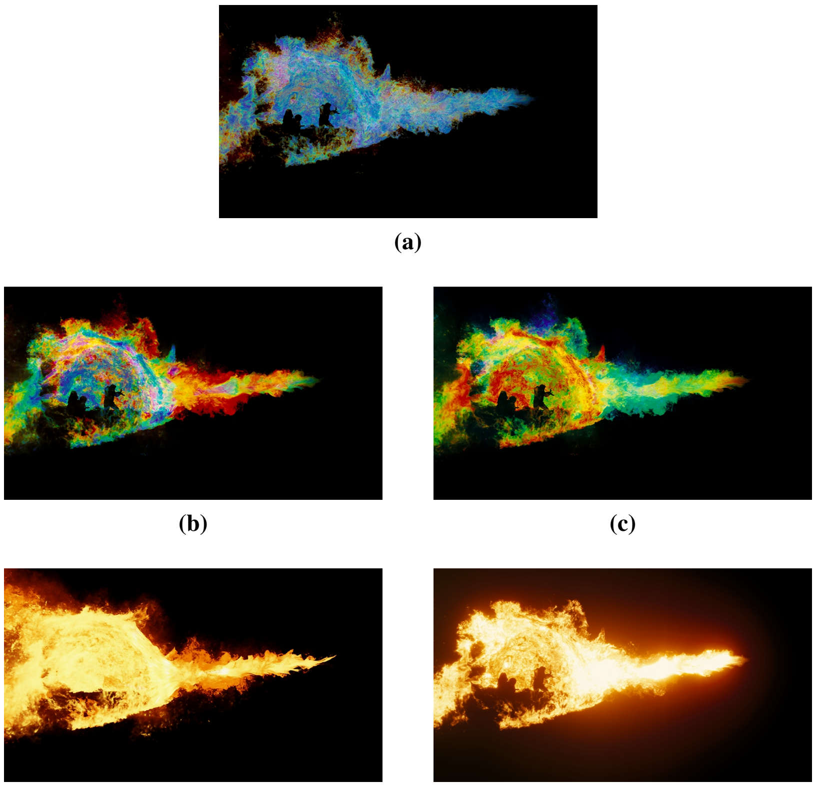
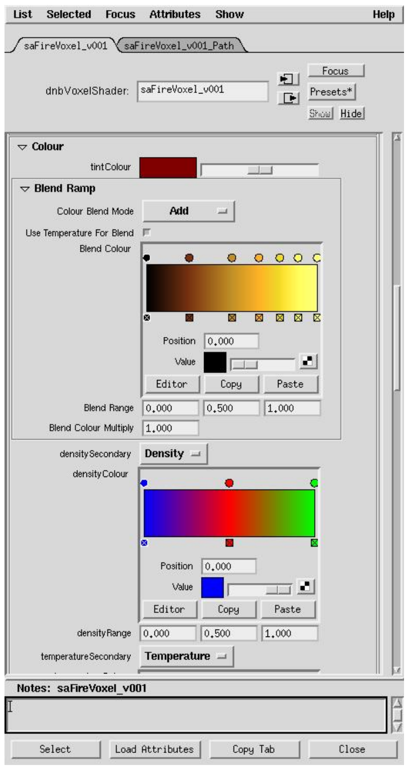


Figure 19: Fire passes: (a) density remap, (b) temperature remap, (c) age remap, (d) typical raw beauty colour output, and (e) composite produced using the remapped secondary outputs



dnbVoxelShader: saFireVoxel_v001

Colour

tintColour

Blend Ramp

Colour Blend Mode: Add

Use Temperature For Blend

Blend Colour

Position 0.000

Value

Blend Range 0.000 0.500 1.000

Blend Colour Multiply 1.000

densitySecondary: Density

densityColour

Position 0.000

Value

densityRange 0.000 0.500 1.000

temperatureSecondary: Temperature

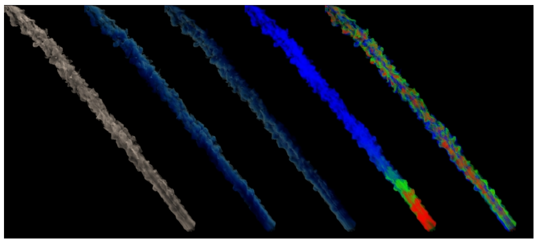
Notes: saFireVoxel_v001

Figure 20: Part of the fire shader parameter UI.

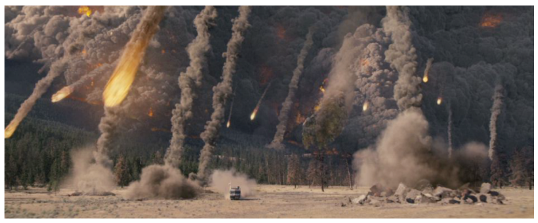
7.3 2012: Smoke Shaders

The standard practice for rendering smoke type effects is a little less well defined, as the desired look can vary greatly. As a result of this, a very general purpose shader is usually used, typically DNB's "uber" / swiss-army knife shaders. On 2012, there were four main types of smoke render - the massive ashclouds, airborne smoke trails coming from lavabombs, ground impact dust when lavabombs or earthbombs hit the Yellowstone environment, and masonry-type dust from collapsing structures in St. Peter's Square.

The trick of rendering shadowmaps with a ("orange") multiplier on the colour channels of the opacity was employed here. The beauty output used a colour-correcting set of multipliers so a good balance of the lighting could be found without having to recalculate shadowmaps. The untweaked light colours were written as secondary outputs, shown



(a)



(b) ©2009 Columbia Pictures Industries, Inc. All rights reserved.

Figure 21: (a) beauty, 2x lighting, age and noise AOVs for smoke trail (b) comped smoke trails as seen in shot

in figure 21a.

Standard secondary outputs like depth, density, age and temperature were then also output as colour ramps, and if texture coordinates had been simulated getting advected through the fluids, 3d noise could be written as a colour for 2d to use - or it could be remapped into a density ramp to tweak the opacity in certain areas to increase the amount of apparent detail in the sim at rendertime. For render efficiency, since this was pre-meccano shaders, checks had to be put in place to ensure particular attributes were needed - the voxel shader passed its requirements back to the fluid shaders via the API, to be picked up and used by the fluid shaders.

7.4 Captain America: Explosion Shaders

On top of the usual outputs generated for things like smoke and fire, for the Hydra Factory sequence on Captain America, it was necessary to give 2d options to play with timings for interactive lighting and surrounding flashes, so the render was set up to spit out various self shadowed lights as independent AOVs, along with a beauty which balanced them all nicely for initial comps. An isosurface normal output was also created, to help with enhancing the output detail so the explosions could hold up full screen.

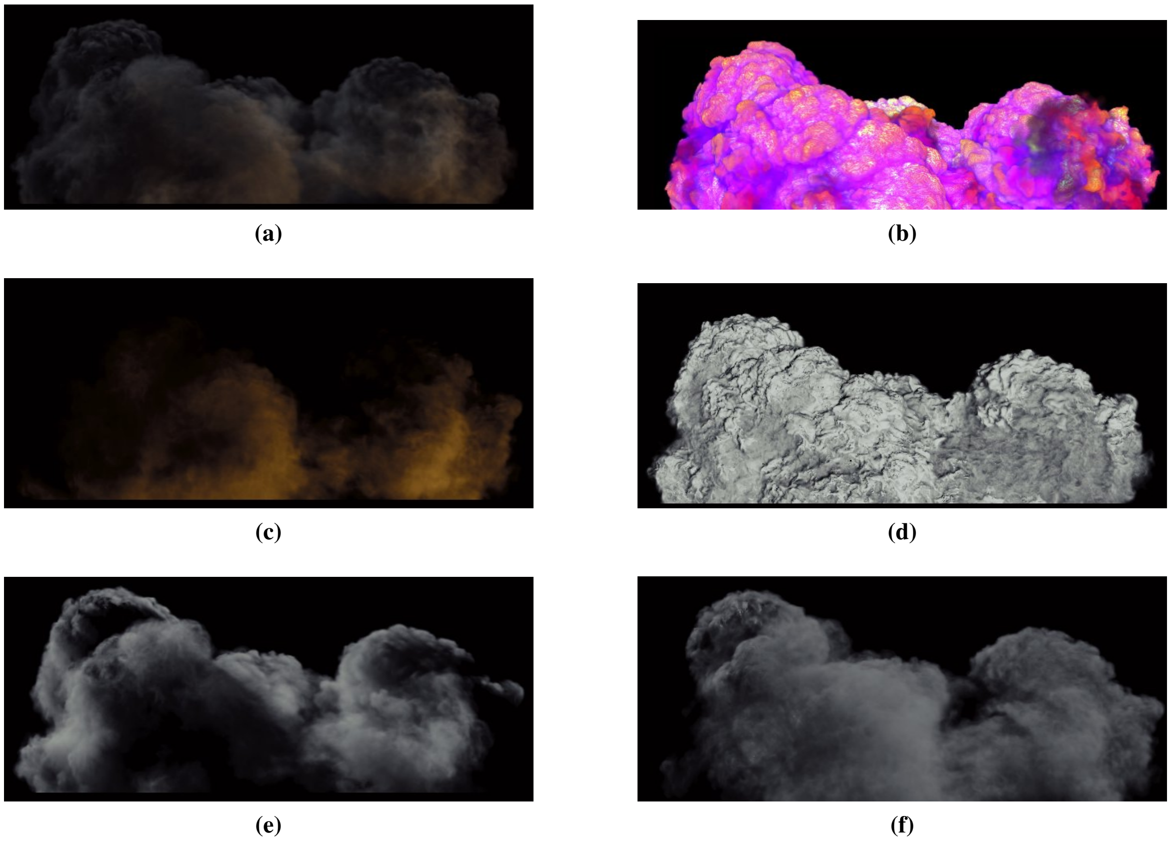


Figure 22: Explosion passes: (a) beauty, (b) temperature ramp, (c) environment underlighting, (d) isosurface output, (e) backlight pass (f) fill light

The resulting renders were a good test of how much detail the renderer was able to handle now compared to a project like 2012 – each sim was 150MegaVoxels, and the renderer managed to brute force a few of these at a time without falling over.

Figure 23: Captain America: The First Avenger ©2011 Marvel Entertainment. All rights reserved.

7.5 2012: Particle Shaders

7.5.1 Standard Particle Shading Practice in DNB

As is typical with many other volume renderers, and as shown earlier, when DNB wants to render particles it gives the user the simplest procedural options of sticking some sort of radially fading soft blob onto each point, or generating noise in the volume around the point. To get something which didn't look like the classic noise blob look it was decided on 2012 to investigate ways to place fluid sims on particles. It should be noted that at the time, we were under the impression that this might be a relatively new approach, and it was only on attending/presenting at the Siggraph Volumetrics 2010 course that it was realised exactly how standard this technique was for renderers that dated more from the time of particles being the primary unit of currency for volume rendering.

The typical DNB workflow of a TD writing a shader which did roughly what was desired was employed, and then the code was handed over to R&D to make it production worthy. In this case we wanted to instance many thousands of pyroclastic bursts to give the appearance of a massive wall of exploding ashcloud. The shots this technique would be used for were isolated from the bulk of our (very dynamic, not as expansive a scale) ashcloud shots but had to match in look, so we wanted to use our base ashcloud sims but in numbers impossible to implement using our normal fluid rendering pipeline.

7.5.2 Instancing Shader Implementation and Practical Limitations

Our workflow was to generate a surface to represent the wall of ashcloud, put particles on the surface, pass the surface normal as an attribute on the particles to DNB, and instance one of a selection of animated ashcloud sims on to each particle. An existing scattering shader used a single fluid sequence instanced on particles as a source for the scatter, so we took this code as a basis for our work, and fleshed out the parameters and attributes that our shader would need to give us the right behaviour - orientation, fluid id/multiple sequences support, growth controls with age, frame of the fluid sequence to pull in, etc. The shader worked by allocating density buffers for the storage of the fluid data, with R&D improving matters by implementing statically allocated density buffers which could be shared across particle systems, and deallocated on demand, to optimise memory use. It was found that although we could now allocate a number of buffers and hence have a reasonably large number of fluid variations, to get near-unique fluid files onto each particle (so that each particle could pick from a number of sequences, and use an arbitrary frame from that sequence to give lots of animation control) meant running into memory issues due to the number of required buffers. So variation was instead achieved by per-particle scale variations and orientation controls making it harder to spot repeating patterns. Additionally, all fluid clips had to animate at the same rate - so a limited number of animation starting positions in the sequence were permitted, and all particles evolved at the same rate. While this approach worked for our shots, it could be a severe limitation on many other setups.

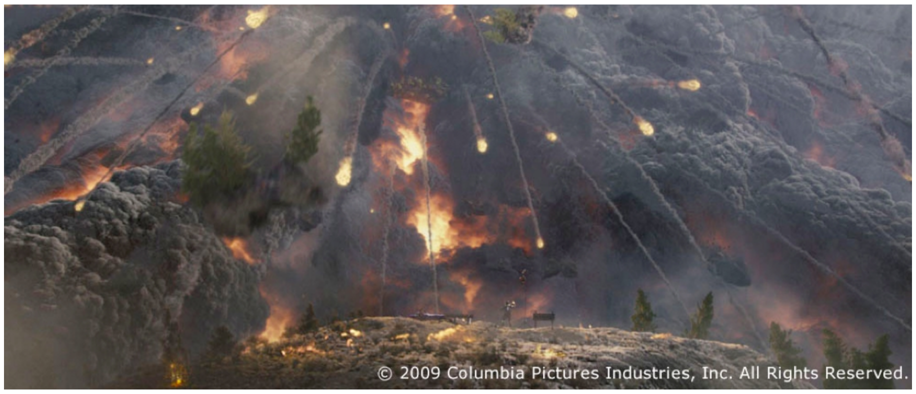

© 2009 Columbia Pictures Industries, Inc. All Rights Reserved.

Figure 24: Fluid Instancing on Particles in 2012 large scale shot

7.6 Captain America: Clouds with Node Based Shaders

7.6.1 Hybrid Approach

On 2012, for large scale ashcloud renders where we wanted individual control over placement, there was a setup that let us control placement of fluids, with particle sets placed sympathetically to light up the core. When we needed to do epic scale ashclouds, we switched to a method where fluids were instanced onto particles. When it came to do a hundred shots of CG clouds, knowing that we might need the flexibility to extend our environment to tens of kilometers, it seemed obvious that the epic option should be invoked. Cloud layouts were initially sourced from satellite photography, whereby we scattered points according to lumakeyed distributions. Onto each of these points we wanted to instance a great looking fluid (whether it came from a simulation or procedural volume modelling techniques wasn't too important to us) rather than relying on procedural rendertime techniques like noise buffers.

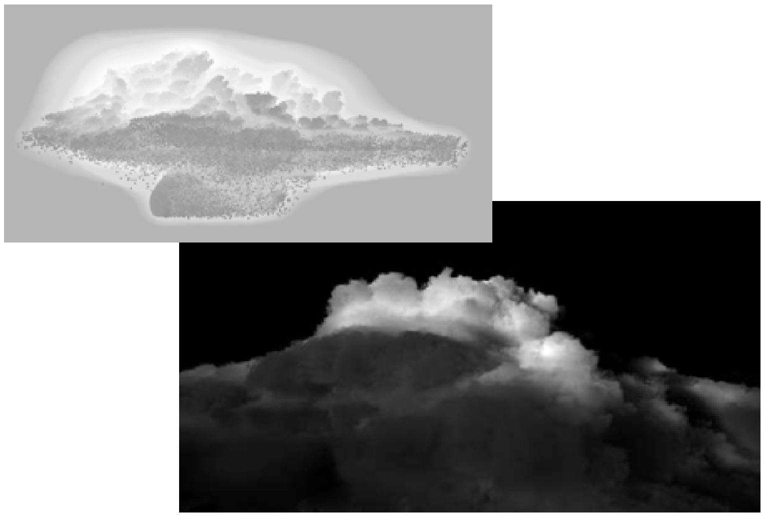
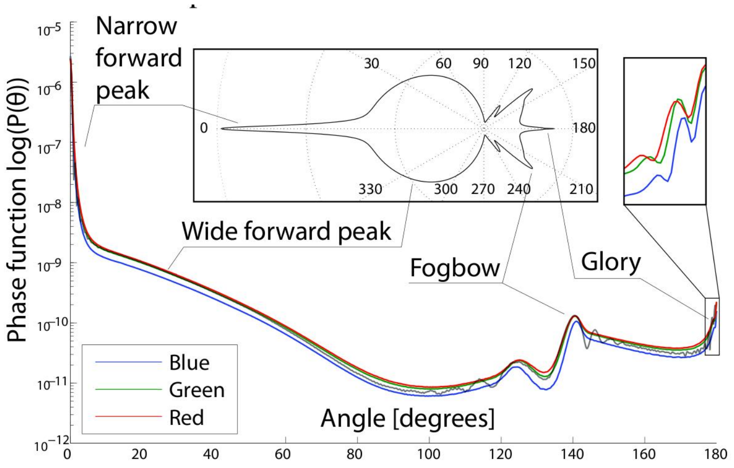


Figure 25: A cumulonimbus particle structure with blurred density; that fluid volume being used to fake multiple scattering effects on the fluids instanced onto the particles

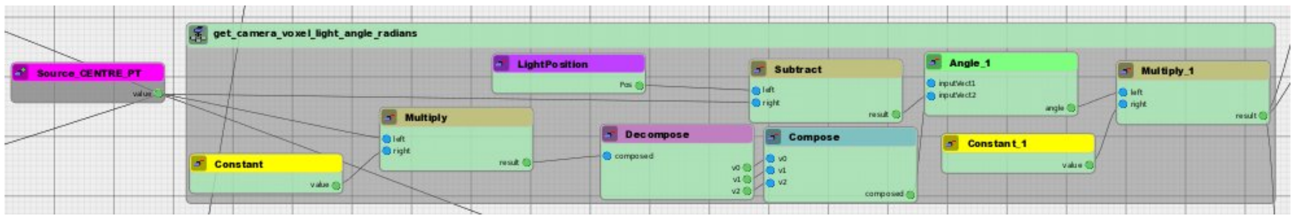
The cloud layouts were diced up into pleasing sections, which we could then import to layout scenes and shift around according to the storytelling needs of the shot/sequence. Early tests with this approach worked well, however we noticed that any attempts at rendertime multiple scattering calculations would not give great results given the volume of work to push through the renderfarm and the number of iterations we would like to be able to do per shot. So the particle layouts were brought into Houdini and used to generate relatively low res fluids, mostly by blurring the density of the particles and storing that in an attribute which would then affect the colour of the scattering output. These fluids were linked to the particles in the layouts so they remained synched using DNeg's publishing system.

7.6.2 Innovation Through Exploration

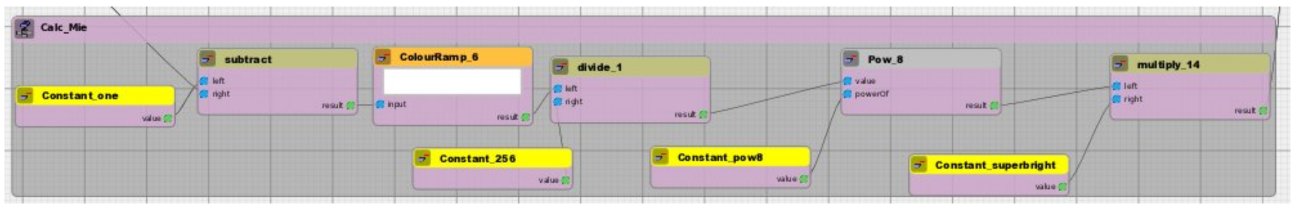
The FX Lead on the Clouds, Nicholas New, was keen to take advantage of the new possibilities opened up by the Meccano shader system. Looking at available research papers, the sky, photos, and interesting books about clouds, it was decided that realism would be greatly enhanced by incorporating more single scattering effects like Fogbows and Glories. Whereas in the past this may have meant some very hairy code would need to be written, instead it was possible to play around with the shader graphs and get something which approximated these effects, with user control over making it look right rather than being slavish to scientific accuracy. One significant paper[6] by the esteemed Antoine Bouthors (now at Weta, and presenting on this course!) had a particularly interesting graph which linked angle of incidence between sun, cloud and camera to some of these scattering effects:



(a) Graph of Mie phase function from Antoine Bouthors' PhD thesis.



(b) Current shading point and light position being used to calculate lookup angle for single scatter effects.



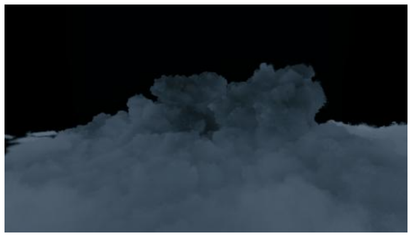
(c) Intuitive controls and adjustments to make scientific graph look good in assorted render cases.

Figure 26: Using the Mie phase function (a) to approximate single scattering effects. To replicate this in our system a new node was created to calculate the important angle (b), and a chain of nodes was derived to let us tinker with the lookup curve to get the right look (c).

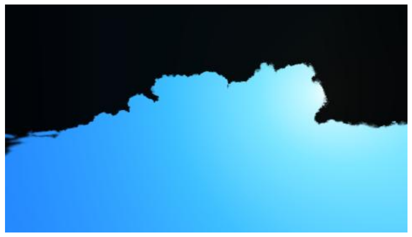
[6]"Realistic Rendering of Clouds in Realtime" PhD thesis http://evasion.imag.fr/~Antoine.Bouthors/research/phd/

7.6.3 Maximum Control

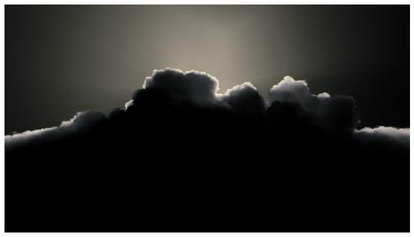
As is typical with many other effects previously discussed, as many useful passes that could be generated for compositing were created. This was particularly important because of the scale of the environment and subsequent heaviness of the rendering. It also became necessary to keep the look of the environment consistent across cuts, so that the sequences would hold together, and some lighting decisions could be made in comp. Here follows an example of some of the outputs created using methods outlined above.



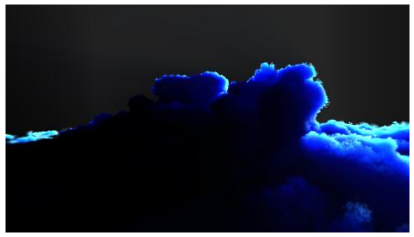
(a)



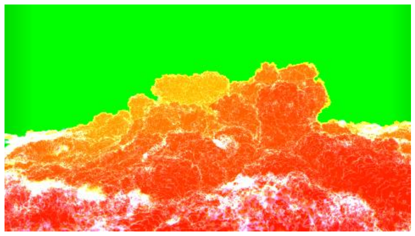
(b)



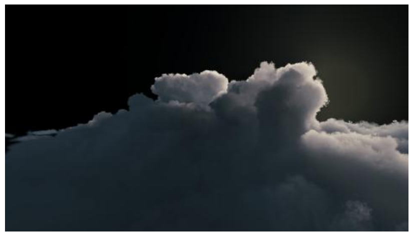
(c)



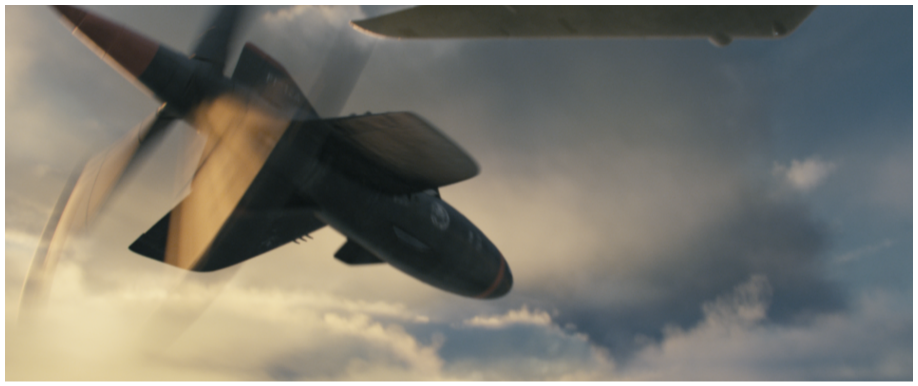
(d)



(e)



(f)



(g) Captain America: The First Avenger ©2011 Marvel Entertainment. All rights reserved.

Figure 27: Cloud passes: (a) ambient light, (b) angle, (c) mie scattering, (d) raw sun, (e) density pass (f) beauty render

7.6.4 Shadow Map Help

In order to increase the complexity at very little cost, we also made use of generic deep shadowmaps as blockers on our sun light. They were completely non dynamic, and had no relation to the actual cloud layout, but we found that renders for cloud layers below the top one generally looked at a lot better with them than with nothing. Doing things correctly, calculating correct shadows from every cloud onto every other cloud, would have been an organizational pain if done globally, and taken significant extra time, so was done only sparingly where we really felt the shot would benefit.

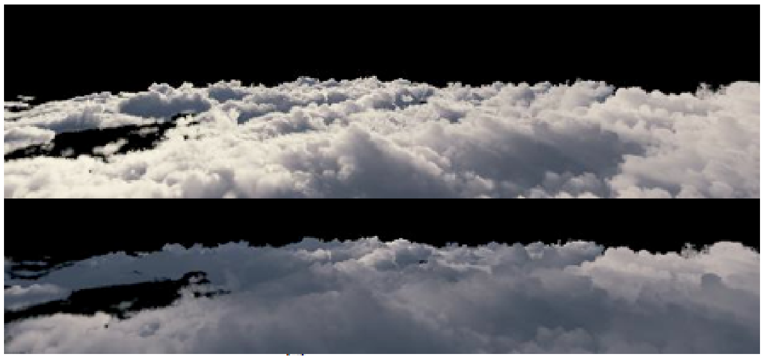


Figure 28: Cloud renders without, and with, generic projection deepshadowmaps acting as blockers

We felt the end result (figure 29) was very successful, helped of course by gorgeous magic hour lighting. It's a testament to Nick and the new DNB shader system and architecture that the client bought off on the look of the clouds with almost the first comp we presented, making for a far easier delivery than could typically be expected for scenes of such complexity.

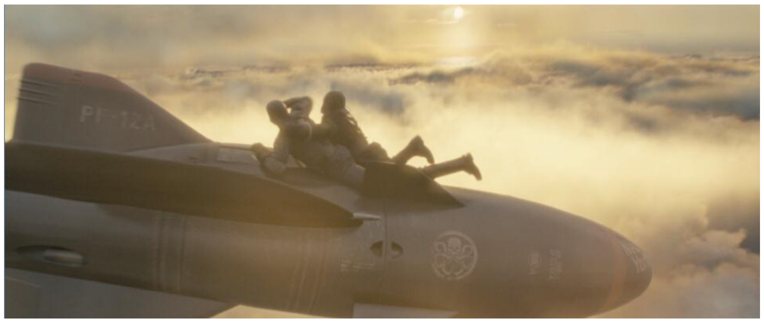


Figure 29: Captain America: The First Avenger ©2011 Marvel Entertainment. All rights reserved.