Volumetric Methods in Visual Effects

SIGGRAPH 2010 Course Notes

Course Organizers

Magnus Wrenninge[1]
Sony Pictures Imageworks

Nafees Bin Zafar[2]
DreamWorks Animation

Presenters

Jeff Clifford
Double Negative

Gavin Graham
Double Negative

Devon Penney
DreamWorks Animation

Janne Kontkanen
DreamWorks Animation

Jerry Tessendorf
Rhythm & Hues Studios

Andrew Clinton
Side Effects Software

Updated: 25 jul 2010

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. In addition to learning the basics of the technology, attendees will also be exposed to the rationales behind the sometimes drastically different development choices taken and solutions employed by the presenters, who have experience with and have authored proprietary and commercial volumetrics tools.

The course begins with a quick introduction to generating and rendering volumes. We then present a production usable volumetrics toolkit, focusing on the feature set and why those features are desirable. Finally we present the specific tools developed at Double Negative, DreamWorks, 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, multithreading/parallelization, holdouts, 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://groups.google.com/group/volumetricmethods

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. He is also involved with fluid simulation R&D and has worked as an effects TD on films such as Spiderman 3, Beowulf, Hancock and Alice In Wonderland. He holds an M.Sc. in Media Technology from Linköping University.

Jeff Clifford has been a member of the R&D department at Double Negative since 2002. He wrote the DNB voxel renderer in 2004 for Batman Begins and has developed it since. He has experience developing many 2D and 3D tools for film production rendering including DNeg's in-house particle renderer and HDRI digital stills pipeline. He holds an M.Sc. in Applied Optics from Imperial College London and has worked on various films including Harry Potter, The Dark Knight and 2012.

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.

Gavin Graham started working at Double Negative in 2000 as an effects TD. Over the next few years he did all manner of shot based effects work while he also assisted R&D in battle testing in-house tools such as DNA the particle renderer, DNB the voxel renderer, and Squirt, the fluid solver. He holds a Computer Science degree from Trinity College Dublin and an M.Sc. in Computer Animation from Bournemouth University. He has over the last few years CG Supervised on such effects heavy films as Stardust, Harry Potter 6, and 2012.

Janne Kontkanen Janne Kontkanen is a Senior Rendering Software Engineer at DreamWorks Animation. His main areas of responsibility and expertise are global illumination, ray tracing, volume rendering, and particle rendering. He has a PhD on rendering techniques from Helsinki University of Technology, Finland. Most of his research publications are in the areas of global illumination and quick ambient occlusion techniques.

Devon Penney is an effects animator at PDI/DreamWorks. He developed the volumetrics toolkit used at DreamWorks Animation. He is interested in performance issues in volume rendering, developing motion blur algorithms, and solutions towards efficient artist workflow. Prior to DreamWorks he obtained an M.S. in computer science from Brown University in 2007.

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.

Presentation schedule

9.00 – 9.15	**Introduction** *(Bin Zafar & Wrenninge)*
9.15 – 10.15	**Basics of volume modeling & rendering** *(Bin Zafar & Wrenninge)*
10.15 – 10.30	**Break**
10.30 – 12.15	**Advanced topics and production solutions**

10.30 – 10.50	**Rhythm & Hues** *(Tessendorf)*
10.50 – 11.10	**Side Effects Software** *(Clinton)*
11.10 – 11.30	**Dreamworks Animation** *(Penney & Kontkanen)*
11.30 – 11.50	**Double Negative** *(Clifford & Graham)*
11.50 – 12.10	**Sony Pictures Imageworks** *(Wrenninge)*

Table of contents

Introduction 6
- An informal history of volumetric effects 7
- A simple volumetrics system 10

Volume modeling 11
- Voxel buffers 12
- Writing to voxel buffers 18
- Interpolation 21
- Geometry-based volume modeling 23
- Rasterization primitives 27
- Instantiation-based primitives 32
- Modeling with level sets 41
- Motion blur 43
- High resolution voxel buffers 46

Volume rendering 51
- Lighting theory 52
- Raymarching 56
- Pre-computed lighting 60
- Holdouts 63
- Motion blur 67

Putting it all together 68

References & Further reading 69

1. Introduction

If you google "volume rendering" or search for it at your favorite book store web site, you will find that most available literature and research regards volume rendering in medical and data visualization contexts. A smaller portion deals with photorealistic rendering of light scattering in participating media, but precious few texts are available that describe how volume rendering is used in practice to create visual effects.

The aim of this course is to give an introduction to volume rendering in visual effects production. Production volume rendering is a fairly isolated subset of volume rendering, and there is little overlap between it and the other volume rendering contexts. We aim to cover only techniques actively used in visual effects production, and while this excludes much of current research into rendering of participating media, we want to highlight the techniques with the most practical applications. We further limit the scope of this course to rendering of true 3D volumes, excluding topics such as sprite- and slice-based methods.

Our goal is to provide enough details about how high-end production volume rendering is accomplished that participants could set about writing their own basic rendering software. Part of the course's purpose is also to discuss the limitations of the techniques used.

1.1. An informal history of volumetric effects

One of the most memorable volumetric effects in cinema history is the “cloud tank” effect from *Close Encounters of the Third Kind*. Developed by Scott Squires, this technique called for filling a tank partially with salt water, then carefully layering on lower density fresh water on top. The clouds were created by injecting paint into the top layer, where it would settle against the barrier between the salt water and the fresh water [Squires, 2009]. Beyond just art direction, this particular cloud effect was a character in it’s own way. The goals the special effects crew had during *Encounters* are the same goals we have today. We want to control how the volumetrics look, and how they move.

Cloudtank effect used in Independence Day.
© 1996 Twentieth Century Fox and Centropolis Entertainment. All rights reserved.

Computer graphics got into the mix shortly thereafter with William Reeves’ invention of particle systems. He used particle systems to create the Genesis sequence in *Star Trek II: The Wrath of Khan*. The title of the associated SIGGRAPH publication provides an excellent preview into what we are trying to do: *Particle Systems – A Technique for Modeling a Class of Fuzzy Objects* [Reeves, 1983]. This basic methodology is still prevalent today, and very relevant to this course.

With the advent of digital rotoscoping and compositing it became common practice in live action visual effects to film elements in staged shoots and composite them onto the plate. This allowed the creation of very complex photoreal effects, since the elements were real.

For purely digital effects, particle systems remained the popular choice. Their use in production indicated a certain look barrier to particle based volumetrics. Particles work great when they are used to model media which is well approximated by particles. The problem occurs when one tries to model a media which is meant to be continuous. The use of particles in these cases leads to a very discontinuous sampling. Particles could be combined with some low frequency tricks such as sprites to look good in special cases, but not so in the general case. One could simply choose to use more particles, but that

looses the advantage of the sparse sampling, and comes at an exponential increase in computational cost.

In the late 90's, an alternative approach started taking root in the visual effects industry [Kisacikoglu, 1998; Lokovic and Veach, 2000; Kapler, 2002]. This technique treated space as a discretized volume, where the contents of a given small volume of space is stored in a sample. The kinds of data stored are properties such as density, temperature, and velocity. Each volumetric sample is called a voxel, and the entire collection is referred to as a voxel buffer or voxel grid. Modeling operations are performed on the grid by rasterizing shapes, particles, or noise. Most morphological operations common in the image processing paradigm are also applicable to volumes. This familiarity in workflow also helped artists adapt to this voxel based pipeline.

The Nightcrawler's "Bamf" effect from X2.
© 2003 Twentieth Century Fox and Centropolis Entertainment. All rights reserved.

One of the first systems successfully used in multiple movies and commercials was the *Storm* software developed by Alan Kapler at Digital Domain. The goal behind *Storm* was to provide a modeling and rendering solution which could be operated efficiently by artists at the resolutions required for feature films. It featured a language where artists could create buffers, model volumetric shapes, perform arithmetic and compositing operations, and control rendering. The modeling commands allowed artists to use different geometric shapes and a rich set of noise algorithms to create high quality effects very quickly. The system was implemented as a plugin to the Houdini animation software which also aided in quick adoption by the artists.

The memory requirements of scenes *Storm* needed to render exceeded 25 gigabytes. A stringent requirement even by today's standards, it was impossible when Digital Domain started working on a CG

avalanche effect for Columbia Pictures' 2002 film *xXx*. Storm utilized in-core data compression techniques, and innovated to use of buffers transformed to fit the camera frustum. These buffers, called "frustum buffers", provided high resolution close to the camera, and low resolution but complete spatial coverage far away from the view point [Kapler, 2003]. For his pioneering efforts in the design and development of *Storm*, Alan Kapler received a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences in 2005.

Digital avalanche in xXx. © 2002 Columbia Pictures. All rights reserved.

The need for high quality volumetric effects led to the development of similar systems across the industry. Some of these systems will be presented in this course. These are the *FELT* system from Rhythm and Hues, the *Mantra* renderer from SideFX Software, *MF* and *d2r* from DreamWorks Animation, *DNB* from Double Negative, and *Svea* from Sony Imageworks. These systems share many common themes, as well as many unique features born out of the specific requirements of the effects for which they were used.

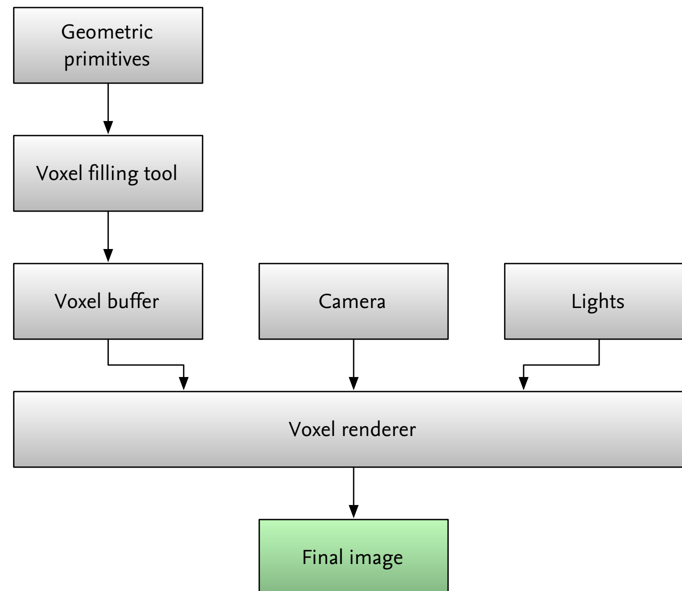
1.2. A simple volumetrics system

A minimal volumetrics system contains three major components. First a data structure for voxel buffers. This means defining a file, and in-core representations. A naive implementation is an object which contains a contiguous array, and provides accessor methods to access values with 3D grid indices or positions.

The second component consists of a set of operations which fill the buffer with data. One such operation may simply evaluate noise at each voxel, and store the value. Another operation may take a list of points with radii, and fill the spherical region around the particle with a given value. These modeling commands can involve filtering, distorting, and combining multiple voxel buffers with arithmetic operations. Each operation could be implemented as a separate command line tool, or one tool which processes a sequence of commands and arguments to these operations.

The final component is a renderer to produce an image of the voxel buffer. In addition to the buffer to render, this component also requires specifications for a camera, and lights.

A typical workflow is to model and animate some primitives such as a set of particles, or meshes. Then one creates a voxel buffer around the location of these primitives. The primitives are then rasterized into the voxel buffer. Further volumetric processing operations are performed. Finally the buffer is rendered by the volume rendering component. The next three chapters will expand further upon these components.



A very simple volume modeling and rendering system

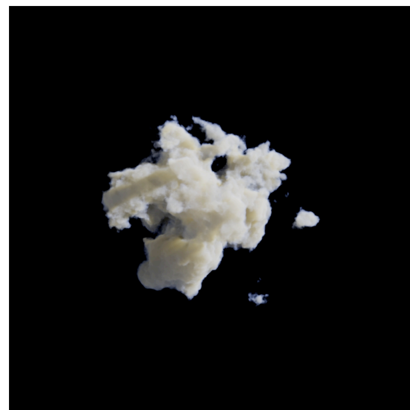
2. Volume modeling

In other forms of volume rendering, such as medical visualization, the data to be rendered is directly available to the system, as in the case of a CT or MRI dataset. When it comes to volume rendering in visual effects, we need to create this data ourselves. The process is called volume modeling, and involves turning geometric data into volumetric data, most often in the form of voxel buffers.

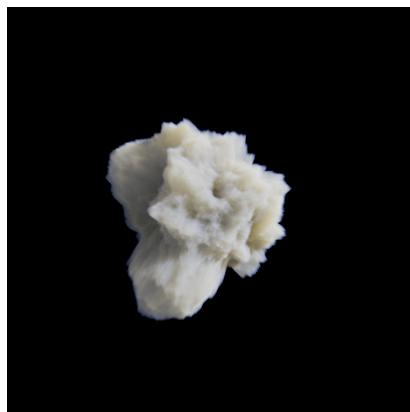
A classic example of volume modeling is the use of pyroclastic noise primitives to model billowing smoke, where each primitive is a sphere that can be represented as a position, a radius and various noise parameters. The use of simple geometric primitives combined with noise functions is one of the most fundamental methods of volume modeling.

Volume modeling is an almost endless topic, as there is an infinite number of ways and methods that one can fill a voxel buffer. This chapter will try to describe the basics, from the voxel data structures needed and elementary modeling primitives, to techniques for scaling to high resolution data sets.

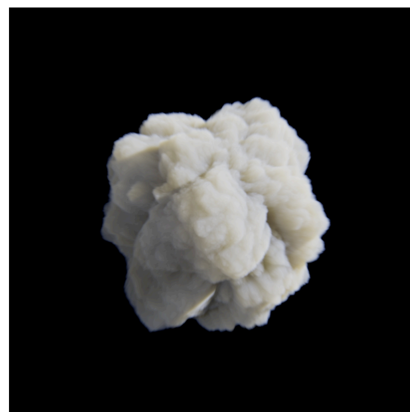
Simple sphere



Windowed noise function



Displacement based on noise

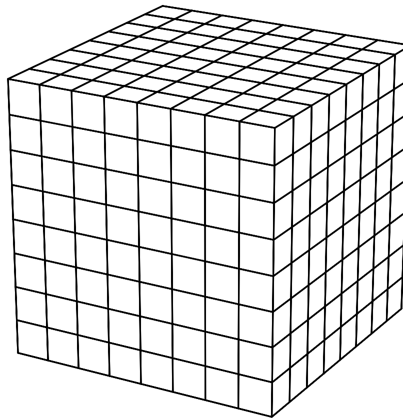


Pyroclastic noise

2.1. Voxel buffers

An ordinary computer image is a two-dimensional orthogonal array that stores either single values (for a grayscale image), or multiple values (for a spectral image, such as RGB). The concept translates directly to three dimensions, where we can imagine a 3D orthogonal array, which stores single or multiple values in each of its cells.

This 3D array goes by many names, such as *voxel grid*, *voxel volume*, *voxel buffer*, etc., and depending on whether it stores scalar- or vector-valued data it is sometimes also be referred to as a *scalar field/buffer/grid* or *vector field/buffer/grid*. Throughout these course notes we will refer to the case of discrete voxel arrays used in a program as *voxel buffers*. In the general case of non-voxelized, arbitrary functions in 3-space, we will refer to those as *fields*.



8x8x8 *resolution orthogonal (uniform) voxel grid*

2.1.1. Implementations

There are countless ways to implement voxel buffers. The simplest ones fold the 3D space into a contiguous 1D array, and store the data using `float*` and `malloc()`, or in a `std::vector<float>`. More complex implementations may allocate voxels as-needed, allowing the size and memory use to scale dynamically. Such techniques become important as the resolution of a voxel buffer increases. Densely allocated buffers are manageable up to resolutions of roughly 1000^3. (On very high-memory machines this may stretch to 2000^3 or so.) To reach higher resolutions we need to use different data structures, such as sparsely allocated buffers. We will return to this topic and ways of dealing with it in the section titled *High resolution voxel buffers.*

Though implementing a simple voxel buffer class is straightforward, there are also free, open source libraries. Field3D[3] is one alternative, which has the benefit of being developed and tested in production for volume rendering and fluid simulation. We will use Field3D's data structures in our examples and pseudo-code.

[3] http://field3d.googlecode.com

2.1.2. Voxel indices

Just as with a 2D image, we can access the contents of a voxel by its coordinate. The bottom left corner of the buffer has coordinate [0,0,0] (unless a custom data window is used, see below), and its neighbor in the positive direction along the *x* axis is [1,0,0]. When referring to the index along a given axis, it is common to label the variable *i*, *j* and *k* for the *x*, *y* and *z* axes respectively. In mathematic notation this is often written using subscripts, such that a voxel buffer called *S* has voxels located at $S_{i,j,k}$.

In code, this translates directly to the integer indices given to a voxel buffer class' accessor method, such as:

```
class DenseField
{
  const float& value(int i, int j, int k)
  {
     // ...
  }
  // ...
};

float a = buffer.value(0, 0, 0);
```

2.1.3. Implementation awareness

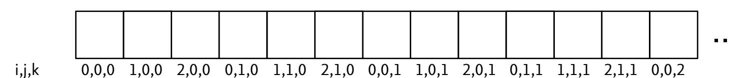
Although it is easy to write code that is agnostic about how voxels are represented in memory, writing efficient code usually means being aware of and taking advantage of the implementation's underlying data structure. For example, a trivial voxel buffer may store its data as a contiguous one-dimensional array, such as

```
float *data = new float[xSize * ySize * zSize];
```

Where the mapping of a 3D coordinate to its 1D array index is calculated as

```
int arrayIndexFromCoordinate(int i, int j, int k)
{
  return i + j * xSize + k * xSize * ySize;
}
```

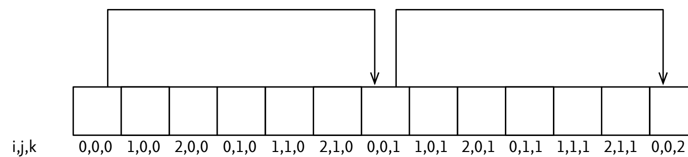
The memory for such a buffer has the following structure:



If we were to loop over all the voxels in the buffer, for example to clear all the values, we might write it as follows:

```
// Naive loop, with x dimension outermost
for (int i = 0; i < xSize; ++i) {
  for (int j = 0; j < ySize; ++j) {
     for (int k = 0; k < zSize; ++k) {
        buffer.lvalue(i, j, k) = 0.0f;
     }
  }
}
```

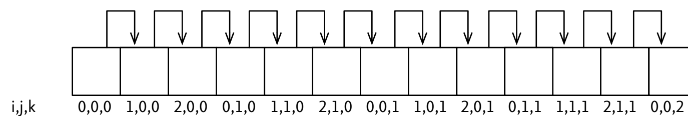
The problem with the code above is that the inner loop steps along the *z* axis, which means the memory access pattern has a stride of `xSize * ySize`. For a buffer of realistic resolution, this will most likely cause a cache miss at each voxel increment and force an entire cache line to be loaded, which cripples performance.



Access pattern for the naive loop

If we instead reorder the loop so that the *x* axis is the innermost, performance is improved since the access pattern is sequential in memory.

```
// Better loop, with x axis innermost
for (int k = 0; k < zSize; ++k) {
  for (int j = 0; j < ySize; ++j) {
     for (int i = 0; i < xSize; ++i) {
        buffer.lvalue(i, j, k) = 0.0f;
     }
  }
}
```



Access pattern for the improved loop

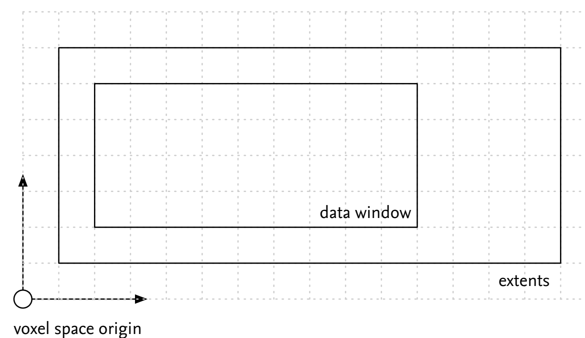
Of course, we are still doing the multiplication to find the 1D array index once per voxel access, something that could be avoided through the use of *iterators*. Iterators allow code to be written without explicit bounds checks in all dimensions:

```
for (DenseField<float>::iterator i = buffer.begin(); i != buffer.end(); ++i) {
  *i = 0.0f;
}
```

Iterators have the benefit of both being more efficient and producing cleaner code. We refer the interested reader to the Field3D programmer's guide[4] for a more in-depth look at iterators.

2.1.4. Extents and data windows

As mentioned earlier, voxel indices do not need to start at [0, 0, 0]. As a parallel, images in the OpenEXR file format have a *display window* and *data window* that specify the intended size and the allocated pixels of an image. The same concept translates well to voxel buffers, where we will refer to the intended size of the buffer as *extents* and the region of legal indices as *data window*.



2D example of extents and data window

In the illustration above, the *extents* (which defines the [0, 1] local coordinate space) is greater than the *data window*. It would be the result of the following code:

```
Box3i extents(V3i(1, 1, 0), V3i(15,7,10));
Box3i dataWindow(V3i(2, 2, 0), V3i(11, 6, 10));
buffer.setSize(extents, dataWindow);
```

Using separate extents and data window can be helpful for image processing (a blur filter can run on a padded version of the field so that no boundary conditions need to be handled), interpolation (guarantees that a voxel has neighbors to interpolate to, even at the edge of the extents) or for optimizing memory use (only allocates memory for the voxels needed).

---

[4] http://code.google.com/p/field3d/downloads/list

2.1.5. Coordinate spaces and mappings

The only coordinate space we've discussed so far is the voxel buffer's native coordinate system. In the future, we will refer to this coordinate space as *voxel space*. In order to place a voxel buffer in space we also need to define how to transform a position from *voxel space* into *world space* (which is the global reference frame of the renderer). Besides *voxel* and *world space*, a third space is useful, similar to RenderMan's NDC space but local to the buffer. This *local space* defines a [0, 1] range over all voxels and is used as a resolution independent way of specifying locations within the voxel buffer. This definition is the same as Field3D uses.

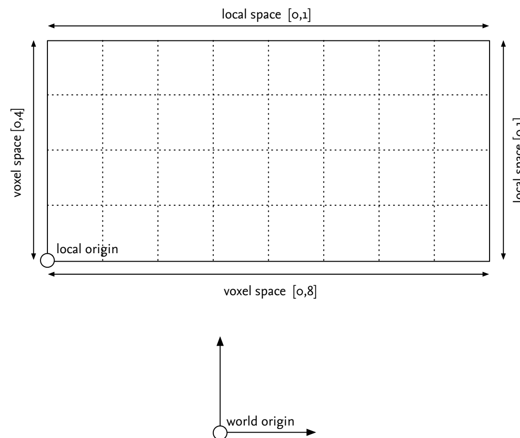


Illustration of coordinate spaces

When constructing a voxel buffer we define a *localToWorld* transform in order to place the buffer in space. This transform is also called *mapping*, and defines the transformation between *local space* and *world space*. Note that the transformation from *local space* to *voxel space* is the same regardless of the buffer's location in space. To sum things up:

- *World space* is the global coordinate system and exists outside of any voxel buffer.
- *Local space* is a resolution-independent coordinate system that maps the full *extents* of the voxel buffer to a [0, 1] space.
- *Voxel space* is used for indexing into the underlying voxels of a field. A field with 100 voxels along the *x* axis maps [100.0, 0.0, 0.0] in *voxel space* to [1.0, 0.0, 0.0] in *local space*.

As a matter of convenience and clarity, we will prefix variables in code and pseudocode with an abbreviated form of the coordinate space. A point `P` in *world space* will be called `wsP`, in *voxel space* `vsP`, and in *local space* `lsP`.

2.1.6. Integer versus floating-point coordinates

Voxel space is different from *local* and *world space* in that it can be accessed in two ways – using integer or floating-point coordinates. Integer access is used for direct access to an individual voxel, and floating-point coordinates are used when interpolating values. It is important to take care when converting between the two. The center of voxel [0, 0, 0] has floating-point coordinates [0.5, 0.5, 0.5]. Thus, the edges of a field with resolution 100 are at 0.0 and 100.0 when using floating-point coordinates but when indexing using integers, only 0 through 99 are valid indices. An excellent overview of this can be found in an article by Paul S. Heckbert – *What Are The Coordinates Of A Pixel?* [Heckbert, 1990]

In practice, it is convenient to define a set of conversion functions to go from `float` to `int`, and `Vec3<float>` to `Vec3<int>`, etc. In this course we will refer to these conversion functions as `discreteToContinuous()` and `continuousToDiscrete()`.

```
int continuousToDiscrete(float contCoord)
{
  return static_cast<int>(std::floor(contCoord));
}

float discreteToContinuous(int discCoord)
{
  return static_cast<float>(discCoord) + 0.5f;
}
```

2.2. Writing to voxel buffers

The fundamental purpose of a voxel buffer is obviously to read and write to it. In this section we will consider a few different ways of writing voxel data, and the methods will serve as the foundation for all subsequent modeling techniques.

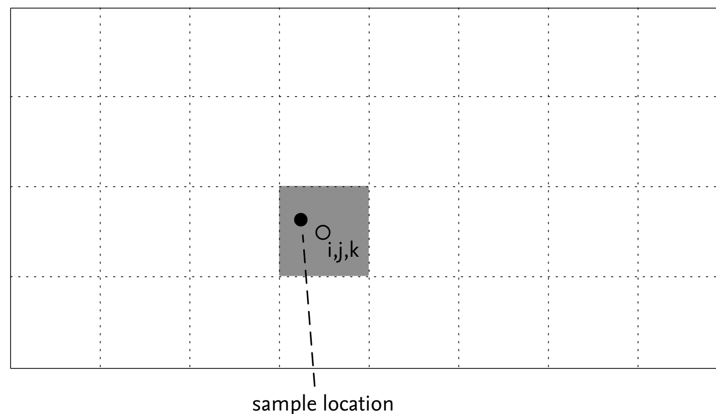
For purposes of illustration, let's consider a simple C++ function for writing a floating-point value to a given voxel:

```
void Rasterizer::writeVoxel(const int i, const int j, const int k, float value)
{
  buffer.lvalue(i, j, k) += value;
}
```

Writing a value directly at a voxel location doesn't get us very far in terms of modeling complex voxel buffer however. As it turns out, the most common modeling operation is the writing of a value in-between voxels. In these notes we will refer to this as *splatting*, though it is sometimes also called *stamping* and *baking* a sample.

2.2.1. Nearest neighbor splat

The simplest way to splat a value that lies in-between voxels is to simply round the coordinates to the nearest integers. While this has some obvious aliasing problems, it can sometimes be a reasonable solution, especially when writing large quantities of low-density values which will blend when taken together.



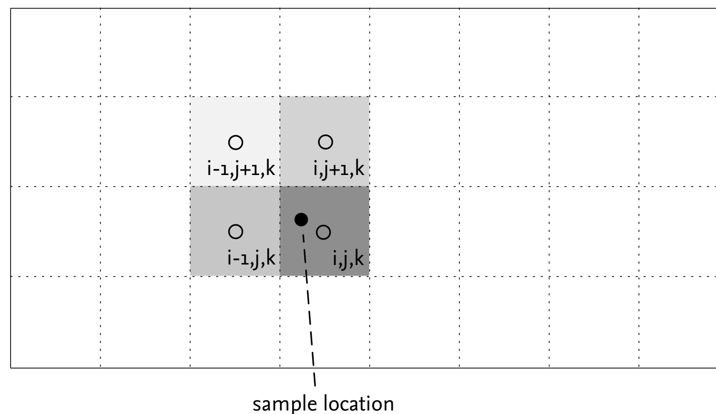
Splatting a sample using the nearest-neighbor strategy

This method can be implemented trivially as:

```
void Rasterizer::nearestNeighborSplat(const V3f &vsP, float value)
{
  // We can use continuousToDiscrete, since it does the same thing
  // that we need - finding which voxel the sample point is
  // located in.
  V3i dVsP = continuousToDiscrete(vsP);
  // Once the voxel is known we use writeToVoxel for access
  writeToVoxel(dVsP.x, dVsP.y, dVsP.z, value);
}
```

2.2.2. Trilinear splat

If antialiasing is important we can use a filter kernel when writing the value. The simplest, and most commonly used form is a triangle filter with a radius of one voxel. This filter will at most have non-zero contribution at eight voxels surrounding the sample location. The value to be written is simply distributed between its neighboring voxels, each weighted by the triangle filter.



Splatting a sample using the trilinear strategy

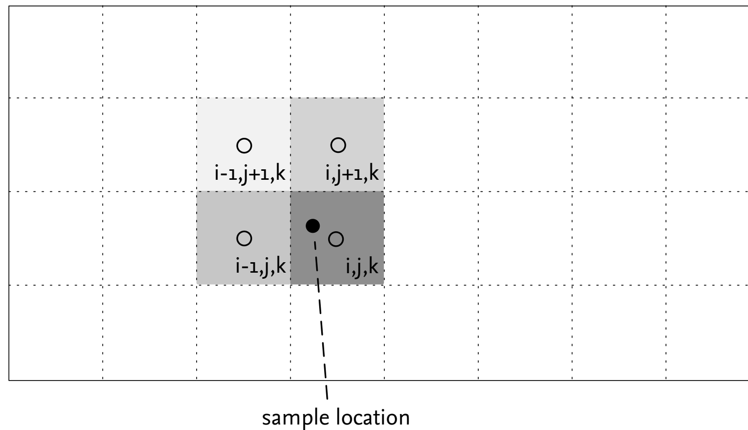
A simple implementation would be:

```
void Rasterizer::trilinearSplat(const V3f &vsP, float value)
{
  // Offset the voxel-space position relative to voxel centers
  // The rest of the calculations will be done in this space
  V3f p(vsP.x - 0.5, vsP.y - 0.5, vsP.z - 0.5);
  // Find the lower-left corner of the cube of 8 voxels that
  // we need to access
  V3i lowerLeft(static_cast<int>(floor(p.x)),
                static_cast<int>(floor(p.y)),
                static_cast<int>(floor(p.z)));
  // Calculate P's fractional distance between voxels
  // We start out with (1.0 - fraction) since each step of the loop
```

```
  // will invert the value
  V3f fraction(V3f(1.0f) - (static_cast<V3f>(c + V3i(1)) - p));
  // Loop over the 8 voxels and distribute the value
  for (int k = 0; k < 2; k++) {
    fraction[2] = 1.0 - fraction[2];
    for (int j = 0; j < 2; j++) {
      fraction[1] = 1.0 - fraction[1];
      for (int i = 0; i < 2; i++) {
        fraction[0] = 1.0 - fraction[0];
        double weight = fraction[0] * fraction[1] * fraction[2];
        m_buffer.lvalue(c.x + i, c.y + j, c.z + k) += value * weight;
      }
    }
  }
}
```

2.3. Interpolation

In order to sample an arbitrary location within a voxel buffer we have to use interpolation. The most common scheme is trilinear interpolation which computes a linear combination of the 8 data points around the sampling location. The concept and implementation are very similar to the trilinear splatting described above.



2D illustration of linear interpolation

Depending on the look required, it may be desirable to use higher order interpolation schemes. Such schemes will come at an increased computational cost. Profiling reveals that a significant portion of the runtime of a volume renderer is spent interpolating voxel data. The primary reason is that a naive voxel buffer data structure offers very poor cache coherence. A tiled data storage scheme combined with structured accesses will improve overall performance, but will require a more complicated implementation.

The following is an implementation of trilinear interpolation:

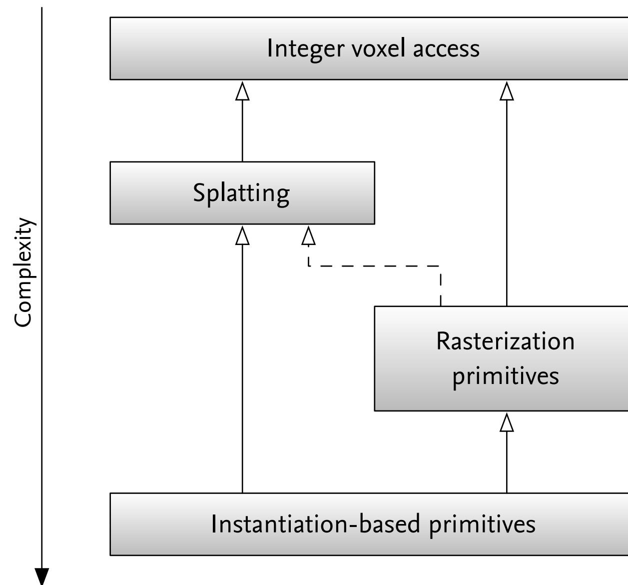
```
float Sampler::trilinearInterpolation(const V3f& vsP)
{
  // Offset the voxel-space position relative to voxel centers
  // The rest of the calculations will be done in this space
  V3f p(vsP.x - 0.5, vsP.y - 0.5, vsP.z - 0.5);
  // Find the lower-left corner of the cube of 8 voxels
  // that we need to access
  V3i lowerLeft(static_cast<int>(floor(p.x)),
                static_cast<int>(floor(p.y)),
                static_cast<int>(floor(p.z)));
  float weight[3];
  float value = 0.0;
  for (int i = 0; i < 2; ++i)
  {
    int cur_x = lowerLeft[0] + i;
    weight[0] = 1.0 - std::abs(p[0] - cur_x);
```

```
    for (int j = 0; j < 2; ++j)
    {
      int cur_y = lowerLeft[1] + j;
      weight[1] = 1.0 - std::abs(p[1] - cur_y);
      for (int k = 0; k <= 1; ++k)
      {
        int cur_z = lowerLeft[2] + k;
        weight[2] = 1.0 - std::abs(p[2] - cur_z);
        value += weight[0] * weight[1] * weight[2] * buffer.value(cur_x, cur_y, cur_z);
      }
    }
  }
  return value;
}
```

2.4. Geometry-based volume modeling

There is an almost infinite number of ways that the voxels around a primitive can be filled with data. In these notes we will cover the most fundamental approaches and also discuss their benefits and drawbacks, and in what circumstances they are used.

In the previous sections we discussed direct (integer) voxel access, and how to splat filtered samples into a voxel buffer. These can be thought of as the first two layers in the voxel modeling pipeline.



Outline of the volume modeling abstraction hierarchy

The third layer is the rasterization layer. *Rasterization primitives* include types such as pyroclastic points, splines and surfaces, but can be any primitive that is converted voxel-by-voxel into a volumetric representation. These rasterization process normally accesses voxels directly (i.e. using the integer voxel access layer), although when considering motion blur they may also use the splatting layer.

The fourth layer is *instantiation-based primitives*. They are referred to by different names at different facilities, sometimes also called *wisps* or *generators*. These primitives are composed of instances of the lower-level primitives, and either create rasterization primitives, or directly create sample points to be splatted. Instantiation-based primitives may also generate other instances of their own or other instantiation primitive types. Because of this potentially recursive nature, they can be very powerful.

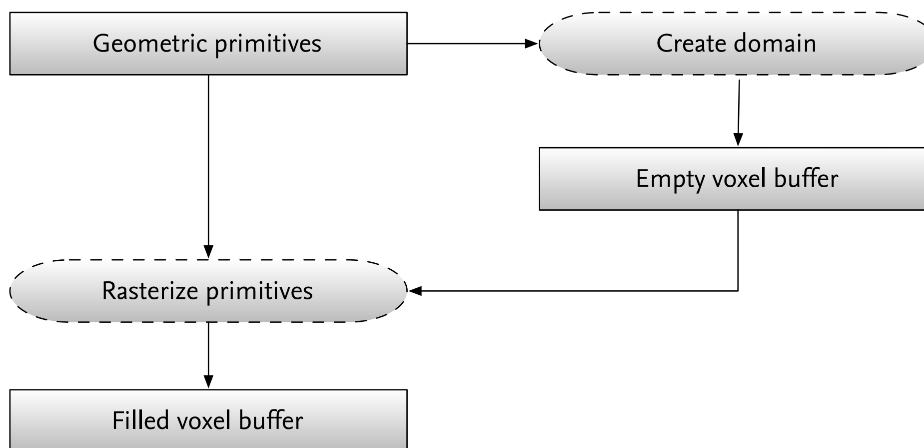
The third and fourth layers can be thought of as two quite different approaches to volume modeling, even though they are often used in conjunction. A useful comparison is that of the difference between a raytracing-based renderer and a micropolygon-based one. Rasterization is similar to raytracing in that it considers each voxel in turn and decides how primitives contribute to it. Instantiation-based primitives (and micropolygon renderers) see primitives as the first-class citizen, and considers which voxels are

affected by a given primitive. Rasterization-based modeling *pulls* values into a voxel, and instantiation-based modeling *pushes* values into voxels.

2.4.1. Defining voxel buffer domains

The first step in volume modeling is to determine the *domain* of the voxel buffer that is being created, so that the buffer encloses the space of the primitives that are being rasterized. A very basic implementation might simply compute an axis-aligned or oriented bounding box for the incoming primitives, but a robust solution needs to consider other factors. For example, almost all volumetric primitives extend out past their geometric representation. If the system allows users to create new primitives as plug-ins, it is important to communicate the bounds of a primitive back to the renderer during the domain calculation. This is especially true for primitives that include displacements driven by user input. Although it is possible to let the user dial displacement bounds manually, usability is improved if they can be handled automatically.

The motion of a primitive also needs to be considered in order to provide enough room to store the entire length of the sample. Motion blur techniques are discussed further in subsequent sections.

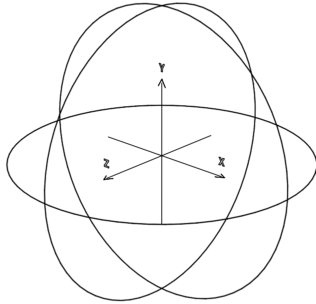


2.4.2. Noise coordinate systems

Volume modeling often, if not always, uses noise functions to add detail to the primitives. Noise functions need to be tied to a coordinate system, but almost any geometric primitive that has a reasonable parameterization method for a 3D coordinate system may be used.

For some primitives, such as a sphere, the coordinate system is trivial. Others, such as splines, require a little bit more work. The important thing to consider is that the parameterization should be smooth and reasonably quick to transform in and out of. Therefor we prefer transformations with closed-form solutions, rather than ones that require numerical iteration to find a solution, but as we will see, not all primitives used in production satisfy this preference.

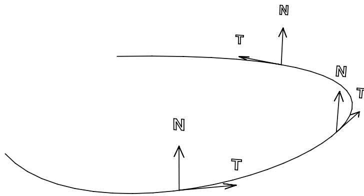
For a sphere, defining a coordinate system is simple. We simply use the object space as the noise coordinate space. Transformations in and out of this coordinate system can be calculated as a simple matrix multiplication.



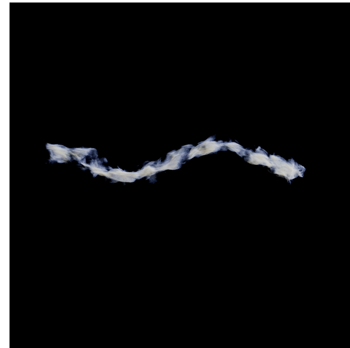
Coordinate system for a sphere (x, y, z)

Primitive with cartesian coordinate system

For a curve or spline it is most common to use a coordinate system that deforms along with the spline. The tangent of the curve itself is used as one basis, and the normal direction of the curve (orthogonal to the tangent) is used as the second. The third basis can be computed as the cross product of the first two. The curve may be a polygonal line or a parametric curve, but regardless of how the vertices are interpolated the transform in and out of this space is much more costly than for a sphere.

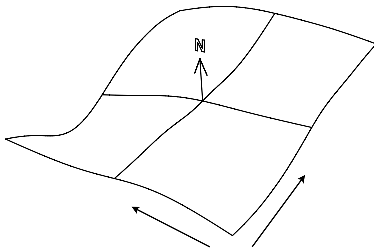


Coordinate system for a curve ($N \times T$, N, T)

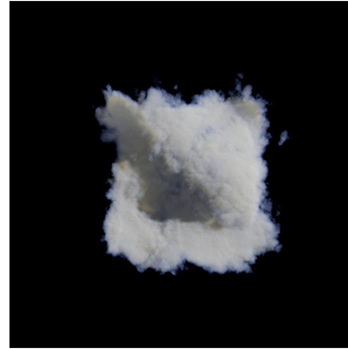


Primitive with curve-based coordinate system

A polygon mesh or surface patch can conveniently be parameterized using the *dP/du* and *dP/dv* partial derivatives as the first two bases, and the normal direction as the third basis. Just like curves, the transformation into this space is costly as the surface primitive may be composed of an arbitrary number of parametric primitives, which need to be searched.



Coordinate system for a surface (dP/du, dP/dv, N)



Primitive with surface-based coordinate system

2.5. Rasterization primitives

Rasterization is the process of building volume data voxel-by-voxel. Fundamentally, there are two approaches to rasterizing. The first is to visit each voxel in the buffer once, the second is to visit each primitive once. Depending on the way the voxel data is stored, one may be more appropriate than the other. For example, some renderers store voxel data as a set of 2D images on disk, each compressed using some form of non-lossy scheme. The overhead of pulling a slice from disk and decompressing it into memory is quite expensive, in which case the better approach is to visit each voxel only once. For these notes, we will assume that the buffer used for rasterization is fully loaded in memory and that there is no penalty for accessing neighboring voxels in any direction (other than potential cache misses) and use the second approach of visiting each primitive once.

2.5.1. Rasterization algorithm

In its most generic form, rasterization involves instancing the primitive representation, bounding it, looping over the voxels that it overlaps, and sampling its density function at each voxel.

```
void Rasterizer::rasterizePrims(const Geometry &geometry, VoxelBuffer &buffer)
{
  int numPrims = numPrimitives(geometry);
  for (int iPrim = 0; iPrim < numPrims; ++iPrim)
  {
     // Set up the primitive object, which contains the logic for sampling its density
     // This call picks up per-prim noise settings and other parameters from the geometry
     RasterizationPrim prim = setupSomePrimitive(geometry, iSphere);
     // Calculate the voxel-space bounding box of the primitive
     BBox vsBBox = voxelSpacePrimBounds(prim, buffer.mapping());
     // Loop over voxels
     for (int k = vsBBox.z.min; k < vsBBox.z.max; ++k) {
        for (int j = vsBBox.y.min; j < vsBBox.y.max; ++j) {
           for (int i = vsBBox.x.min; i < vsBBox.x.max; ++i) {
              // Voxel-space and world-space position of the voxel's center
              V3f vsP, wsP;
              vsP = discreteToContinuous(i, j, k);
              buffer.mapping().voxelToWorld(vsP, wsP);
              // Evaluate the density of the sphere primitive
              float density = prim.evaluate(wsP);
              // Store result in voxel buffer
              writeVoxel(i, j, k, density);
           }
        }
     }
  }
}
```

The `setupSomePrimitive` call is responsible for creating the object that determines the primitive's density function. It finds the current primitive's geometry and attributes.

The second step, `voxelSpacePrimBounds`, normally just returns the voxel-space bounding box of the primitive's vertices, with added padding to account for displacement.

Once the primitive is prepared and the region of voxels to traverse is known, the primitive is evaluated at each voxel location and the density is recorded in the voxel buffer.

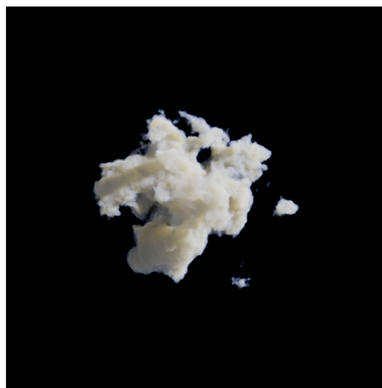
2.5.2. Rasterizing primitives

Sphere-based primitives always carry two fundamental attributes: their position (center), and their radius. On top of these an arbitrary number of attributes are used to define the various noise parameters than control its look.

For a sphere-shaped primitive the bounding box is a fairly tight fit, but for curves and surfaces many voxels will be calculated that lie far away from the primitive's region of influence. This has the downside of causing lots of unnecessary voxels to be computed. The rasterization loop can be improved in those cases, for example by determining the distance to a primitive before calculating the density function. However, even with that optimization, the world-to-local space transform is quite expensive for curves and surfaces, and in practice point-instantiation techniques are used for those types of primitives. The next chapter will describe that approach in more detail.

2.5.3. Solid noise primitives

One of the most straight-forward sphere-based primitives is the solid noise primitive. It uses the location of the sphere and its radius to "window" a noise function, so the density function is simply the sum of the windowing function and a fractal function.



Solid noise point

The function can be written as:

$noiseDensity(P) = fbm(P) + (1 - |P/radius|)$

where P is in the local space of the primitive. We notice that because of the fractal function, the density function may be positive outside the radius of the sphere. If the maximal amplitude of the fractal function fbm is A, it follows that the function has non-negative values at most A units away from the radius, as $A + (1 - |1+A|) = 0$.

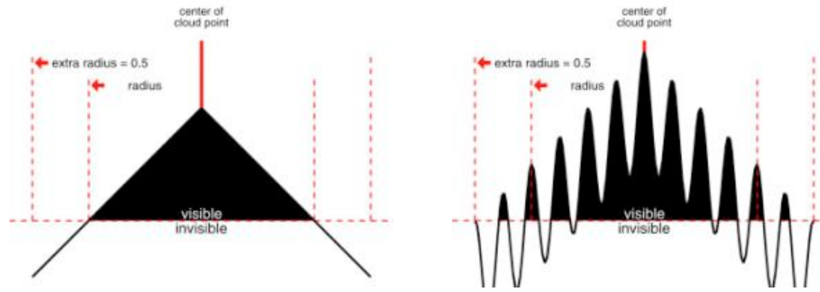


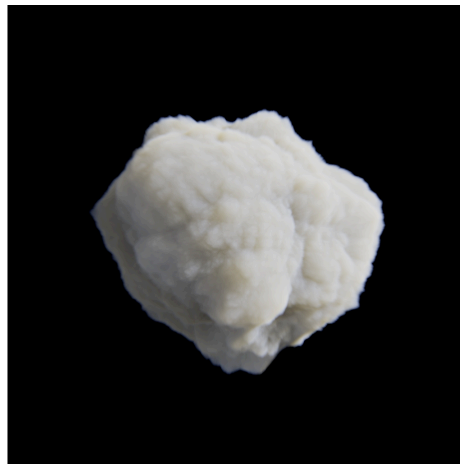
Illustration of density function and the required bounds padding

Because of this added distance, solid noise points are an example of a primitive that requires padding of its bounds calculations (as mentioned in *Defining voxel buffer domains*).

2.5.4. Pyroclastic sphere primitives

Pyroclastic primitives have been mentioned several times so far, so let's see how one can be implemented. A pyroclastic noise function uses a distance function to determine its location in the scene, and adds a procedural noise value (usually a fractal function, for example *fractal brownian motion*[5]) to the distance function. By thresholding the final value we create the pyroclastic look, although for antialiasing purposes it's better to use a smoothstep function, so that the transition in density is gradual.

pyroclasticDensity(P) = max(radius - |P/radius| + abs(fbm(P)), 0)
density = smoothStep(pyroclasticDensity, 0, 0.05)



A single pyroclastic noise primitive

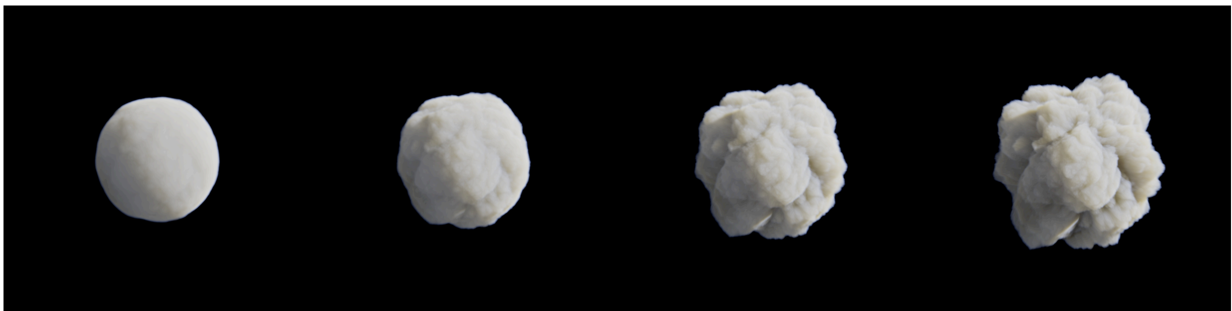
[5] For a complete description of the fbm function, see *Ebert et al – Texturing & Modeling (Morgan Kaufmann publ.)*

The pyroclastic look comes from using the noise function as a displacement, rather than by directly rendering it, and because the displacement is done per-voxel on the distance function itself, it is possible to produce overhangs, where parts of the density disconnects from the main body. If this is undesirable, the noise function lookup point can be projected onto the sphere primitive, effectively making the displacement amount constant for all points along the same normal vector.

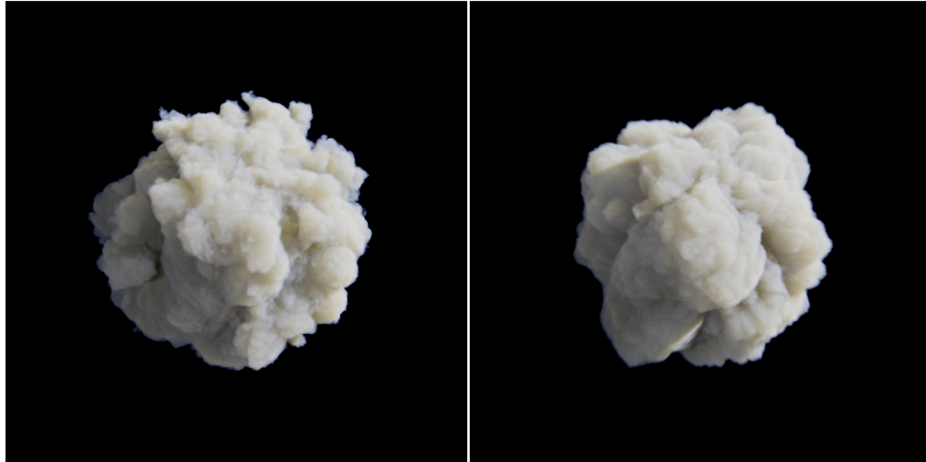
In its simplest form, the code would be:

```
// Assume that the primitive has already picked up all of its parameters,
// i.e. data members, from the geometry during the setup() call.
float PyroclasticSpherePrim::evaluate(Vector wsP)
{
  // Position in local space (i.e. relative to primitive)
  V3f lsP = wsP - m_position;
  // This projection lets us switch between 2D and true 3D displacement
  if (restrictDisplacementTo2D) {
     lsP = projectToSphereSurface(lsP);
  }
  // Position in noise space
  V3f nsP = lsP / m_scale + m_noiseOffset;
  // Compute distance function
  float distanceToCenter = lsP.length();
  // Compute noise function
  float noiseValue = fbm(lsP, m_octaves, m_gain, m_lacunarity) * m_amplitude;
  // Modulate distance function by noise
  float modulatedDistance = distanceToCenter + noiseValue;
  // Return full density if modulated point is within radius of implicit sphere
  // This can be replaced with a gradual change as density increases, if preferred
  float density = 0.0f;
  if (modulatedDistance < m_radius) {
     density = m_density;
  }
  return density;
}
```

By varying the noise parameters (amplitude, scale, gain), and animating the noise offset, it is possible to create a wide range of looks even from such a simple primitive. And yet more variations can be had by using a vector-valued noise function to displace the sample point used by the pyroclastic noise function.



Varying noise amplitude



3D displacement vs. 2D displacement

2.5.5. Sampling and antialiasing

Rasterization is prone to aliasing artifacts and sampling problems in the same way that surface rendering is. Where the projected pixel size, or the spot size, is used in surface rendering to antialias shader functions, we can use the voxel size to do the same in volume rasterization. The sampling frequency is simply the inverse of the voxel size. Once the sampling frequency is knows, we can apply the same frequency clamping and other antialiasing techniques as used in surface shading. Larry Gritz' section in the 1998 Advanced RenderMan SIGGRAPH course notes [Gritz, 1998] is a good starter.

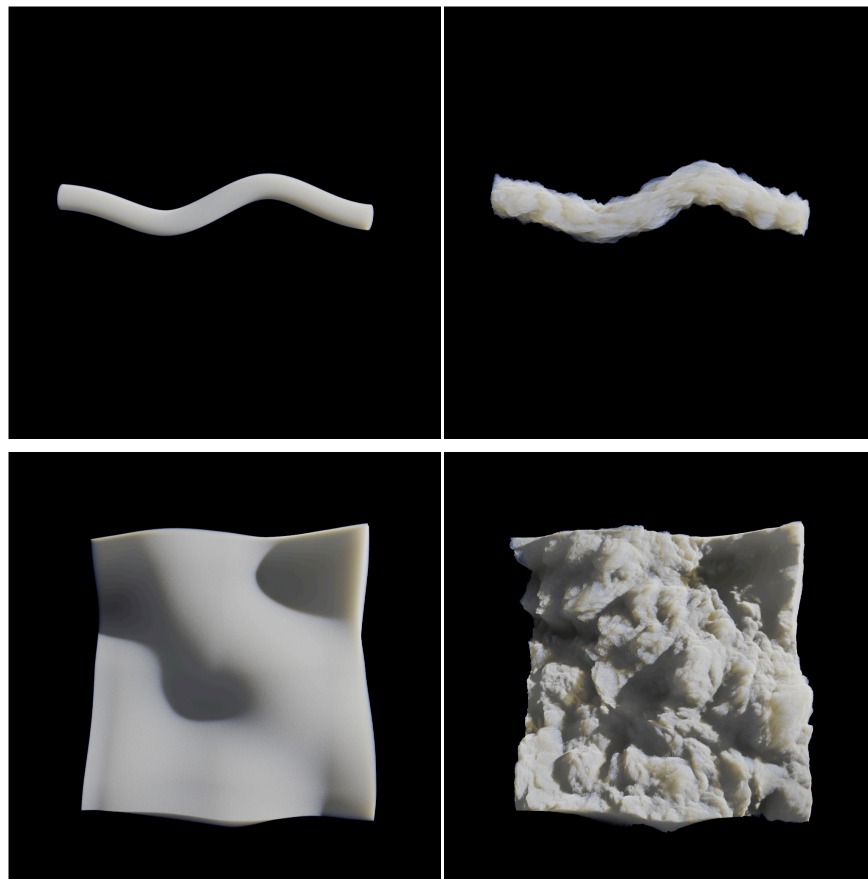
Similarly to how the sample positions in surface rendering may be randomized, we can add a small offset to each voxel's sample position using some function with a nice poisson distribution that prevents sample locations from bunching up.

2.6. Instantiation-based primitives

Rasterization primitives work well when modeling clouds, fog and other phenomena that are inherently continuous in nature and where the primitives fill in a major portion of the voxels in the buffer. In cases where primitives contain a lot of negative space the overhead of traversing and calculating the density function for all voxels can be quite substantial, and the more sparse the primitives become, the worse the performance. The problem is inherent to the pull-nature of the rasterization algorithm, and it is difficult to optimize away the wasteful sampling without incurring too much overhead in bookkeeping. Instantiation-based primitives avoid this problem as their push-nature means that calculations only take place for parts of the primitive that actually contribute density. This means that calculation costs are proportional to the amount of voxel data that is actually visible, instead of proportional to the coverage of the base primitive, as in the case of rasterization primitives.

Although instantiation-based primitives theoretically can instantiate any other primitive, the most common case is the class which instantiates points (usually in very large numbers) to fill in a volume. We will refer to those as point instantiation primitives.

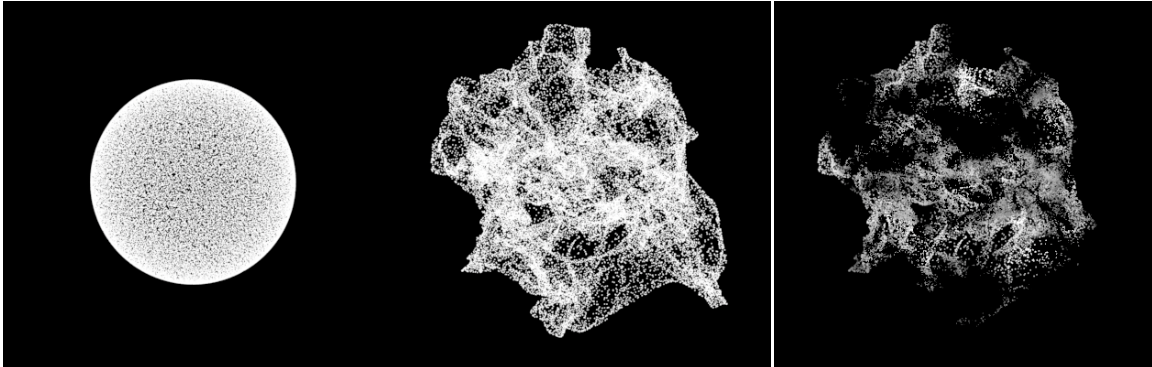
Point instantiation primitives have many advantages beside their inherent efficiency. They also support the full range from smooth-looking to granular primitives, simply by varying the number of instances used to fill the primitive.



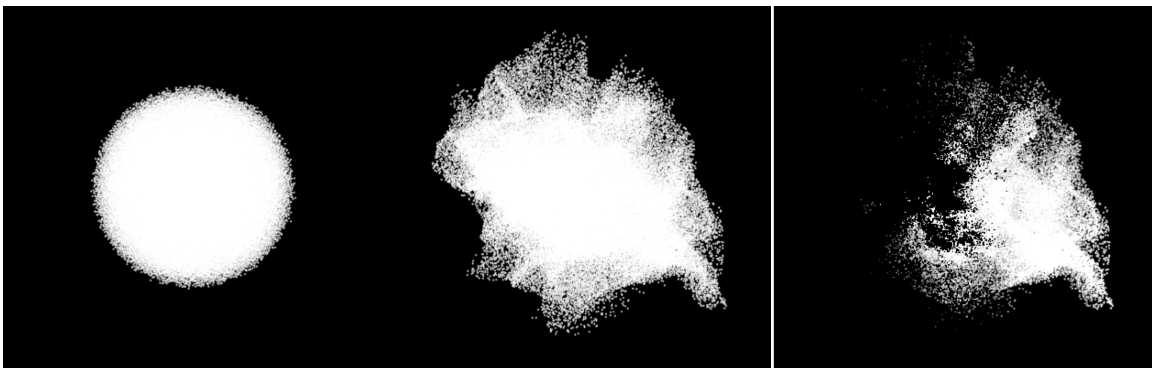
Examples of primitives using point instantiation

2.6.1. Point instantiation

The simplest point instantiation primitive uses a sphere as its base primitive. The first step in its generation is the scattering of points. Depending on the desired look, the scattering may be done only on the surface of the sphere, or inside the entire volume of the sphere. The images below show the different results of the two techniques.



left) Points scattered at sphere radius
middle) Points displaced by noise function
right) Point color modulated by noise function



left) Points scattered to fill inside of sphere
middle) Points displaced by noise function
right) Point color modulated by noise function

Once the points are scattered, any number of noise- or texture-based modulations and displacements may occur. In our simple example the points are displaced by a vector-valued fractal noise function, and then a scalar-valued noise function is used to modulate their color. From this simple foundation, point instantiation primitives used in production typically add large numbers of control parameters and noise functions, each responsible for manipulating the final appearance in a different way.

The following code implements a simple sphere-based instancing algorithm

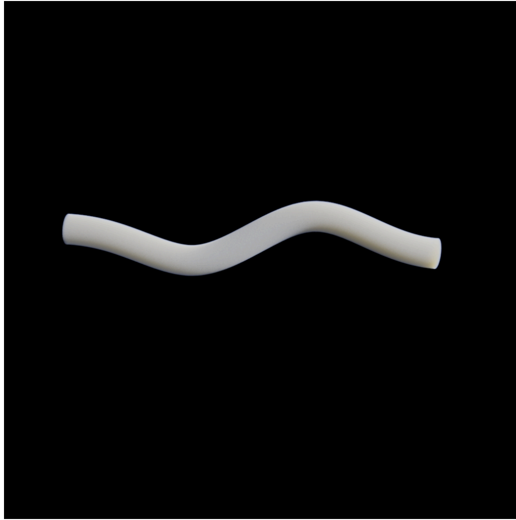
```
void instanceSphere(const SpherePrim &prim, PointCloud &output)
{
  Rand32 rng(prim.id);
  for (int i = 0; i < prim.numInstances; i++) {
     // Instantiate a point
     V3f lsP, wsP;
     if (prim.doShellOnly) {
        lsP = hollowSphereRand(rng);
     } else {
        lsP = solidSphereRand(rng);
     }
     // Displace the instanced point by some noise function
     lsP += fbm(lsP, prim.dispNoisePeriod, prim.dispNoiseOctaves) * prim.dispNoiseAmplitude;
     // Transform point to world space
     prim.localToWorld(lsP, wsP);
     // Modulate density by second noise function
     float density = fbm(lsP, prim.noisePeriod, prim.noiseOctaves) * prim.noiseAmplitude;
     // Add point to output collection
     output.addPoint(wsP, density);
  }
}
```

Note that in this example we accumulate all points in a collection which gets sent to the rasterizer once instantiation is complete. It is also possible to directly rasterize each point as it is instanced.

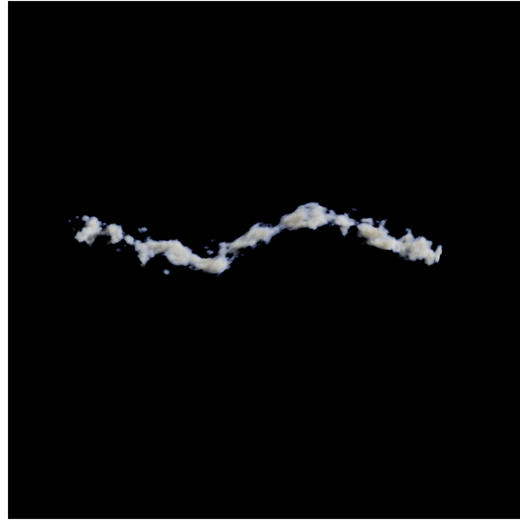
2.6.2. Curve-based point instantiation

Sphere-based primitives are particularly convenient because of their simple parameterization, and because of how easy it is to define a coordinate space that travels with the primitive. As we will see, curves and surfaces can also be used, but their coordinate spaces are a little more involved to define.

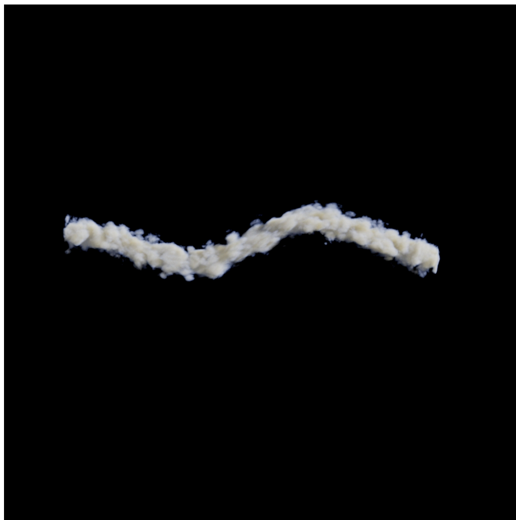
The following images show various noise techniques applied to a curve primitive. Each primitive uses roughly 40 million points. (Images are of reduced size but at around 350 dpi, zoom in for a better view.)



Constant-radius curve primitive



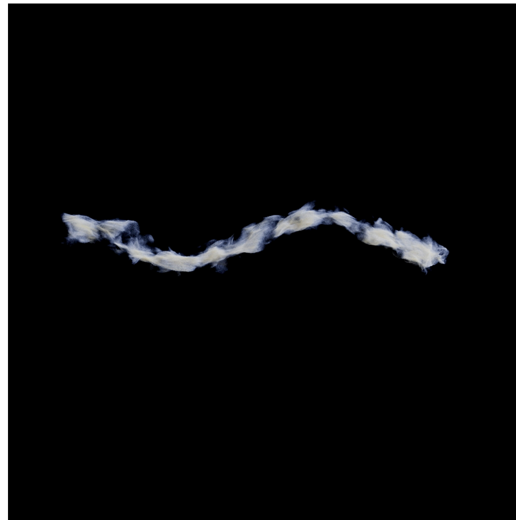
Perlin noise modulating density



Absolute-valued perlin noise modulating density

Displacing points along normal using absolute perlin noise (pyroclastic)

Points displaced by vector-valued perlin noise



Density first modulated by scalar-valued perlin noise, then displaced using vector-valued perlin noise

In the example above the point distribution is completely random, which can lead to bunching up of point locations and lead to a grainy look in the final result. Depending on the desired look, this may or may not be a good thing. If a smooth look is the goal, it may be better to use a blue noise/poisson distribution of points.

The following code shows a simple curve-based instancing algorithm.

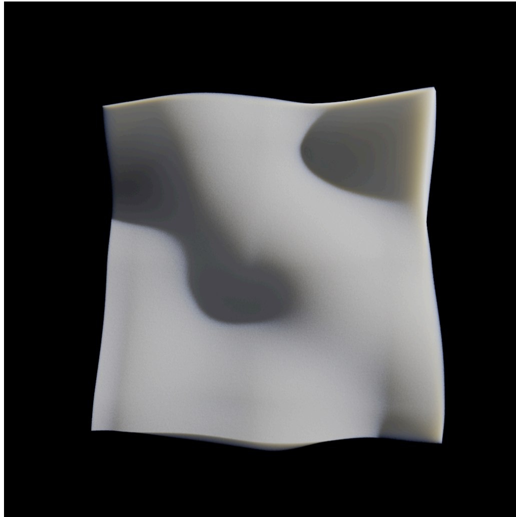
```
void instanceCurve(const CurvePrim &prim, PointCloud &output)
{
  Rand32 rng(prim.id);
  for (int iSeg = 0; iSeg < prim.numSegments; iSeg++) {
    for (int i = 0; i < prim.numInstances; i++) {
      // Pick a random position in the current line segment.
      // This will be our coordinate along the T basis
      float u = rng.nextf();
      // Pick a random 2d position on a [-1,1][-1.1] disc.
      // This will be the coordinate in the (N, NxT) plane.
      V2f st = sampleDisc(rng);
      // Find T and N basis vectors at current location and calculate third basis
      V3f T = prim.interpolateT(iSeg, u);
      V3f N = prim.interpolateN(iSeg, u);
      V3f NxT = cross(N, T);
      // Transform point on curve to world space
      V3f wsP = prim.interpolateP(iSeg, u);
      // Displace along N and NxT by st vector, scaled by radius
      float radius = prim.interpolateRadius(iSeg, u);
      wsP += NxT * st[0] * radius;
      wsP += N * st[1] * radius;
      // Pyroclastic noise is created by displacing the point further
      // along the radial direction
      V3f nsP(st[0], st[1], u);
      V3f wsRadial = normalize(NxT * st[0] + N * st[1]);
      float displ = fbm(nsP, prim.dispNoisePeriod, prim.dispNoiseOctaves) *
```

```
                       prim.dispNoiseAmplitude;
        wsP += wsRadial * displ;
        // Modulate density by second noise function
        float density = fbm(nsP, prim.noisePeriod, prim.noiseOctaves) * prim.noiseAmplitude;
        // Add point to output collection
        if (density > 0.0f) {
           output.addPoint(wsP, density);
        }
  }
}
```

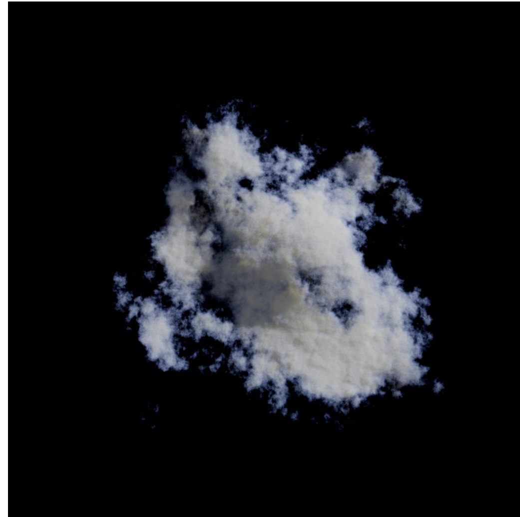
The `CurvePrim::interpolate*()` functions are used to calculate the values in-between the control vertices of the curve, and they may use any method for this. If curves are finely tessellated then a piecewise linear function may be enough, although it is more common to use a spline function.

2.6.3. Surface-based point instantiation

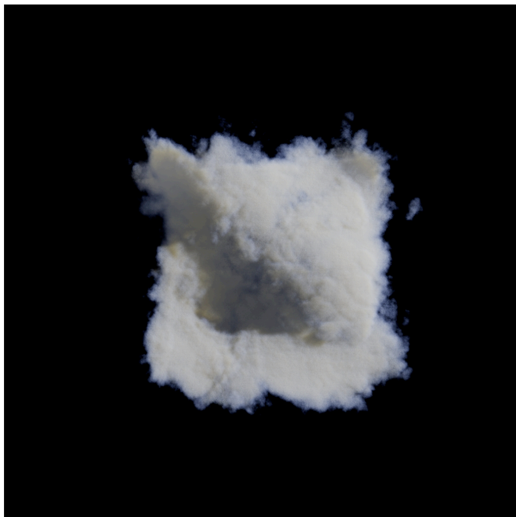
The following images show various noise techniques applied to a surface primitive. Each primitive uses roughly 400 million instanced points in order to achieve a smooth result at 1024x1024 pixels. A frustum-shaped voxel buffer of resolution ~1200x1200x250 was used. It should be noted that primitives that modulate their density using noise are slightly "wasteful", because we need to instantiate a point, calculate its noise space position and evaluate the full fractal noise function before knowing whether to cull it due to zero density.



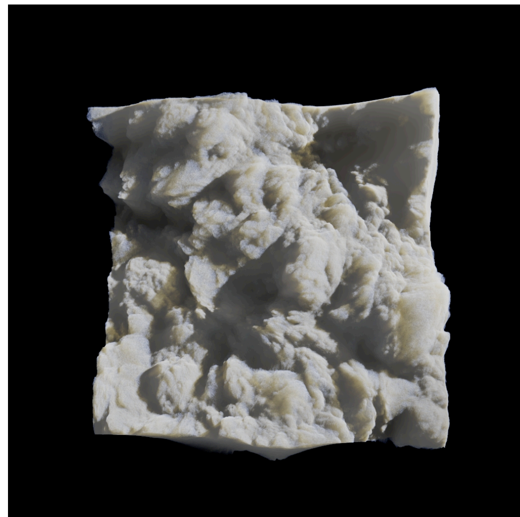
Constant-thickness surface primitive



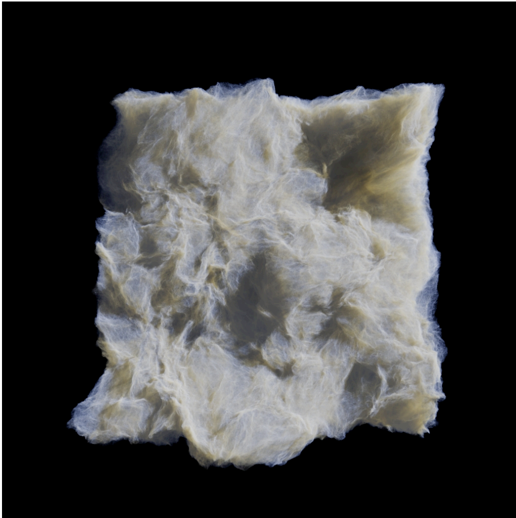
Perlin noise modulating density



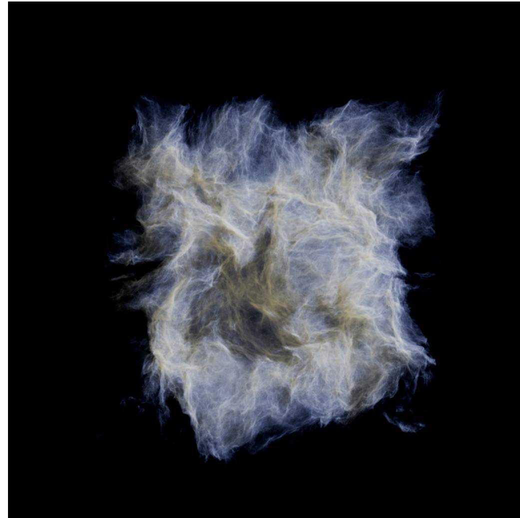
Absolute-valued perlin noise modulating density



Displacing points along normal using absolute perlin noise (pyroclastic)



Points displaced by vector-valued perlin noise



Density first modulated by scalar-valued absolute perlin noise, then displaced using vector-valued perlin noise

The following code shows a simple surface-based instancing algorithm.

```
void instanceSurface(const SurfacePrim &prim, PointCloud &output)
{
  Rand32 rng(prim.id);
  for (int iSeg = 0; iSeg < prim.numSegments; iSeg++) {
    for (int i = 0; i < prim.numInstances; i++) {
      // Pick a random position on the current surface patch.
      // These will be our coordinates along the dP/du, dP/dv bases
      V2f st;
      st[0] = rng.nextf();
      st[1] = rng.nextf();
      // Pick a random 1d position to determine offset along N basis
      float u = rng.nextf();
      // Find T and N basis vectors at current location and calculate third basis
      V3f N = prim.interpolateN(iSeg, st);
      // Transform point on surface to world space
      V3f wsP = prim.interpolateP(iSeg, st);
      // Displace along N vector, scaled by radius
      float radius = prim.interpolateRadius(iSeg, st);
      wsP += N * u * radius;
      // Pyroclastic noise is created by displacing the point further
      // along the radial direction
      V3f nsP(st[0], st[1], u);
      float displ = fbm(nsP, prim.dispNoisePeriod, prim.dispNoiseOctaves) *
                    prim.dispNoiseAmplitude;
      wsP += N * displ;
      // Modulate density by second noise function
      float density = fbm(nsP, prim.noisePeriod, prim.noiseOctaves) * prim.noiseAmplitude;
      // Add point to output collection
      if (density > 0.0f) {
        output.addPoint(wsP, density);
```

```
            }
        }
    }
```

Just as curves can be any number of segments, each surface primitive can be made up of an arbitrary number of pieces. Traditionally, each piece is a quad-connected set of vertices, which can be used to create either a regular polygon mesh or a parametric surface. Either way, the SurfacePrim::interpolate*() functions need to evaluate quickly for any coordinate in the patch based on its *st* coordinate.

2.7. Modeling with level sets

Level sets are a technique for tracking interfaces. From its introduction to the computer graphics community in the late 1990s the level set method has quickly become one of the workhorses of the industry. Level sets are useful for collision detection, fluid simulation, and rendering. They are also featured in popular third party applications, such as Houdini and Real Flow.

Typically interfaces in graphics, such as the model of a character, are represented explicitly with polygon meshes or NURBS, for example. There is a rich history of tools and techniques for dealing with such explicit representations. However it is very difficult to implement operations like unions or differences with explicit representations. Additionally, topological changes due to animation need to be handled in special ways which are not robust. The level set method works by representing an orientable manifold surface as a function which tracks the signed distance to the nearest point on the interface from any point in space. In the general case the level set method defines the evolution of the level curve in the normal direction at a certain speed. Most of the time we are interested in the Euclidean distance, which leads to a special case of level sets called signed distance fields (SDF).

Level sets are typically stored in the same volumetric data structures we have been discussing. Each voxel stores the level set value, ϕ, at that location. This is the distance to the nearest point on the interface. As the name signed distance field suggests these values are oriented based on whether the location is inside or outside of an object. For our discussion we assume that level set values outside the object are positive, $\phi > 0$, and negative, $\phi < 0$, inside the object. The zero level, $\phi = 0$, represents the exact interface.

2.7.1. Constructive Solid Geometry Operations

We can extend our voxel buffer machinery with level set specific methods to obtain some really powerful features. The most trivial ones to implement are CSG operations. This pseudocode for a union operation demonstrates one of the reasons behind the viral popularity of level sets, they are extremely trivial to implement.

```
/*! Performs a CSG union between level sets A and B, and
stores the results in A. Assumes A and B have the same transform .*/
void union(VoxelBuffer& A, const VoxelBuffer& B) {
  BBox dims = lightBuffer .dims();
  for (int k = dims.min.z; k <= dims.max.z; ++k) {
     for (int j = dims.min.y; j <= dims.max.y; ++j) {
        for (int i = dims.min.x; i <= dims.max.x; ++i) {
           A = std::min(A.value(i,j,k), B.value(i,j,k));
        }
     }
  }
}
```

The difference between two buffers A and B can be calculated by computing the maximum value at each voxel between value in A and the negated value in B. Intersections between two buffers are computed by taking the maximum value at each voxel. In user interface terms the intersection corresponds to a copy operation, the union is a paste operation, and the difference is a cut operation.

Operation	Implementation
Union	min(A, B)
Intersection	max(A, B)
Difference	max(A, -B)

2.7.2. Rendering Level sets

Level sets can be rendered as a solid object, or as a volumetric element. The simplest volumetric treatment assigns a constant density value to each inside voxel, $\phi \leq 0$. In order to avoid aliasing artifacts a roll-off can be applied to the voxels in a band near the surface.

```
phi = levelSet.value( i,j,k );
if ((phi <= 0) && (phi >= -bandwidth))
  density = defaultDensity * smoothstep(-phi, 0, bandwidth);
```

Surface rendering of level sets can be performed directly, or it can be converted back to an explicit mesh. Ray tracing level sets is very efficient because the level set values can be used to accelerate the ray intersection tests. We evaluate the level set value at the start position of the ray. This value tells us how far along the ray we have to advance before we are at the surface. We then evaluate the level set at this new location, and iterate a fixed number of times to find an accurate intersection point if one exists. We can convert level sets to polygon meshes using the popular marching cubes algorithm.

In order to be representable as a level set, an object must have a clearly defined inside and outside. Museth et al. provide discussion of conversion techniques in their paper *Algorithms for Interactive Editing of Level Set Models*. We recommend the excellent text *Level Set Methods and Dynamic Implicit Surfaces* by Stanley Osher and Ronald Fedkiw for more details on level sets and the useful things you can do with them.

2.8. Motion blur

So far we have only considered primitives that are stationary. Of course, to create a production-quality volume renderer we need to consider primitives in motion as well. When it comes to surface rendering, micropolygon-based renderers record the motion per-fragment and assigns a time to each pixel sample. A raytracing-based renderer also assigns a time to each ray, and displaces the contents of the scene so that the ray sees the appropriate state.

In volume rendering, true motion blur is often too expensive to calculate. In some cases, such as eulerian motion blur of procedural fields and simulation data, the motion blur calculation can be done correctly. However, when considering thousands or millions of volumetric primitives we simply cannot produce correct motion blur – in fact, the use of rasterization into voxel buffers prevents it.

The most common solution to producing *almost correct* motion blur in voxel buffers is to smear each sample along its motion vector. Smearing has the following properties: It distributes the value evenly across all the voxels it touches, and the sum of all values written to those voxels is equal to the original value.

2.8.1. Line drawing

The first approach we can use to smearing the sample is to employ standard line-drawing in 3D. In order for the motion blur to look smooth we need to antialias the line. Fortunately, algorithms for drawing an antialiased line are commonplace in computer graphics, and we refer the reader to the standard literature for implementation details.

2.8.2. Splat-based smearing

The second approach is to draw multiple trilinear splats to make up the line. This has the benefit of being easier to implement, and as we'll see below it also introduces the opportunity to control the quality of the smear.

```
void Rasterizer::trilinearSmear(const V3f &vsStartP, const V3f &vsEndP,
                                float value)
{
  V3f vsMotion = vsEndP - vsStartP;
  int numSplats = ceil(vsMotion.length()) + 1;
  float valuePerSplat = value / static_cast<float>(numSplats);
  for (int i = 0; i < numSplats; ++i) {
     float fractionTraveled =
        static_cast<float>(i) / static_cast<float>(numSplats - 1);
     trilinearSplat(vsStartP + fractionTraveled * vsMotion, valuePerSplat);
  }
}
```

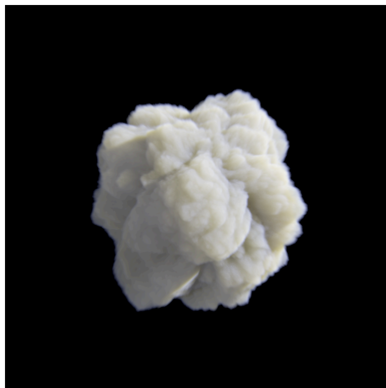
By calculating the fraction of the distance travelled instead of incrementing the position at each step of the loop we avoid accumulation of errors in the splat positions.

Using splats has another interesting possibility – undersampling. Though we know how many samples we *should* use, we could potentially use fewer to speed up the rasterization. Just as with nearest-neighbor splatting, when a lot of primitives are involved, their random distribution tends to hide undersampling and noise artifacts. Thus, we may add a scaling factor to the function, which lets the user control how many samples should be used to draw each line.

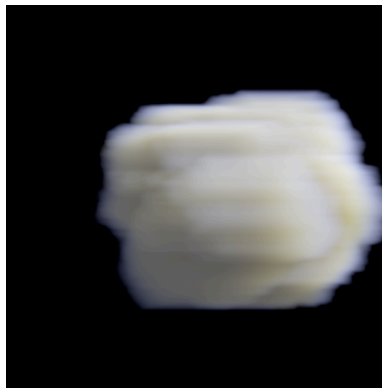
```
void Rasterizer:: trilinearSmear(const V3f &vsStartP, const V3f &vsEndP,
                                 float value, float samplingFactor)
{
  V3f vsMotion = vsEndP - vsStartP;
  int numSplats = ceil(vsMotion.length() * samplingFactor) + 1;
  // ...
}
```

2.8.3. Smearing problems

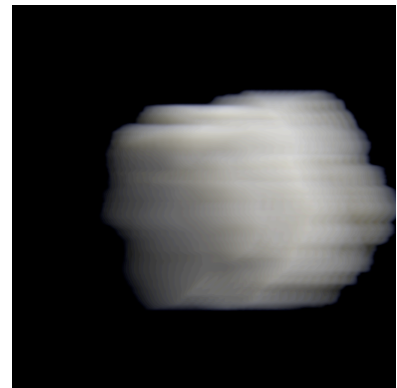
Of course, smearing the samples is technically incorrect. In an abstract sense, we are folding the temporal domain into the spatial domain, and in doing so we lose all information about when a given primitives occupies a given position in space. This problem becomes apparent in the loss of lighting detail during subsequent rendering. A sharp feature that is smeared will no longer shade the same as when stationary, and the result tends to look artificially soft. However, this downside is usually acceptable when considering the alternative of calculating full deformation blur during rendering.



Stationary primitive



Smeared primitive produces incorrect lighting



Average of multiple frames shows a more correct result

We find another problem if the camera is moving at the same speed as the primitives being rasterized, any motion should be cancelled out and the result look sharp. But because the motion blur is baked into the voxel buffer this is not possible.

2.8.4. Post-rasterization smearing

An alternative to smearing each individual sample is to use a separate buffer to accumulate a velocity vector for each voxel. Once all rasterization and/or splatting is done, the velocity is used to smear the entire buffer in a single step. This is often faster than smearing each sample as it is written to the buffer, but it suffers from a potentially large problem, depending on the input geometry. The problem occurs when the input primitives overlap and have drastically different motion vectors. In this case it becomes impossible to calculate a valid direction to smear in. It is possible to resort to keeping track of the average motion vector in each voxel that has overlapping primitives, but this can cause visual artifacts in the final render.

A variant of this method is to simply retain both the density buffer and the velocity buffer and calculate the motion blur during rendering. This still suffers from the problem with overlapping primitives but does avoid the problem of reduced shading detail in motion blurred areas. Microvoxel-based volume renderers lend themselves well to this approach.

2.9. High resolution voxel buffers

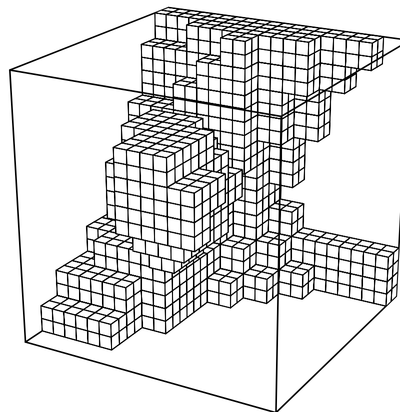
Up until now we've expected that voxels exist everywhere within the domain of the voxel buffer. And in this domain each voxel is the same size. This is fine when the volumetric element that is being modeled is small in screen-space, but if we need to get close to the element, or if the element extends across the entire visible frame, the resolution required in order to provide sharp details will likely range in the thousands along each axis.

Two approaches are most common in visual effects production when trying to solve this problem. The first addresses the problem of unused voxels occupying memory, and the second amounts to adapting the voxel size so that voxels close to camera are small and those far away are large.

2.9.1. Sparse data structures

Any time a dense voxel buffer stores a zero density it is effectively wasting memory. This happens because dense buffers blindly allocate storage for every voxel in its domain without considering what areas will be populated. Since most volumetric elements tend to have some sort of connectedness and generally don't occupy the entire domain of the voxel buffer, finding a data structure that allocates memory more intelligently would help improve memory use.

One of the simplest such structures is the *block-based sparse* buffer. (What is referred to here as *blocks* is sometimes also called *tiles*.) It can be thought of as a two-level-deep hierarchical data structure where the domain of the buffer is subdivided into coarse *blocks*, and where a *block* can contain either a single value (usually zero, though for storage of level sets it can be useful to assign a different value), or an N^3 array of voxels, representing the actual voxel data in the block.



A sparsely allocated voxel buffer (with unallocated blocks hidden)

The illustration above shows a sparsely allocated voxel buffer with blocks of size 2^3 (for purposes of clarity). A more common block size would be between 8^3 and 32^3.

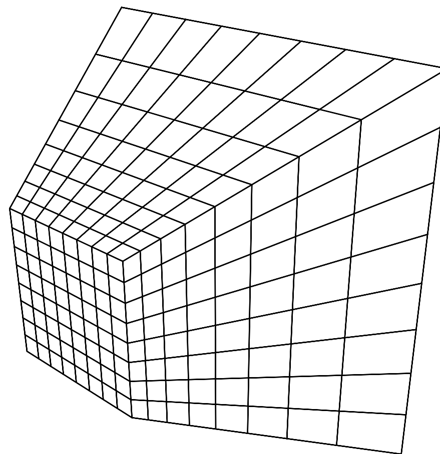
A block remains unallocated (storing just a single value) until the first write-access to one of its voxels. Once the first write happens, all of the block's voxels are allocated. Each block is thus effectively its own, small, dense buffer once allocated.

Using a fixed-depth hierarchy means that voxel read and write access is $O(k)$, or constant time, though with a larger k than an ordinary dense buffer. The allocation that happens on the first write access is amortized over all subsequent accesses.

Field3D provides an implementation of this type of data structure in its `SparseField` class.

2.9.2. Frustum-shaped voxel buffers

The second approach is to adapt the voxel size to account for the fact that objects far away from camera require less detail than those close up. The most common way of accomplishing this is to use frustum-shaped voxel buffers, usually referred to simply as *frustum buffers*. Frustum buffers are tied to a camera (usually the main shot camera), and follow any animation applied to the camera. When seen from the side (below), each voxel looks stretched and sheared, but when viewed from the camera, each row of voxels lines up perfectly with the projection of a pixel into the scene.



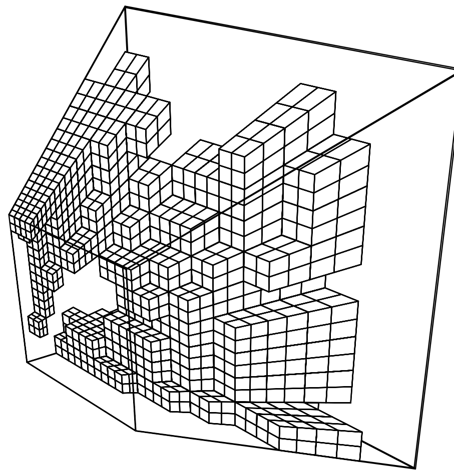
A frustum-shaped voxel buffer

The resolution of the above buffer is 8x8 in the *XY* plane, with 8 *slices* along the *Z* axis. The *XY* resolution is normally locked to the resolution of the camera, times some multiplier, and the number of *Z* slices is left as a user parameter. The *Z* resolution is normally much lower than either of the *XY* axes, usually on the order of a few hundred.

The transform from from world space to voxel space can be implemented as:

```
void FrustumMapping::worldToVoxel(const V3f &wsP, V3f &vsP)
{
  // The camera's 0..1 NDC space matches the local space of the voxels
  V3f nsP;
  m_camera->worldToNdc(wsP, nsP);
  localToVoxel(nsP, vsP);
}
```

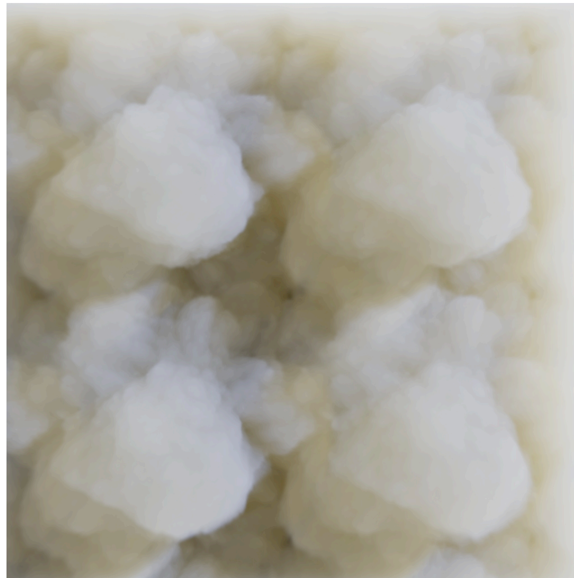
To further optimize memory use we can combine sparse buffers and frustum buffers. Since the coordinate transform is independent of the data structure used for voxel storage, this is straightforward. This approach combines the benefit of finer detail close to camera with the empty space optimization of the sparse buffer.



A sparsely allocated frustum-shaped voxel buffer

2.9.3. Problems with frustum buffers

Frustum buffers are not without drawbacks. *Light leaks* may be visible along the edges of the frame during rendering, since no density is available outside the view of the camera to stop light from penetrating into the buffer. This can usually be addressed by padding the bounds of the frustum buffer, so that it extends outside the view of the camera, though in extreme cases the amount of padding needed negates the performance gains and reduced memory usage offered by the frustum buffer.



Light leaks along right edge of frustum

Another problem occurs when primitives in the scene are widely distributed along the camera's depth axis. This forces each *Z* slice to become excessively deep, which manifests itself as aliasing artifacts, called *slicing artifacts*. These are visible as a posterization-like look, with poor lighting detail. The artifacts can be reduced by careful antialiasing of the volumetric primitives during the rasterization phase, but to completely avoid them an increase in *Z* resolution is required.

Sufficient detail along z axis (150 slices)

Slicing artifacts due to low z resolution (25 slices)

Frustum buffers are also more prone to aliasing due to the noise functions used and if an insufficient number of *z* slices is used the effect can be very visible. The example below shows a render using the same 25 slices as above, but with noise antialiasing disabled:

Aliasing artifacts due to excessive high-frequency detail in noise function

It can also be difficult to correctly capture moving primitives at the edge of the frustum. Primitives that motion blur into the buffer need to be considered even if only a few samples of their smeared contribution fall into the buffer, otherwise leaks will occur, similar to the light leaks discussed earlier. In certain cases it may also be difficult and/or expensive to determine if a primitive should be included, for example if a primitives both enters and exits the frustum buffer in a single time frame.

3. Volume rendering

Now that we have the foundations in place for creating the data for volumetric effects we move on to looking at how to render the data. Volume rendering is about the mathematics of how light behaves in a *participating medium*, where the medium may be anything from smoke or water vapor to clouds and atmospheric haze. The equations that govern volume rendering are applicable across a large range of media, and we can use the same approach to render almost any kind of volumetric effect.

Volume rendering for visual effects production also extends past the physical in order to achieve certain desired looks, and to better integrate with the rest of the production pipeline. It often becomes necessary to bend the rules for what *should* happen, and the constant challenge is to find methods for doing so that are controllable yet plausible.

This chapter will describe the fundamental components required for creating a production-grade volume renderer, and will cover basic scattering theory and the raymarching approach to solving the scattering problem. It will also cover efficiency and optimization, integration issues and the challenges associated with motion blur. Most topics will be familiar to those familiar with volume rendering in general, but in this course we aim to describe which specific techniques are used in day-to-day production environments, and how those techniques are integrated to create a practical and functioning system. There are of course other approaches to volume rendering than those described here, and several different ones are actively used in the visual effects community as well, but in these course notes we will focus on the raymarching-based approach.

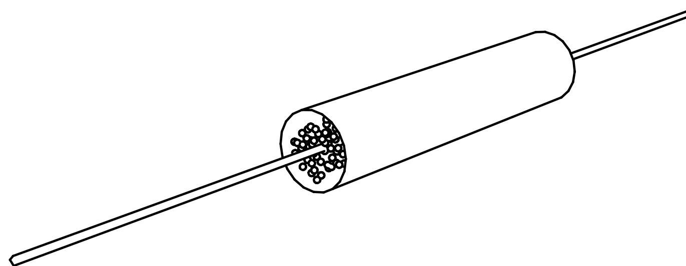
3.1. Lighting theory

When designing a volumetric effects we want to describe both its shape and motion (the *modeling* part of volume rendering), as well as its appearance (the real *rendering* part of volume rendering). When describing the appearance it is useful to break it down into some fundamental *characteristics*, which can then be combined in various relationships to achieve a wide variety of looks.

As it turns out, there are only three fundamental characteristics needed to describe any given volumetric element. Each describes a different physical process.

- *Absorption* (sometimes also called *extinction*) is the loss of radiant energy along a ray of light due to energy being converted into some other form, such as heat. Black smoke is a good example of an *absorbing* medium.
- *Emission* adds radiant energy and happens where the medium itself is luminous. Flames and fire are examples of highly *emissive* media.
- *Scattering* describes how likely a medium is to cause a ray of light to collide with a particle and change its direction. As an example, water vapor is an almost completely *scattering* medium. There are two types of scattering to consider. First, light traveling from a distant object towards the camera has a probability of being reflected off to another direction, which is called *out-scattering*. Another possible outcome is that a ray of light traveling in some random direction gets reflected into the view ray of the camera, which is called *in-scattering*. Each of these probabilities is equally likely to occur, since the light being reflected has no idea of the position and orientation of any observers.

Each of these characteristics can be isolated and discussed in terms of how they affect a ray of light traveling through space. When deriving the mathematical model for how light is affected by participating media it is useful to consider a differential cylinder: a cylinder, infinitely thin, of unit length, through which a ray of light passes.



A differential cylinder filled with a participating medium

We will use the following notation in the equations that follow:

- p – the position of the cylinder, and the interaction location
- $\boldsymbol{\omega}$ – the direction of the ray
- L – the radiance quantity
- L_i – the incoming radiance, before any interaction with the medium
- L_o – the outgoing radiance, after the interaction with the medium

3.1.1. Modeling absorption

In discussing the light scattering properties of volumes we will draw some parallels to surfaces and their BRDFs. A dark surface is dark because of its low reflectivity. The laws of physics dictate that whatever incident radiant energy that is not reflected away from the surface must be absorbed – energy does not disappear, it only changes form, in this case into heat. Volumes share this property, but instead of describing the fraction of light absorbed after interaction with a surface the absorption coefficient determines how likely it is that a ray of light is absorbed as it travels through a volume.

The unit of the absorption coefficient σ_a is a reciprocal distance (i.e. m^{-1}), which essentially describes the probability that a ray of light is absorbed as it travels through one length unit of the medium. Being a reciprocal means that it can assume any positive value – it can be infinitely large.

Mathematically formulated, the absorption interaction can be described as:

$$L_o = L_i + dL$$
$$dL = -\sigma_a L_i \, dt$$

3.1.2. Modeling emission

When the medium emits light, for example as the result of some chemical reaction, or due to the thermal properties of the medium, it adds to the radiance of a ray passing through it.

The emissive term $L_e(p, \boldsymbol{\omega})$ is a measurement of radiance, which describes the amount emitted in direction ω along a one unit long section of a ray.

The mathematic formulation for emission is:

$$L_o = L_i + L_e$$
$$L_e = \sigma_e \, dt$$

3.1.3. Modeling scattering

The scattering property describes the likelihood that a ray of light traveling through the medium will bounce and reflect to another direction. As mentioned before, this interaction accounts for both in-scattering and out-scattering, although when calculating lighting effects in a volume, the effect of out-

scattering is usually folded into the absorption calculation, since the net result is identical. We refer to the *extinction term* when considering absorption and out-scattering together.

The unit of the scattering coefficient σ_s is (just as absorption) a reciprocal distance, which describes the probability that a ray of light is scattered as it travels through one length unit of the medium. This means that a ray traveling through a medium with $\sigma_s = 0.1$ will travel on average a distance of $0.1^{-1} = 10$ units before a scattering event occurs. This distance can also be referred to as the *scattering length.*

Given a light source *S*, whose function $S(p, \boldsymbol{\omega}')$ describes the quantity of light arriving at point *p* from direction ω', we can formulate the scattering interaction as:

$$Lo = L_i + dL_{in} + dL_{out}$$

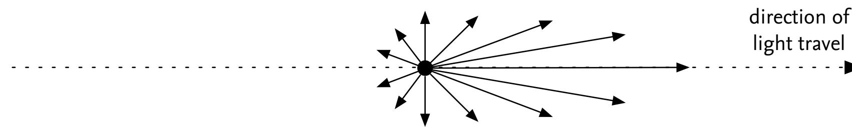
$$dL_{in} = \sigma_s\, p(\boldsymbol{\omega}, \boldsymbol{\omega}')\, S(p, \boldsymbol{\omega}')$$

$$dL_{out} = -\sigma_s\, L_i\, dt$$

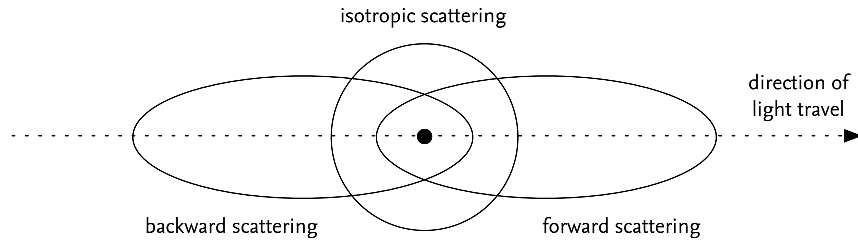
The function $p(\boldsymbol{\omega}, \boldsymbol{\omega}')$ is called the phase function, and the next section will detail what it is and how it affects the scattering interaction.

3.1.4. Phase functions

The property of a volume that relates closest to a surface BRDF function is the *phase function.* A BRDF defines how much of light hitting a surface while traveling in direction $\boldsymbol{\omega}$ will scatter to direction $\boldsymbol{\omega}'$, and similarly the phase function determines how much light traveling through a medium in direction $\boldsymbol{\omega}$ will, upon scattering, reflect to direction $\boldsymbol{\omega}'$, i.e. $probability = p(\boldsymbol{\omega}, \boldsymbol{\omega}')$. Phase functions (at least the ones relevant to our purposes) have a few important properties. First, they are isotropic, meaning that the function is rotationally invariant, and only the relative angles between ω and ω' need to be considered, thus we can write $p(\boldsymbol{\omega}, \boldsymbol{\omega}') = p(\theta)$. Second, they are reciprocal, so $p(\boldsymbol{\omega}, \boldsymbol{\omega}') = p(\boldsymbol{\omega}', \boldsymbol{\omega})$. Third, they are normalized such that integrating across all angles for ω while holding $\boldsymbol{\omega}'$ constant gives exactly 1.



The length of each vector illustrates the probability of scattering in that direction



Isotropic and anisotropic phase functions

Phase functions come in two flavors, *isotropic* and *anisotropic*. An isotropic (not to be confused with isotropic in the rotationally invariant sense, as in the previous paragraph) phase function scatters lights equally in all directions. Anisotropic phase functions are biased either forward or backward, as seen from the direction of light travel before the scattering event.

Isotropic phase functions are perfectly sufficient when rendering low-albedo media, such as ash clouds, dust etc., but for media such as clouds and atmospheres, anisotropy is an important element to include in lighting calculations. Anisotropic behavior in participating media can be thought of as the parallel to specular BRDFs, and isotropic to diffuse/lambertian BRDFs. In everyday life, anisotropy is responsible for the *silver lining* in clouds, where the edge of a cloud becomes increasingly bright as the sun reaches a grazing angle.

Phase functions are well researched, and the two most common ones are the *Rayleigh* model (which describes atmospheric scattering, the interaction of light with particles the size of molecules), and the *Mie* model (which is more general and can handle much larger particle sizes, for example water vapor and droplets suspended in the atmosphere). In production rendering we often use a few other, simpler models, since Rayleigh and particularly Mie are expensive to evaluate. *Henyey-Greenstein* is a simple model that can handle both isotropic and anisotropic media, and a similar, but cheaper one is the *Schlick* phase function. These functions can all be found in the standard literature, but one especially good overview of phase functions in the context of volume rendering can be found in the book *Physically Based Rendering*[6].

[6] *Matt Pharr & Greg Humphreys – Physically Based Rendering (Morgan Kaufmann publ.)*

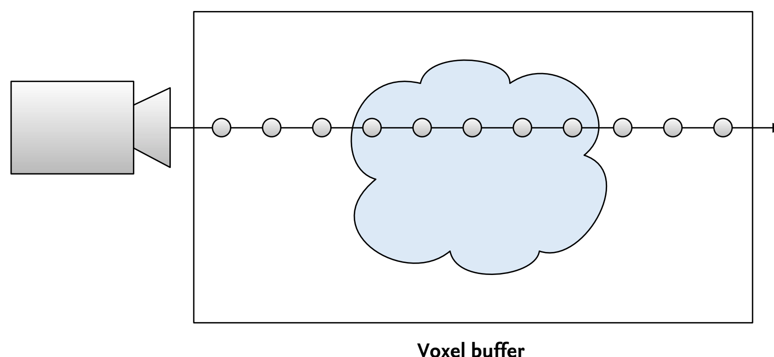
3.2. Raymarching

Transmittance is the fraction of light which passes through a volumetric sample after being impeded by the material in the sample. It is the ratio of the outgoing light to the incoming light. Beer-Lambert's Law relates the absorption capacity of a material to the transmittance. We write this equation as: $T = e^{-\sigma\rho l}$.

- T is transmittance
- σ is the coefficient of absorption
- ρ is amount of absorbing material in the sample
- l is the length of the path through the sample.

Opacity is the fraction of light which is absorbed by the sample. $\alpha = 1 - T$. With respect to rendering, opacity is accumulated to compute the alpha channel.

As in any other kind of rendering we want to figure out the light that gets to the camera. Real participating media attenuates the light, modulates the frequency, and alters the path of light. In other words a lot of complicated physical processes which we are going to simplify. This simplification was introduced by Kajiya and Von Herzen in 1984 in their seminal SIGGRAPH paper *Ray Tracing Volume Densities*. The process is to trace a ray from the cam- era through the volume, and compute the illumination in small segments along the ray. The illumination at each segment must be attenuated by the density between the camera and the segment. The ray is traced until the ray exits the volume, or it can see no further into the volume. The illumination calculation at each segment requires determining how much light is reaching that segment. This implies that we must perform another raymarch from each light to the segment. We must also account for material properties such as albedo, and the light absorption capacity of the material. This is a very simplified process, and does not account for some common effects such as volumes that emit light, such as fire, or scattering effects. But this is a good starting point.



In order to compute the pixel value for a ray, $\boldsymbol{x}$, we initialize color, C and opacity, α, to zero. Transmittance, T, is initialized to 1. We then intersect the ray against the voxel buffer to determine where we need to perform the integration. Finally we integrate this interval by sampling along $\boldsymbol{x}$ in steps of length $\triangle x$. The contribution of a sample i is:

$$
\begin{aligned}
T_i &= e^{-\sigma\rho\triangle x} \\
T_{total} &= T_{old} * T_i \\
C_i &= T_{total}\, L(\boldsymbol{x}_i) c(\boldsymbol{x}_i) \rho(\boldsymbol{x}_i) \triangle x \\
C_{total} &= C_{old} + C_i \\
\alpha_{total} &= \alpha_{old} + (1 - T_i) * (1 - \alpha_{old})
\end{aligned}
$$

where

T_i : *Transmittance at* $\boldsymbol{x}_i$
T_{total} : *Total transmittance from the start of the ray to* $\boldsymbol{x}_i$
$L(\boldsymbol{x}_i)$: *Incident lighting at sample location*
$c(\boldsymbol{x}_i)$: *Color of the material at sample location*
$\rho(\boldsymbol{x}_i)$: *Density at sample location*
$\triangle x$: *Ray step length*
α : *Opacity*

In order to compute the incident lighting we need to compute the transmittance from the light to the sample position. This requires us to perform another ray march between the sample position and the light. Employing the same mechanism used above:

$$
\begin{aligned}
T_i &= e^{-\sigma\rho\triangle x} \\
T_{total} &= T_{old} * T_i \\
L &= T_{total} * C_{light} * P(\theta)
\end{aligned}
$$

where

C_{light} : *Light color*
$P(\theta)$: *Phase function*

The following method is a simple integration module. Note that no lighting calculations are performed.

```
float Raymarcher::integrate(const V3f& pos, const V3f& dir, const float absorption) const
{
    // Determine intersection with the buffer
    float t0, t1;
    if (false ==  m_buffer->intersect(pos, dir, t0, t1))
        return 1.0f;

    // Calculate number of integration steps
    const int numsteps = int(ceil(t1 - t0) / m_stepsize);

    // Calculate step size
    const float ds = (t1 - t0) / numsteps;
    V3d stepdir(dir);
    stepdir *= ds;

    V3d raypos(pos);
    raypos += stepdir;

    const float rhomult = -absorption * ds;

    // Transmittance
    float T = 1.f;

    for (int step = 0; step < numsteps; ++step) {
        float rho = m_buffer.trilinearInterpolation(raypos);
        T *= std::exp(rhomult * rho);
        if (T < 1e-8)
            break;
        raypos += stepdir;
    }

    return T;
}
```

In order to accelerate the lighting calculation we precompute the transmittance values from each light through the volume. During rendering we interpolate this data set to determine lighting at the sample point. This is discussed in further detail in the section on pre-computed lighting.

It is often necessary to model volumetric materials which are emitting light, such as fire. Such materials are like diffuse densities, except that we can consider them to have intrinsic lighting. The rendering procedure involves augmenting the lighting function, $L(\boldsymbol{x}_i)$, with an additional source for the emissive light.

3.2.1. Artistic controls

This simple illumination model offers a lot of room for art direction. The constants which appear in the expressions above are the simplest of these controls. While these parameters are rooted in physical accuracy we need not keep to this constraint. The absorption coefficient, σ, can be different between the

lighting and rendering calculations. It can also be varied for each color channel for a complex lighting effect.

We have found it useful to provide a multiplier for the density, ρ, parameter. This provides the user a simple way to manipulate the source data at render time. The step length, $\triangle x$, parameter is the simplest way to trade off between quality and rendering speed. Large step sizes means the ray integration has to consider a lot fewer samples, and is quicker to compute. Consequently it also fails capture all of the detail available in the voxel buffer. The concept of frustum buffers is based on the observation that we may not require high quality integration far away from the source of the ray. Step length is typically expressed in world space units.

3.2.2. Implementation

The modular nature of the ray marching algorithm hints at how we can implement a generic volumetric shading architecture. In a simple scheme we can have a main renderer, material shader plugins, and light shader plugins. The main renderer is responsible for rendering a given voxel buffer. The user specifies which material shader to apply to that buffer, as well as the lights. The renderer invokes the ray marching, and performs the integration. At each sample along the ray the material shader is called with the shading position, voxel buffer, and information about the lights. The material shader is then responsible for returning a shaded color and transmittance for a single sample. This requires that the shader loop over all the lights and invoke the light shaders. In the case of using precomputed lighting the method to obtain lighting from the light shaders can simply return the appropriate value for the given sample position.

3.3. Pre-computed lighting

Lighting calculations are one of the computationally heaviest steps of the raymarching algorithm. Since at each step we need to sample incoming light, the number of samples needed quickly approaches the billions. A completely unbiased approach to lighting would, for each raymarch step (of which there are at least 100 per pixel) along a primary ray, perform another raymarch toward each light in the scene to determine the amount of occlusion (multiplying the number of samples by another 100 or so). A brute force implementation of this scheme is incredibly slow, and as such it is mostly useful as a way of verifying the result of other techniques. Production renderers usually use other approaches to speed up the calculation, most of which rely on pre-computation.

3.3.1. Voxelized lighting

One method that is easy to implement and allows for very fast lookup times is to simply sample and voxelize the incoming light across the full domain of the scene. Since this is decoupled from the sample density of the camera rays, it is possible to sample less frequently than the final raymarch, and then interpolate values from the voxel representation. While this approach allows for fast lookups during final rendering, the pre-computation can be expensive, as for each voxel a full raymarch toward each light is performed.

```
void voxelizeLights(const Scene &scene, const std::vector<Light> &lights,
                    VoxelBuffer &lightBuffer)
{
  BBox dims = lightBuffer.dims();
  for (int k = dims.min.z; k <= dims.max.z; ++k) {
     for (int j = dims.min.y; j <= dims.max.y; ++j) {
        for (int i = dims.min.x; i <= dims.max.x; ++i) {
           V3f vsP = discreteToContinuous(i, j, k);
           V3f wsP;
           lightBuffer.mapping().voxelToWorld(vsP, wsP);
           Color incomingLight = 0.0f;
           for (int light = 0; light < lights.size(); ++light) {
              float intensity = lights[light].intensity(wsP);
              // Raymarch toward light to compute occlusion. This is a costly operation.
              float occlusion = computeOcclusion(lights[light].wsPosition(), wsP);
              incomingLight += intensity * (1.0 - occlusion);
           }
           lightBuffer.lvalue(i, j, k) = incomingLight;
        }
     }
  }
}
```

The biggest downsides to voxelized lighting is the cost of pre-computation. Because each voxel is calculated separately, we suffer similar performance problems as brute-force calculations, although we gain control over the sampling density (i.e. the resolution of the voxelized lighting buffer), which can be used to tune the quality/speed trade-off.

3.3.2. Deep shadows

One important aspect of the lighting calculation that isn't taken advantage of in the voxelized lighting approach is that light travels linearly from each light source. If the incoming light at voxel V1 in the illustration below has been computed, then the incoming light calculation at voxel V2, V3, etc. could potentially use the occlusion value at V1 to speed up their calculations. In practice however, figuring out which values can be re-used quickly becomes difficult and incurs its own performance overhead from storing book keeping data.

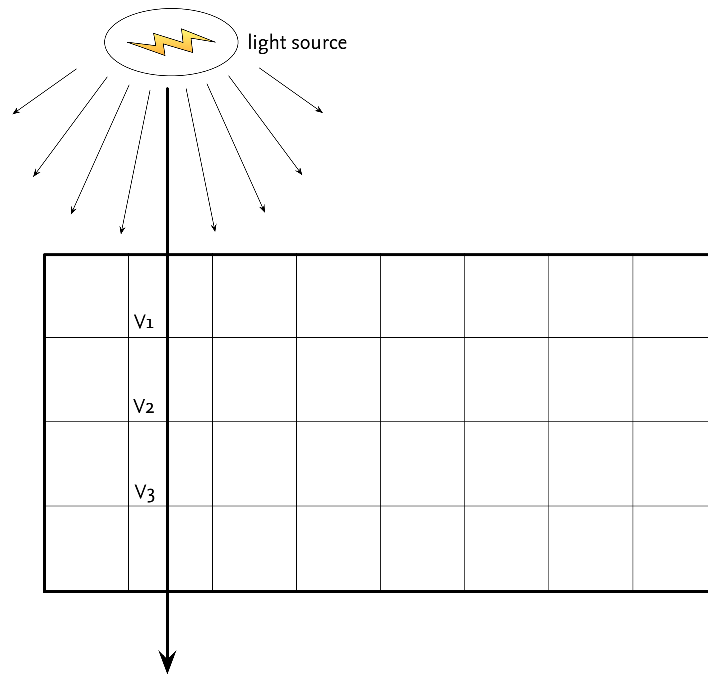
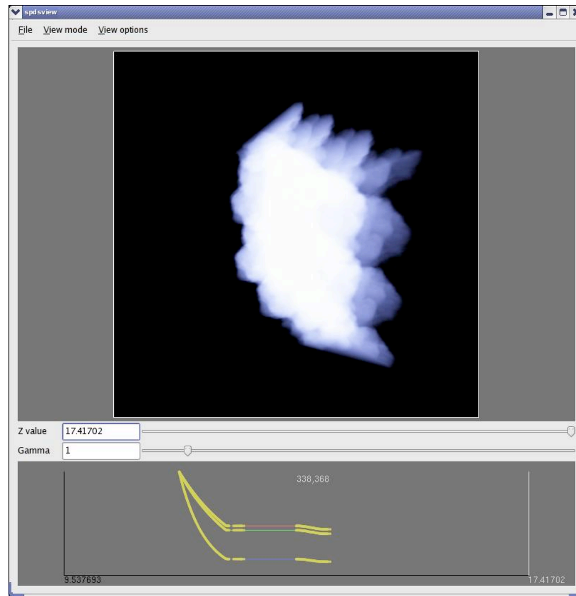


Illustration of light propagation

A better approach is to simply calculate occlusion as seen from the light. This is equivalent to how *shadow maps* work, and it requires us to choose a resolution controlling how finely the scene is to be sampled. For each pixel in this map, we then need to calculate the transmittance function.



Visualization of a deep shadow map, with spectral transmittance function for a pixel displayed below

This turns out to be trivial, as it is the exact same calculation that we perform when raymarching a volume from the camera and only accumulating transmittance/opacity. The only difference is that we need to record what the accumulated transmittance is at each raymarch step, so that this can later be queried for any point in space that is visible from the light source. The simplest way of storing the data is as a sequence of depth/transmittance pairs, ordered away from the light.

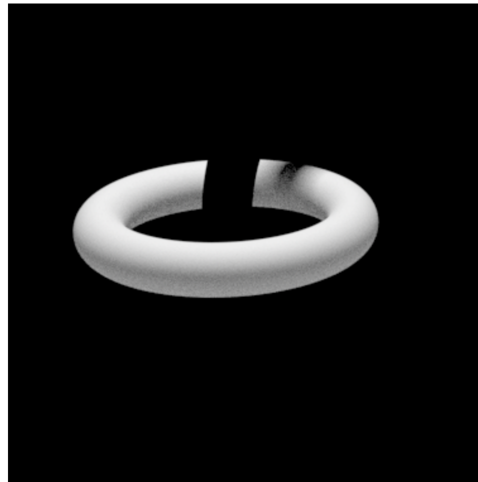
Technically, we could voxelize this data set and use it in beauty renders the same way as in the previous section, but a more efficient way to store this data is to leave it in its native form, i.e. a monotonically decreasing function per pixel seen from the light source. When storing it in this way, it is equivalent to the *deep shadow maps* technique described by Tom Lokovic and Eric Veach in their paper *Deep Shadow Maps* [Lokovic, 2000].

Deep shadow-style maps have several advantages. The transmittance function is mostly smooth and can be compressed efficiently to reduce the storage required both in memory and on disk. Also, since the number of samples in the transmittance function can vary from pixel to pixel, there is little cost associated with storing empty pixels. The downside is that the cost of lookups into the transmittance function are higher than for a voxelized representation, because each lookup requires a search in the transmittance function vector in order to find the appropriate depth sample.

3.4. Holdouts

Volumetric elements rarely exist in isolation, and in almost any shot there are other elements which the volumetric must integrate with in the final composite. *Holdouts* are a common way of dealing with integration of multiple elements, both in surface and volume rendering. A holdout is an object in the scene that occludes and shadows other objects but does not itself show up in the final frame. *Matte objects, phantom objects* and *holdout objects* are all different names for the same thing.

Ground and object A visible, object B as holdout



Object B visible, ground and object A as holdouts

Composited image (additive)

Color correction applied to object B's image

Holdouts allow a render to be broken into multiple images that can be manipulated individually, and that composite easily into a final frame. They also allow the breaking down of complex scenes, as object occlusion is handled per-pixel, without the need to know the correct depth sort order. In the example

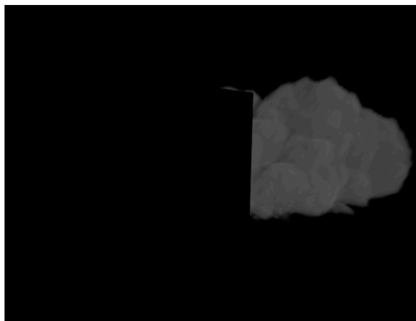
above, separating objects A and B without the use of holdouts would be tedious, as they both occlude each other in different parts of the image.

Holdouts are fairly trivial to implement in a surface shader, but in volume rendering, they can be implemented in a number of different ways, depending on the rendering approach and the type of holdout types that are supported.

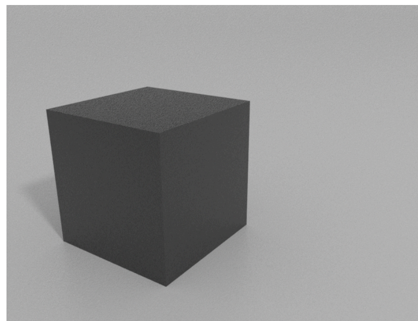
3.4.1. Holdouts in volume rendering

In the microvoxel-based renders that handles both surfaces and volumes (such as PRMan or Mantra), geometry-based holdouts are a convenient way of defining holdout objects. For a raymarch-based volume renderer however, geometric holdouts are somewhat cumbersome to deal with. In raymarching, holdout objects need to be able to answer a visibility query for any point in space along a ray, as seen from the camera's position, which means a ray must be traced against the geometry each time the information is required. To make things even more expensive, the holdout value should represent the pixel coverage or transparency at a given depth, so the holdout value usually needs to be supersampled and jittered across the pixel, because a simple binary visible/hidden answer will cause aliasing artifacts at geometry edges.

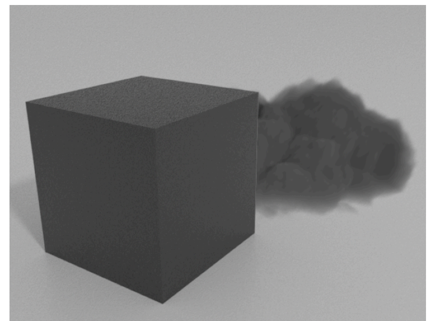
We can take advantage of the fact that this holdout query is the same form of query discussed in the precomputed lighting section, where a *transmittance function* determined how much light could travel between a light source and a given point in space. Holdout functions answer the inverse question: how much light could travel from a given point in space all the way to the camera?



Volumetric element with surfaces held out



Rendered image of surfaces



Composite using over operator

3.4.2. Implementing holdouts

If we consider the calculation of transmittance in the raymarch loop

```
T *= exp(-tau * stepLength);
```

We can rearrange this expression in the following way

```
T = T * exp(-tau * stepLength)
T = 0 * (1 - exp(-tau * stepLength)) + T * exp(-tau * stepLength)
T = lerp(0, T, exp(-tau * stepLength))
```

Which is to say that our incremental multiplication of *T* may be seen as a linear interpolation (or *lerp*) between zero and *T* by the factor *exp(-τ * stepLength)*. While this is rather useless in itself, it becomes a much more convenient formulation once we introduce holdouts.

A holdout object in volume rendering only needs to answer one question: at a given point in the scene, how visible is that point to the camera? That is to say

```
Color Tholdout = holdout.transmittance(wsP);
```

Logically, this function should start at 1.0 and be monotonically decreasing for any line pointing away from the camera, since once an obstacle is found its reduction of transmittance cannot be undone. If we consider the zero value in our `lerp()` expression above, that corresponds to the expected result of a holdout object that occludes nothing, i.e.

```
Tholdout = 1;
opacity = 1 - Tholdout;
opacity = 0.
```

As it turns out, accounting for the holdout object is as easy as replacing the zero in the lerp expression with `1 - Tholdout`. So the complete formulation of the raymarcher's transmittance calculation becomes:

```
Tholdout = holdout.transmittance(wsP);
T = lerp(1.0 - Tholdout, T, exp(-tau * stepLength));
```

If we were to create a procedural holdout function that emulated a semi-transparent glass pane right in front of the camera lens, we might expect it to always return 0.5, for example. Logically, this would be reflected in our final alpha and color values by making them half as large.

3.4.3. Holdout maps

In the previous chapter we saw that deep shadow maps are a convenient way of storing transmittance functions, in fact the information that is required of the holdout function is exactly the same information used in lighting calculation. One solution to supporting geometric holdouts is therefor to create a transmittance function for each pixel in the final output by raytracing any holdout geometry as a pre-process to the volume integration step. Also, users of some common surface renderers (for example Pixar's PRMan and SideFX's Mantra) can output a deep shadow or deep shadow-like representation of

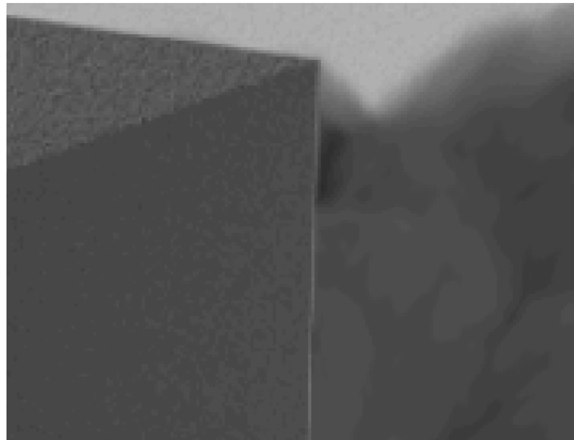
depth-varying pixel coverage or transmittance, which can then be used directly in a stand-alone volume renderer.

3.4.4. Problems with holdouts

In our first holdout example we saw how two interlocking rings could be held out against each other and then composited correctly using an additive operation. This type of render is a *two-sided holdout*, where each element is held out in all other images being rendered. When using a dedicated volume renderer to produce elements that need to composite against images from a surface renderer this becomes a catch-22, because the surface renderer would need to be aware of how to render the volume data as a holdout object, though of course the whole point of writing a standalone volume renderer was that we couldn't (or didn't want to) do it inside the surface renderer.

Because of this we often have to resort to *one-sided holdouts*, where the surfaces are rendered in their entirety and only the volume rendered image has objects held out. One-sided holdouts are technically inaccurate, but can be composited with reasonable results using an *over* operation instead of an add. The composite is correct for all pixels that have either no holdout or full holdout effect, but breaks down where partial occlusion happens.

To illustrate this we look closer at the previous example:



Light pixels bleeding through holdout edge

As we can see, the high intensity pixel values of the ground are creeping into pixels that should really only contain values form the dark box and the smoky volume. This is often referred to as a *matte line*, and shows the breakdown of one-sided holdouts. They occur because the high intensity values are already mixed with the foreground elements into the antialiased edge pixels, and there is no way to tell what portion of the pixel's value was contributed by values at any given depth; that information has been lost.

One way to work around one-sided holdouts is to output a *deep image* from both the volume renderer and the surface renderer, and do a *deep composite* of the two in a 2D application.

3.5. Motion blur

In the modeling section we discussed techniques for motion blur of volumetric primitives and how that is commonly treated at the rasterization step rather than at render time. Here we will introduce motion blur in a few new contexts, all of which need to be handled by a production grade volume renderer.

Calculating perfectly accurate motion blur is expensive (this is why we voxelized our volumetric primitives and smeared them to begin with), and implementation techniques tend to vary from facility to facility. The techniques range from brute-force solutions to clever tricks that fake the effect, though in production rendering each one can be a perfectly valid solution. Because the solutions vary from facility to facility, we will introduce the topic in this section, but the solutions will be discussed in the advanced sections later on in these notes.

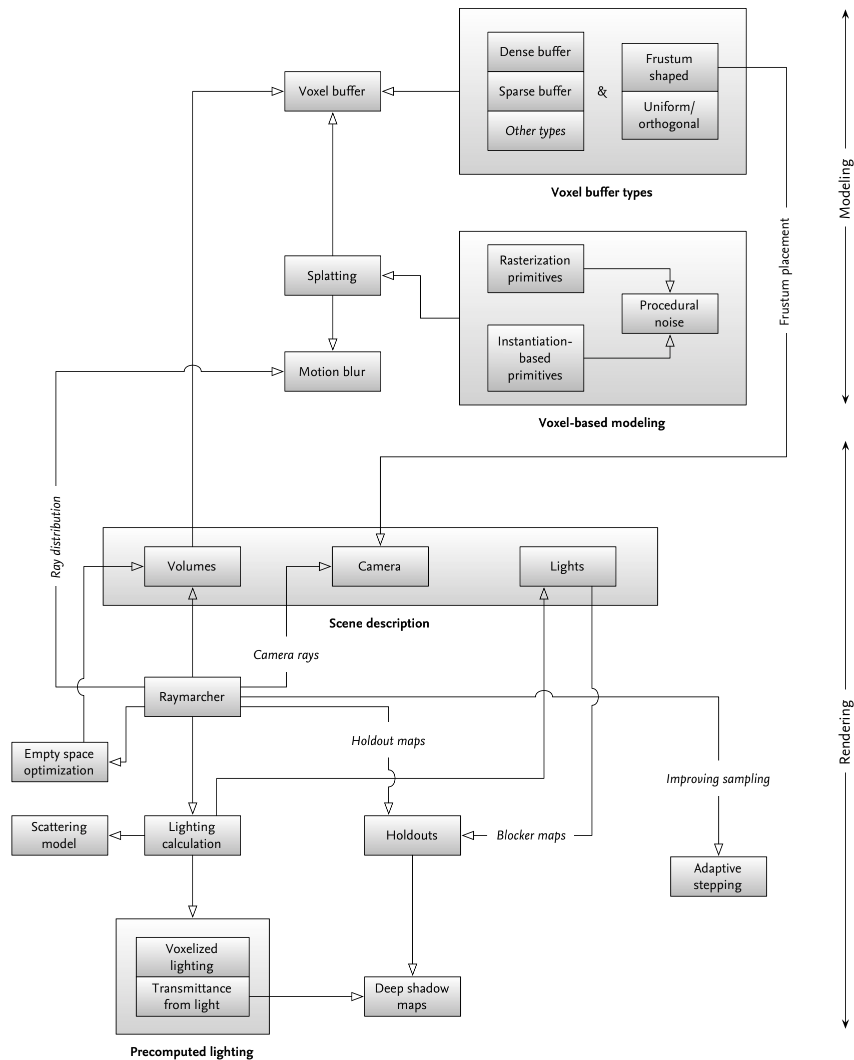
There are three main types of motion blur that need to be handled:

- **Object motion blur** is caused by motion of a volume or voxel buffer as a whole which often occurs when attaching a fluid simulation to a moving object in the scene. To account for the motion of the volume we need two or more *motion samples*. Each motion sample describes the local transform of the volume at a given time, usually the shutter open and shutter close times.
- **Deformation motion blur** occurs when the velocity throughout a particular volume varies. For example, a fluid simulation can be motion blurred at render time by looking not only at the density field but also at the velocity field.
- **Camera motion blur** occurs whenever the render camera itself is moving.

4. Putting it all together

The material covered so far should provide the basis needed to implement a simple volume modeling and rendering application. With those basics in mind, the next few chapters will dive into detail about how some actual battle-proven implementations work, and will show that the same problem often gets solved quite differently from facility to facility. Our hope is that, taken together, they will give a good overview of the state-of-the-art in production volume rendering.

The following diagram attempts to illustrate how each of the pieces described so far are connected to one another, and will serve the reader as an guide to how the advanced topics that follow fit into the volume rendering pipeline.



5. References & Further reading

References

GRITZ, L. 1998. Basic Antialiasing in Shading Language, Advanced RenderMan SIGGRAPH course, In *course notes pp. 62-80.*

HECKBERT, P. S. 1990. What Are The Coordinates Of A Pixel? In *Graphics Gems I, pp. 246-248. Andrew Glassner (editor), Academic Press.*

KAJIYA J. T., HERZEN B. P. V. 1984. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques (New York, NY, USA, 1984), ACM Press, pp. 165–174.*

KAPLER, A. 2002. Evolution of a VFX voxel tool. In *SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications, pages 179–179, New York, NY, USA. ACM.*

KAPLER, A. 2003. Avalanche! Snowy fx for xXx. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications, pages 1–1, New York, NY, USA. ACM.*

KISACIKOGLU, G. 1998. The making of black-hole and nebula clouds for the motion picture "sphere" with volumetric rendering and the f-rep of solids. In *SIGGRAPH '98: ACM SIGGRAPH 98 Conference abstracts and applications, page 289, New York, NY, USA. ACM.*

LOKOVIC, T. AND VEACH, E. 2000. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pages 385–392, New York, NY, USA. ACM Press/ Addison-Wesley Publishing Co.*

REEVES, W. T. 1983. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph., 2(2):91–108.*

SQUIRES, S. 2009. Cloud Tank effect. http://effectscorner.blogspot.com/2009/02/cloud-tank-effect.html.

Books

Pharr & Humphries – Physically Based Rendering. Morgan Kaufmann publ.
Ebert et al – Texturing & Modeling. Morgan Kaufmann publ.
Jules Bloomenthal (edited by) – Introduction to Implicit Surfaces. Morgan Kaufmann publ.

Websites

http://flam3.com/

Resolution Independent Volumes

Jerry Tessendorf and Michael Kowalski
Rhythm and Hues Studios

July 23, 2010

ii

amp=1
corr = 1.5
levy = -1.5
numchildren = 1000000000
octaves = 5
rough = 0.5

This document can be found at `http://tessendorf.org/reports.html`

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, 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 was a critical success early in FELT's development.

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

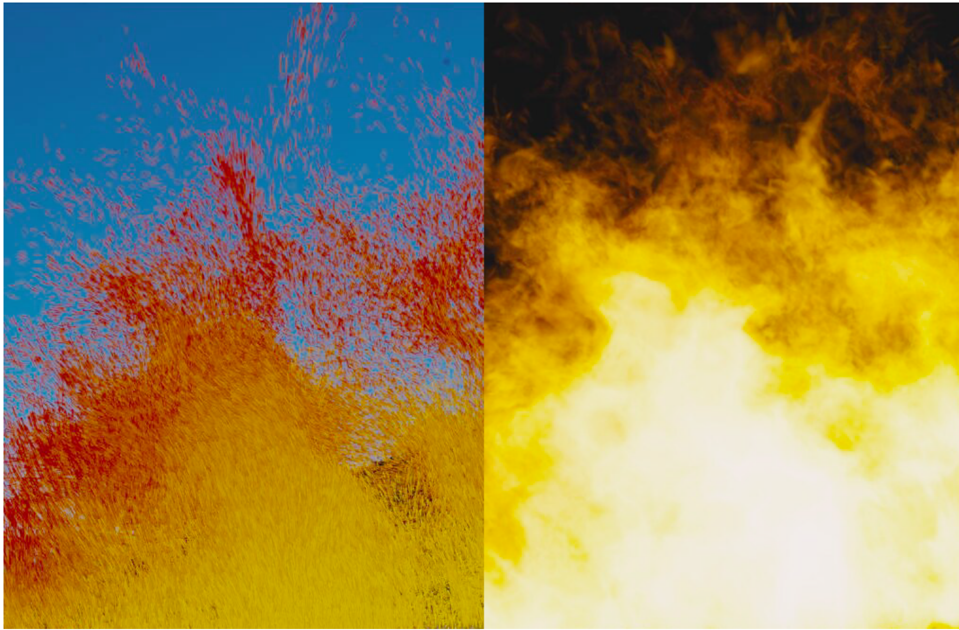


Figure 1: Early imagery showing the conversion of a particle system into a volumetric fire. The FELT algorithms used for this included early versions of gridless advection and SELMA.

Contents

1 Introduction **1**

1.1 A Brief on Volume Rendering 2
1.2 Some Conventions 3

2 The Value Proposition for Resolution Independence **7**

3 Cloud Modeling **11**

3.1 Cumulous cloud structure of interest 12
3.2 Levelset description of a cloud 12
3.3 Layers of pyroclastic displacement 14
3.3.1 Displacement of a sphere 14
3.3.2 Displacement of a levelset 16
3.3.3 Layering strategy 17
3.4 Clearing Noise from Canyons 20
3.5 Advection 23
3.6 Spatial control of parameters 23

4 Warping Fields **29**

4.1 Nacelle Algorithm 29
4.2 Numerical implementation 31
4.3 Attribute transfer 32

5 Cutting Up Models **35**

5.1 Levelset knives 35
5.2 Single cut 36
5.3 Multiple cuts 37

6 Fluid Dynamics **39**

6.1 Navier-Stokes solvers 39
6.1.1 Hot and Cold simulation scenario 40
6.2 Removing the grids 41
6.3 Boundary Conditions 44

7 Gridless Advection **51**
7.1 Algorithm 51
7.2 Examples 52

8 SEmi-LAgrangian MApping (SELMA) **61**

List of Figures

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. v

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. 13
3.2 Examples of classic pyroclastically displaced spheres of density. 15
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. 18
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. 21
3.5 Volume renders with various values of billow. Left to right, top to bottom: billow=0.33, 0.5, 0.67, 1, 2. 22
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. 24
3.7 Volume renders with various setting of advection, for billow=1. Top to bottom: No advection, medium advection, strong advection. 25
3.8 Volumetric bunny with spatial control over the pyroclastic displacement. 26

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. 33

4.2 Texture mapping of the object shape by transfering texture coordinates from the reference shape. 34

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. 38

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 $50 \times 50 \times 50$. 42

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 $50 \times 50 \times 50$. 43

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 $50 \times 50 \times 50$. 45

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 $50 \times 50 \times 50$. 46

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 $50 \times 50 \times 50$. 47

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. 49

7.1 Illustration of the effect of a single step of gridless advection. The unadvected density field is a sphere of uniform density. 52

7.2 Unadvected density distribution arranged from a collection of spherical densities. 53

7.3 Density distribution after 60 frames of advection and sampling to a grid each frame. 54

7.4 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. 54
7.5 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. 55
7.6 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). 56
7.7 Density distribution after 60 frames of gridless advection. The fine detail in the density field is resolved by using a fine raymarching step. 56
7.8 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. 58
7.9 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. 59

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. 63
8.2 Comparison of the performace of Gridless Advection and SELMA. 64
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. 65

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

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 fixed 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:

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

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

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

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

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 $O = 1 - T$.

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 is 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, 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.

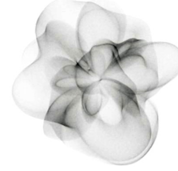
Cleaner coding of algorithms

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 accamodations for such.

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, 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 the air. 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. 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 accamodate 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

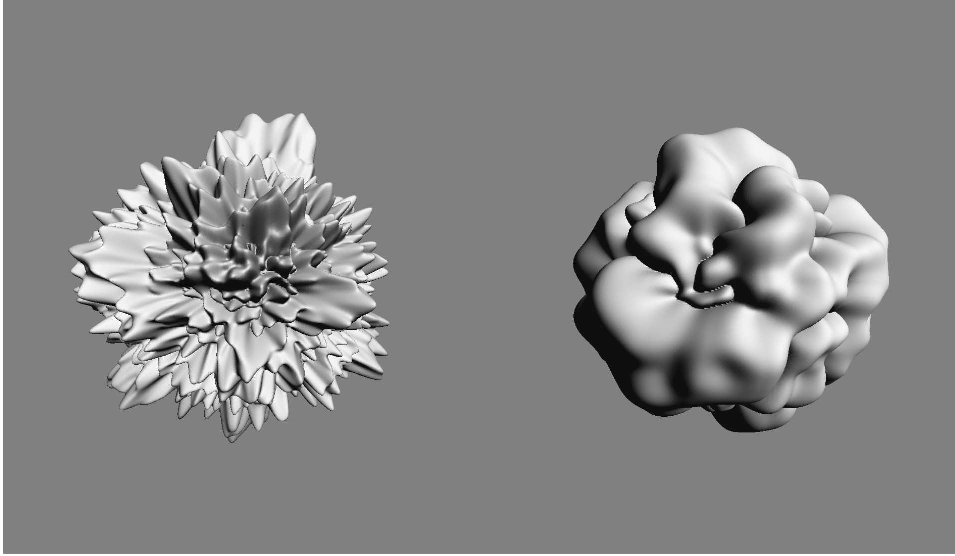


Figure 3.2: Examples of classic pyroclastically displaced spheres of density.

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

3. If $d - R < d_{\mathrm{bound}}$, then compute the displacement: The point on the unit sphere surface is $\mathbf{n} = (\mathbf{x} - \mathbf{x}_{\mathrm{sphere}})/d$. The displacement is $r = |\mathrm{Perlin}(\mathbf{n})|$. 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 particularly 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, $\mathbf{x}_{\mathrm{base}}$ 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 $r_i = \mathrm{Perlin_i}(\mathbf{n})$ 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. 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:

```
 // Function to generate and array of displacement layers
func scalarfield[] NoiseLayers( int nbGenerations, scalarfield scale, scalarfield
```

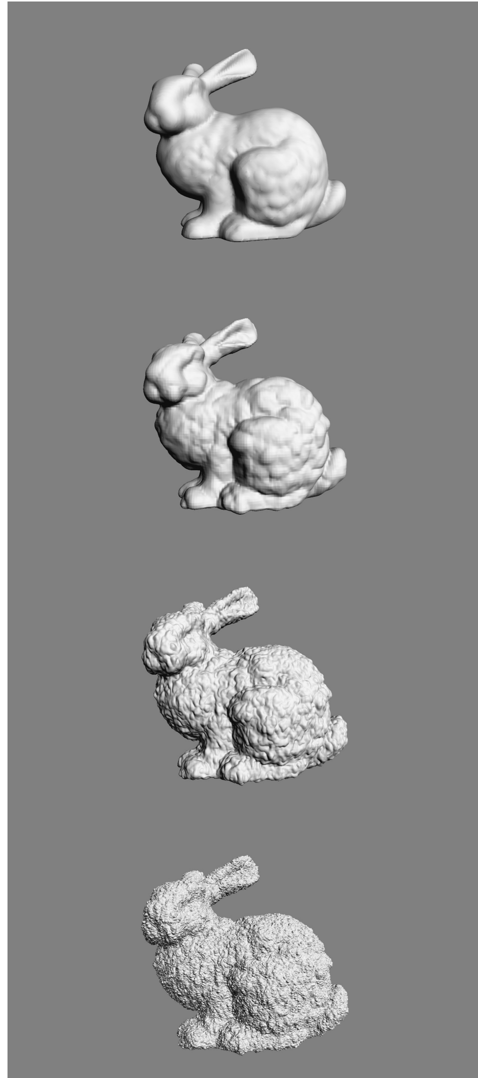


Figure 3.3: Illustration of layering of pyroclastic displacements. From top to bottom: No displacements; one layer of displacements; two layers; three layers. The displacements are applied to the levelset representation of the bunny, and the displaced bunny was converted into geometry for display.

```
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.

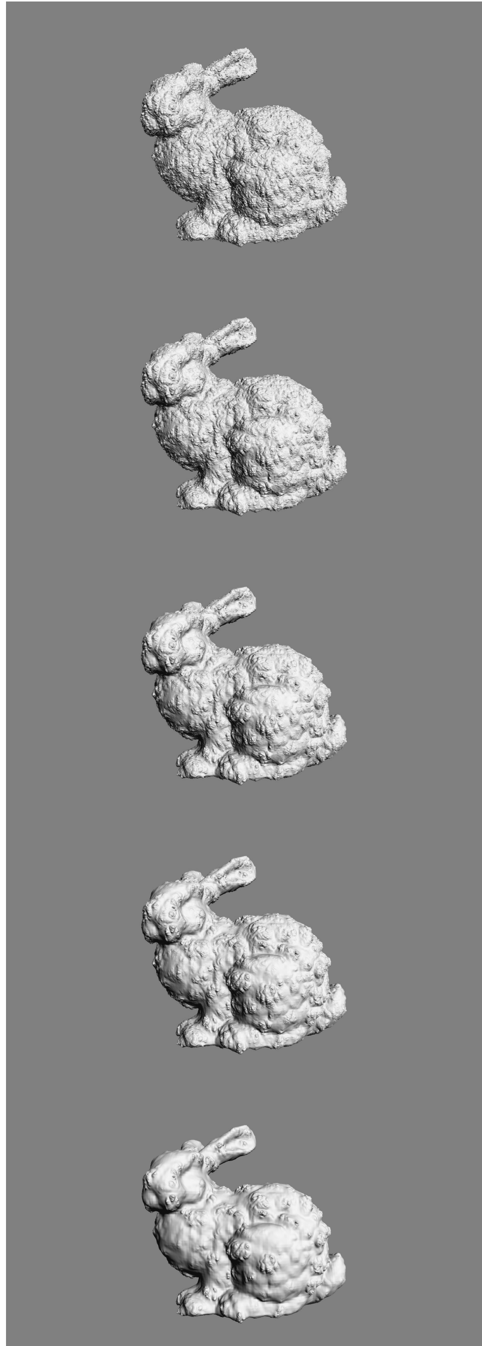


Figure 3.4: Illustration of clearing of displacements in the valleys using the billow parameter. The bottom of figure 3.3 illustrates the three layers of displacement with no billow applied. The noise is FFT-based, and $Q = 1$. From top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

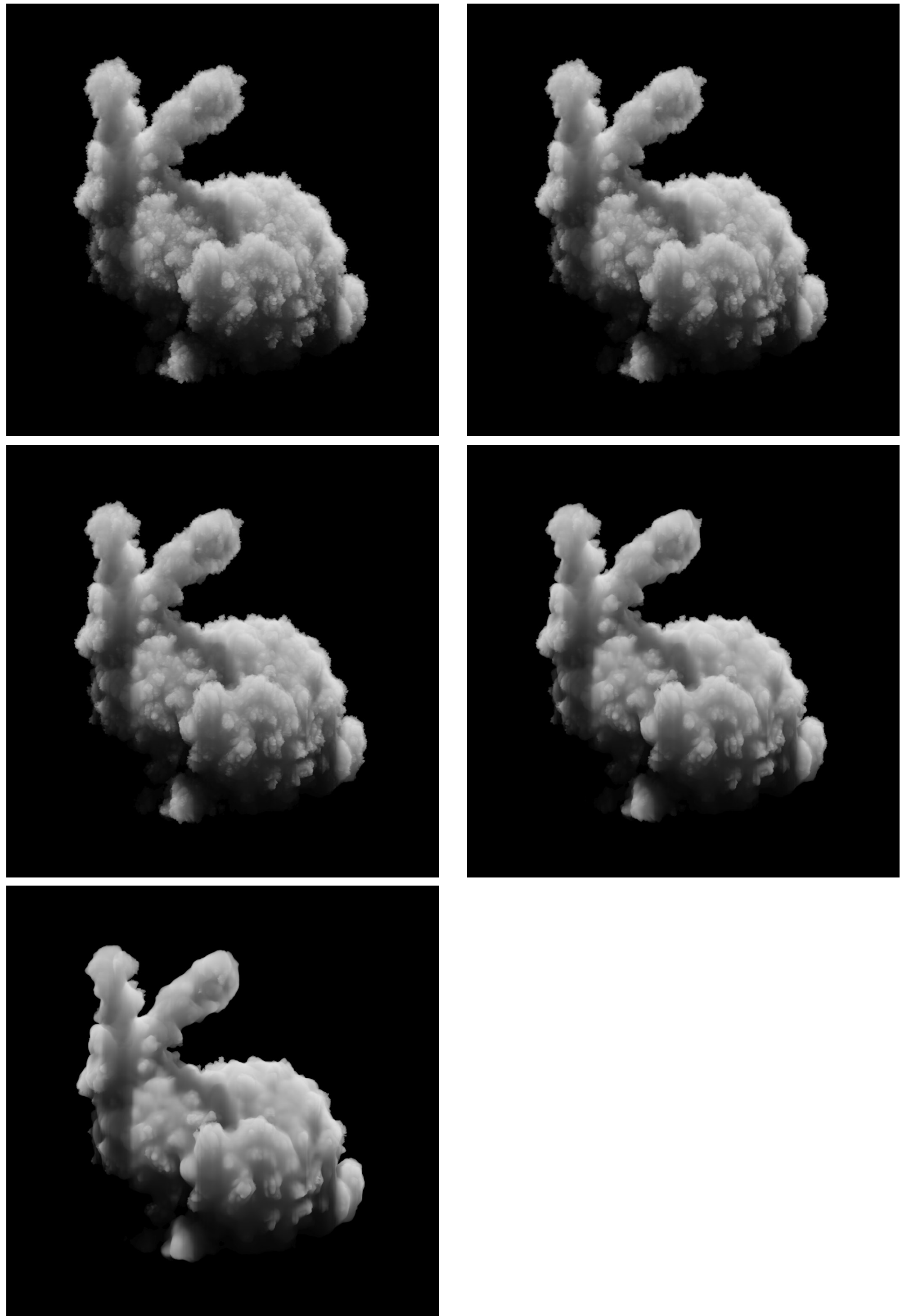


Figure 3.5: Volume renders with various values of billow. Left to right, top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

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.

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

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 accamodate 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 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

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.

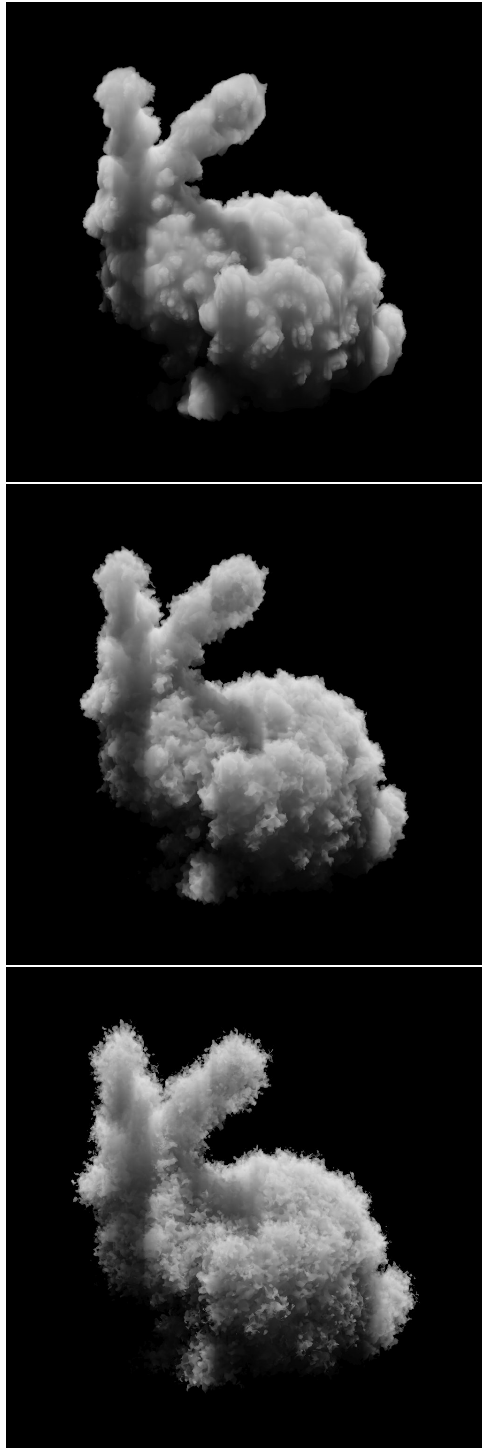


Figure 3.7: Volume renders with various setting of advection, for billow=1. Top to bottom: No advection, medium advection, strong advection.

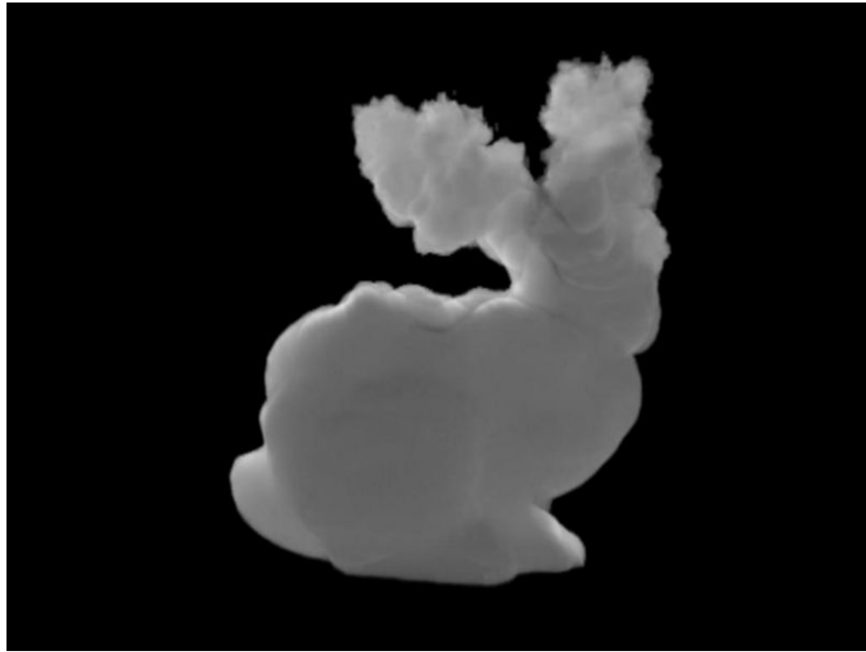


Figure 3.8: Volumetric bunny with spatial control over the pyroclastic displacement.

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.

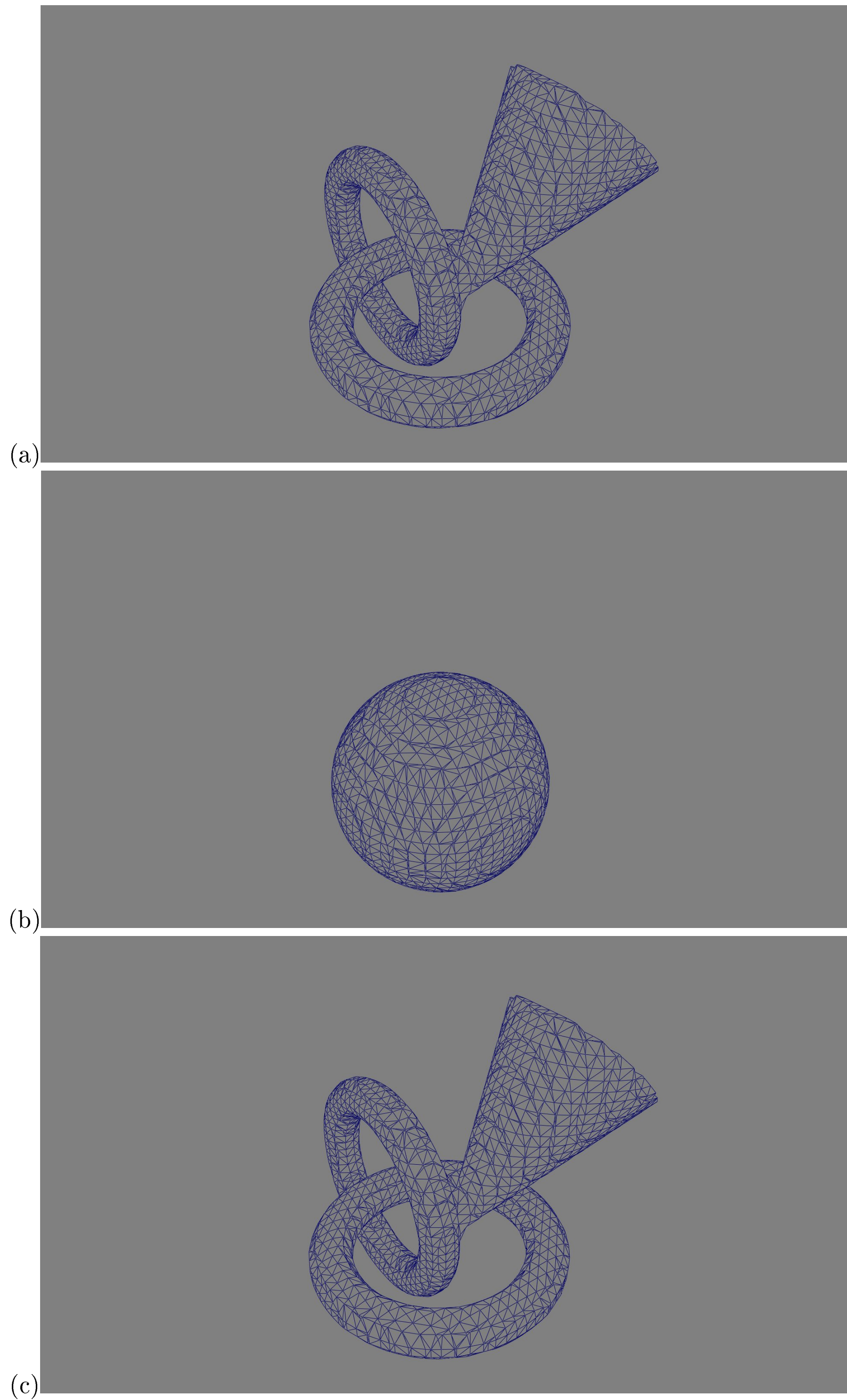


Figure 4.1: Warping of a reference sphere into a complex shape (cone and two torii). (a) Object shape; (b) Reference sphere; (c) Warp shape output from 1 iteration.

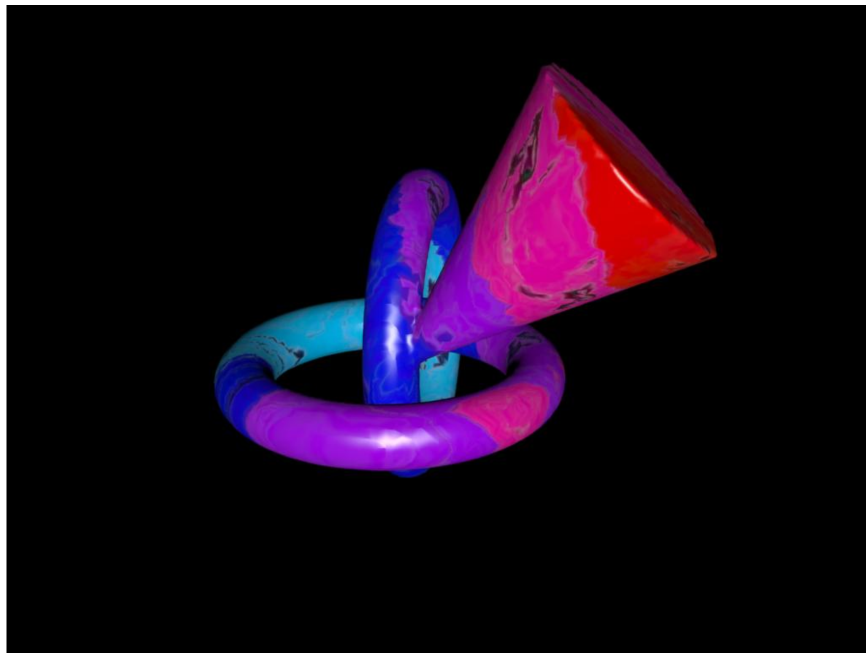


Figure 4.2: Texture mapping of the object shape by transfering texture coordinates from the reference shape.

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:

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

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.

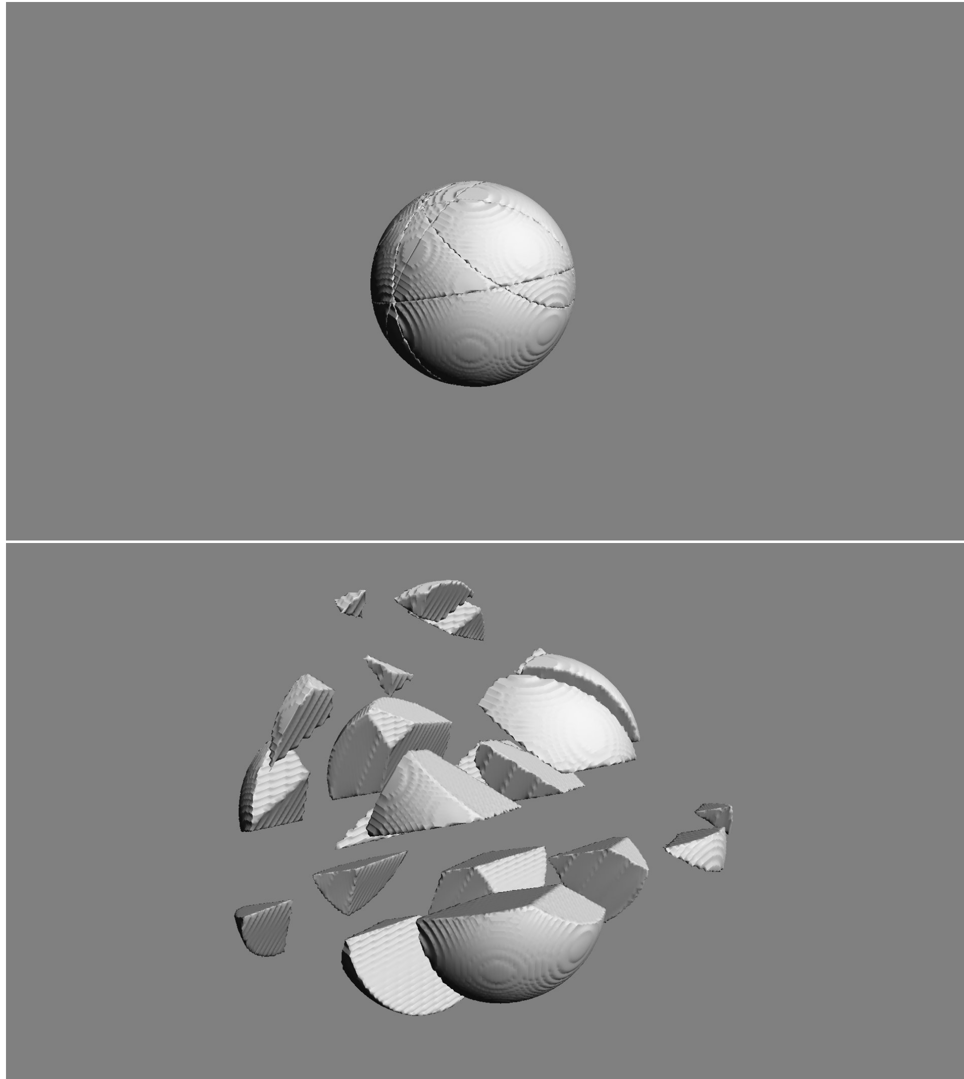


Figure 5.1: A sphere carved into 22 pieces using 5 randomly placed and oriented flat blades. The top shows the sphere with the cuts visible. The bottom is an expanded view of the pieces.

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 );
```

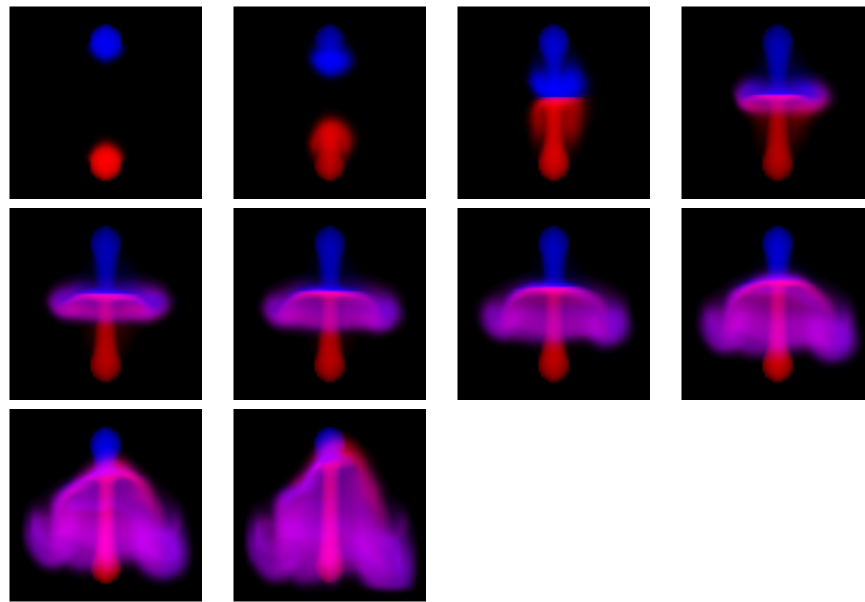


Figure 6.1: Simulation sequence for hot and cold gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

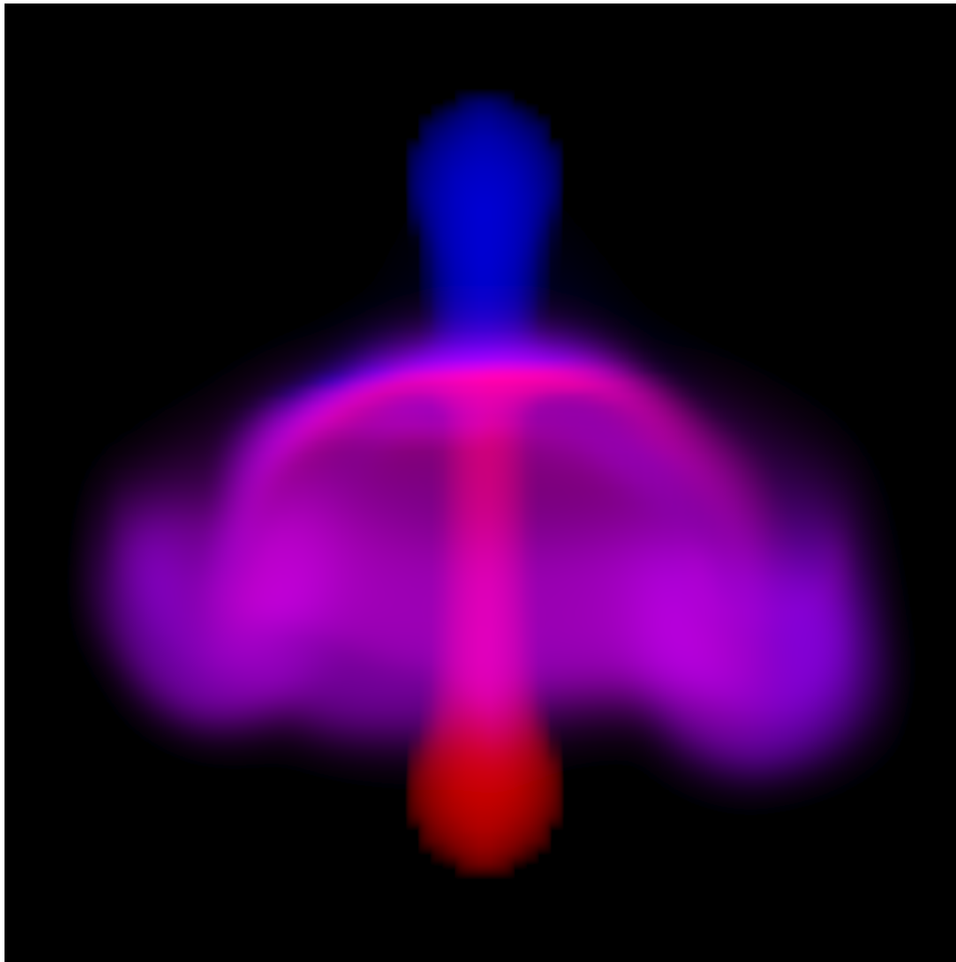


Figure 6.2: Frame of simulation of two gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

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

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

The FELT code for this is

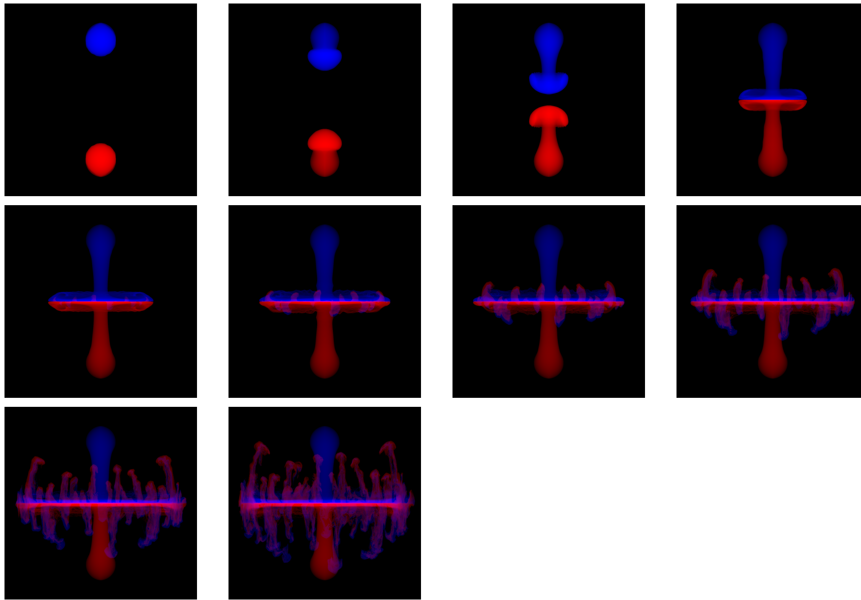


Figure 6.3: Sequence of frames of a simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected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

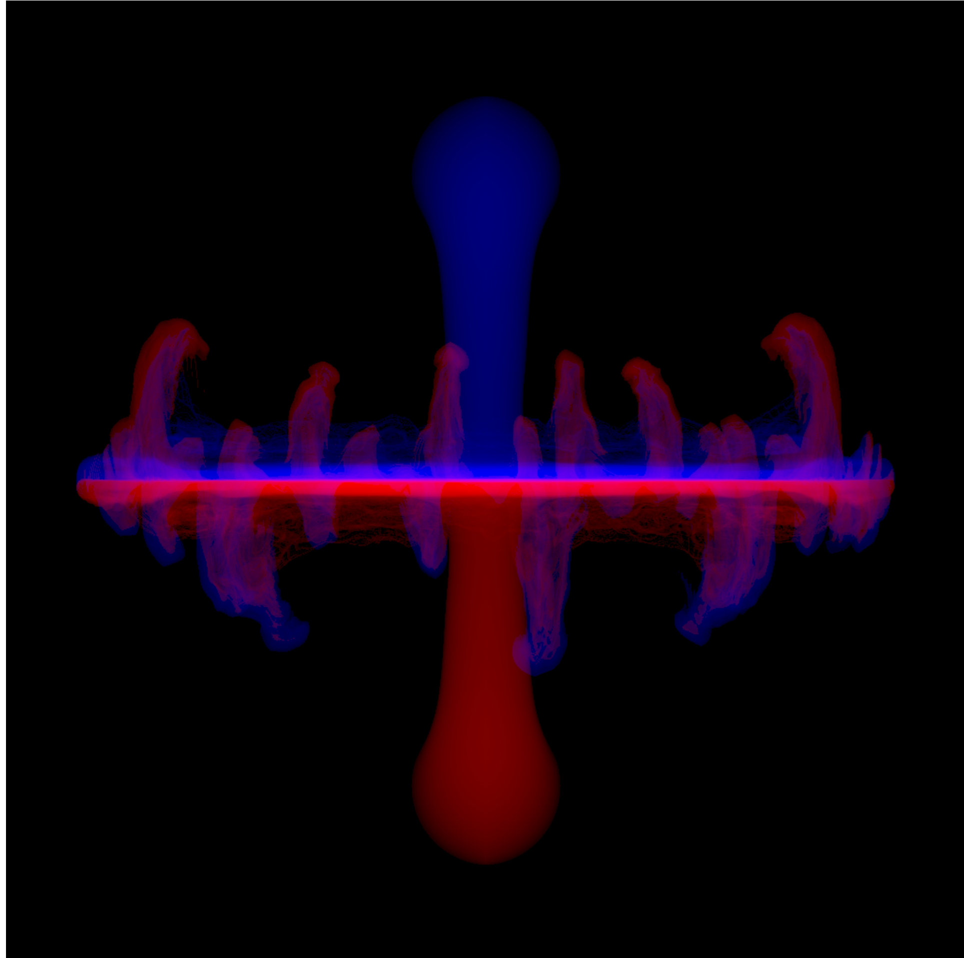


Figure 6.4: Frame of simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected but not sampled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

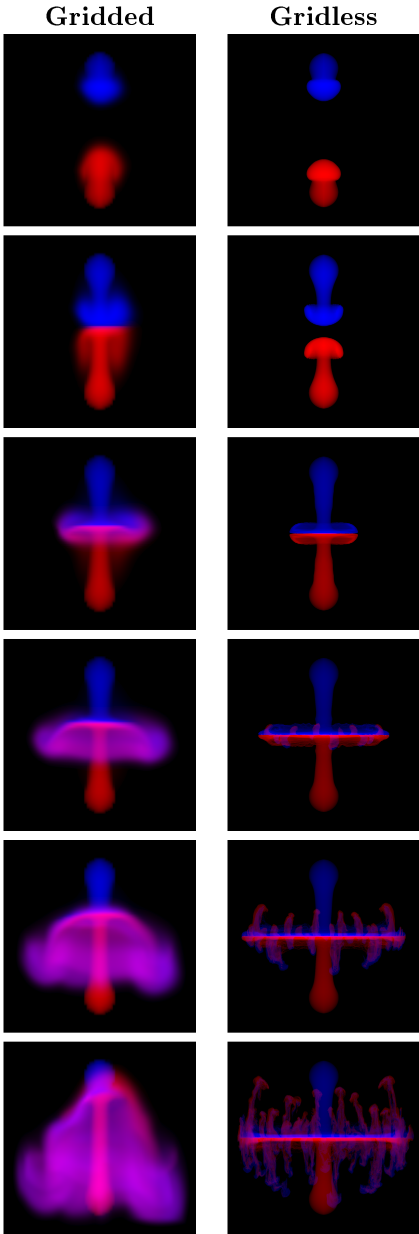


Figure 6.5: Simulation sequences with density gridded (left) and gridless (right). The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution is $50 \times 50 \times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

```
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.

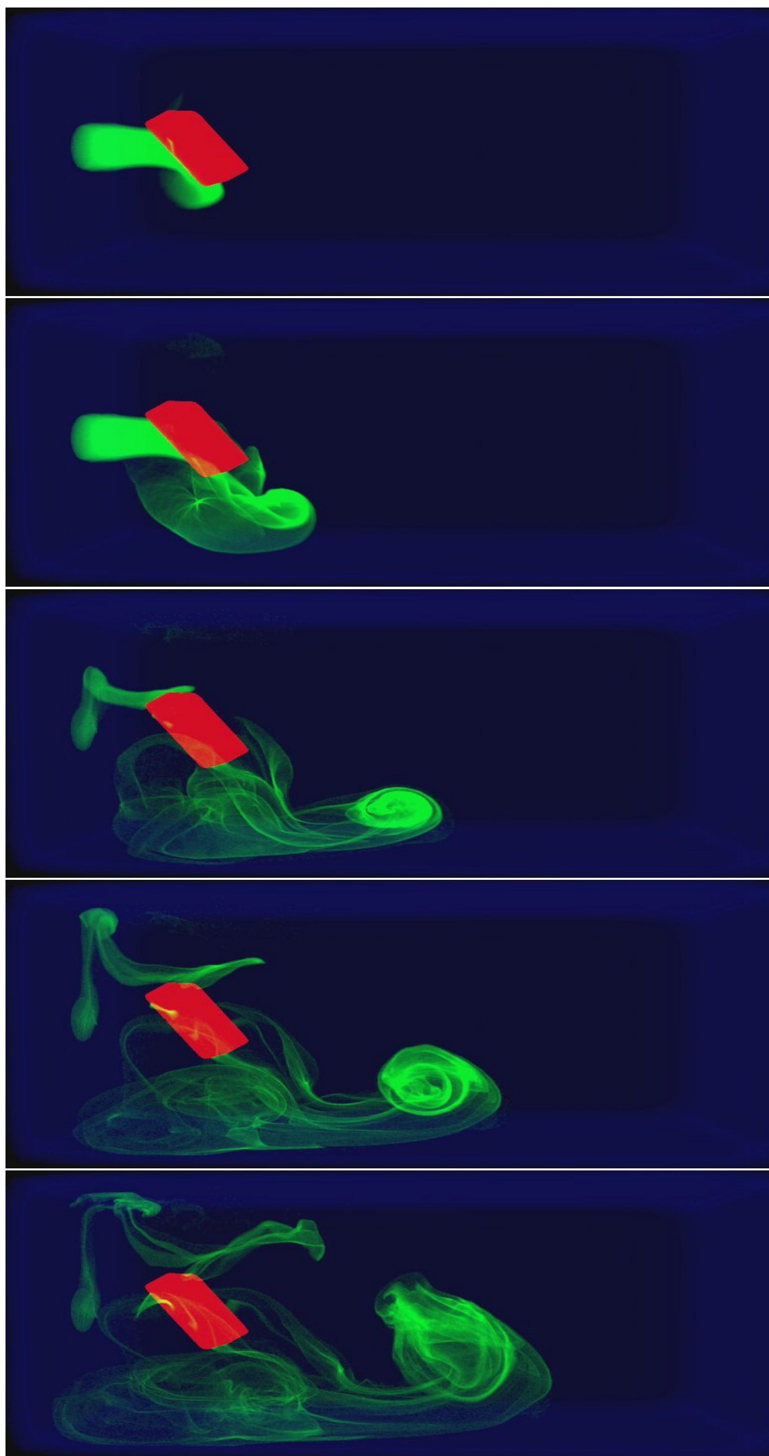


Figure 6.6: Time series of a simulation of bouyant flow (green) confined within a box (blue boundary) and flowing around a slab obstacle (red). Frames 11, 29, 74, 124, 200 from a 200 frame simulation.

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 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. Figure 7.1 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.

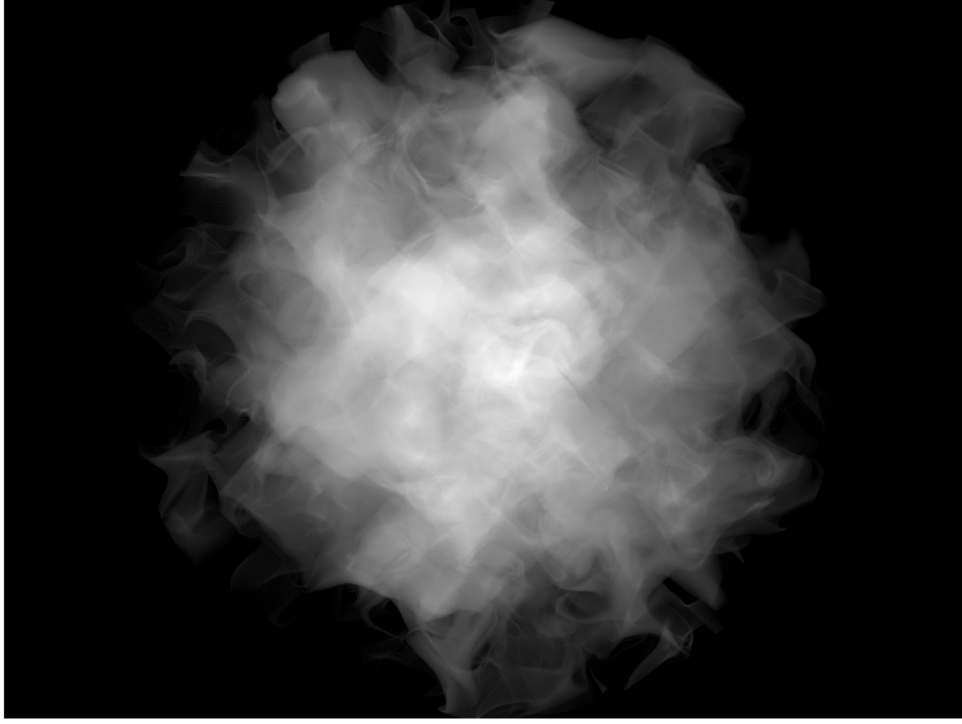


Figure 7.1: Illustration of the effect of a single step of gridless advection. The unadvected density field is a sphere of uniform density.

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.

7.2 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.

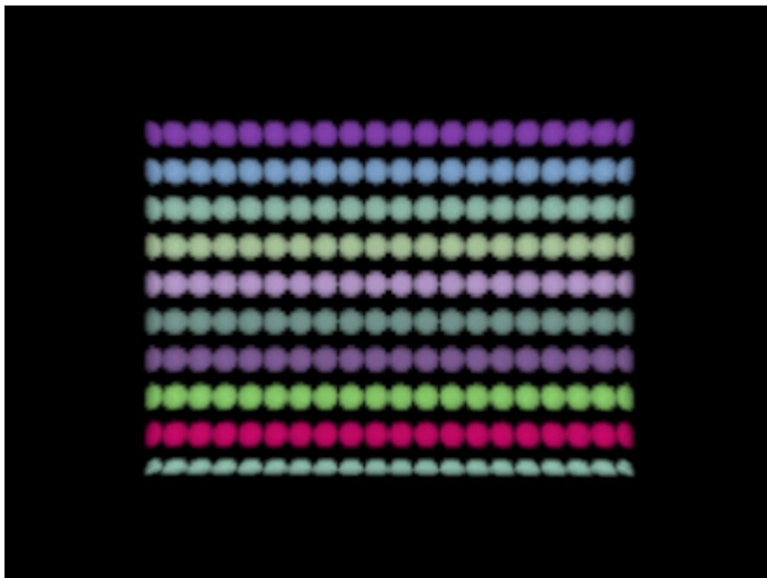


Figure 7.2: Unadvected density distribution arranged from a collection of spherical densities.

Figure 7.2 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 7.3. 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 7.1, the result is in figure 7.4. 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 7.5. 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 7.6 by raymarching with a step size 1/10-th the grid resolution. Finally, just to carry it to the extreme, figure 7.7 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 simu-

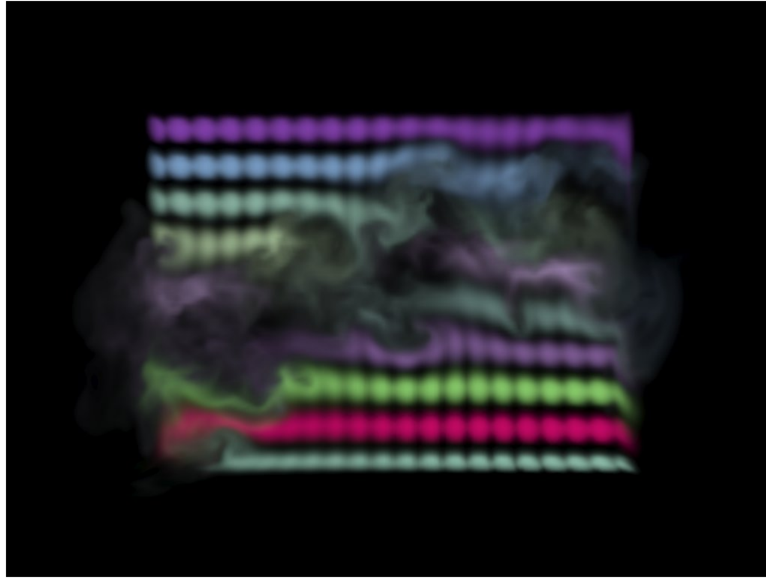


Figure 7.3: Density distribution after 60 frames of advection and sampling to a grid each frame.

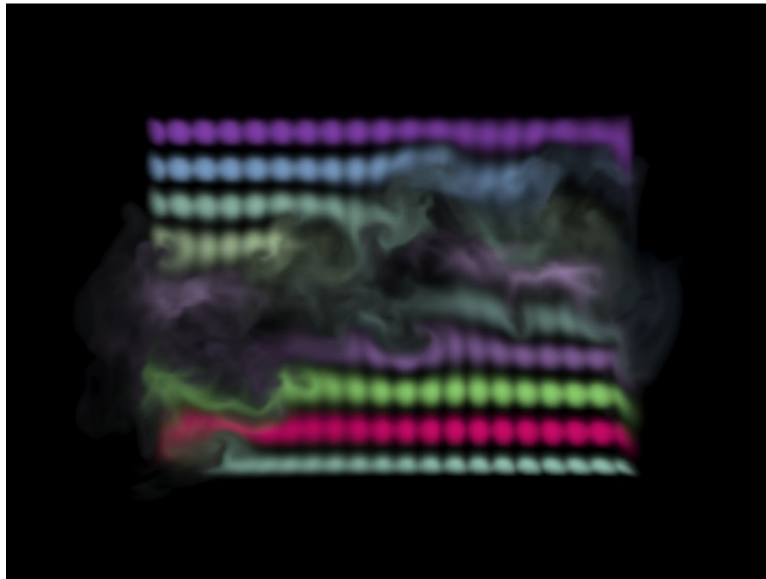


Figure 7.4: Density distribution after 59 frames of advection and sampling to a grid each frame, and one frame of gridless advection. The edges of filaments have been subtley sharpened.

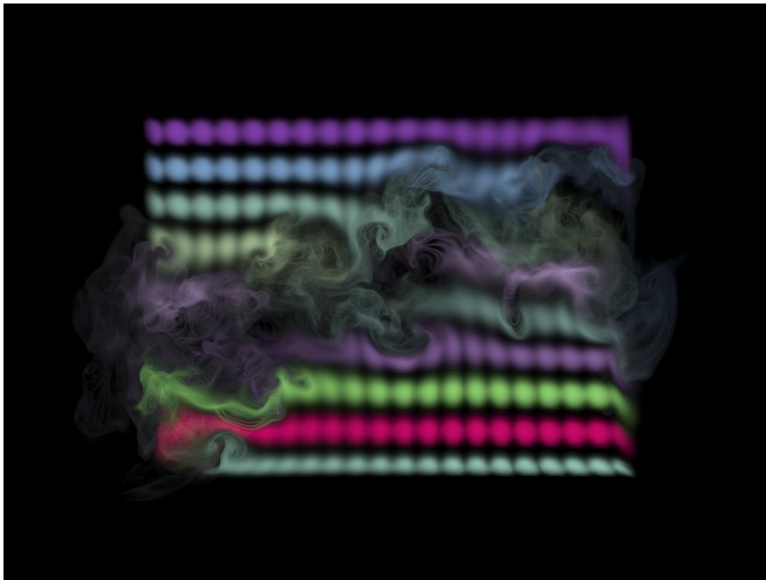


Figure 7.5: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The sharpening of details has increased to the point that the detail is finer than the raymarch stepping, causing significant aliasing in the render.

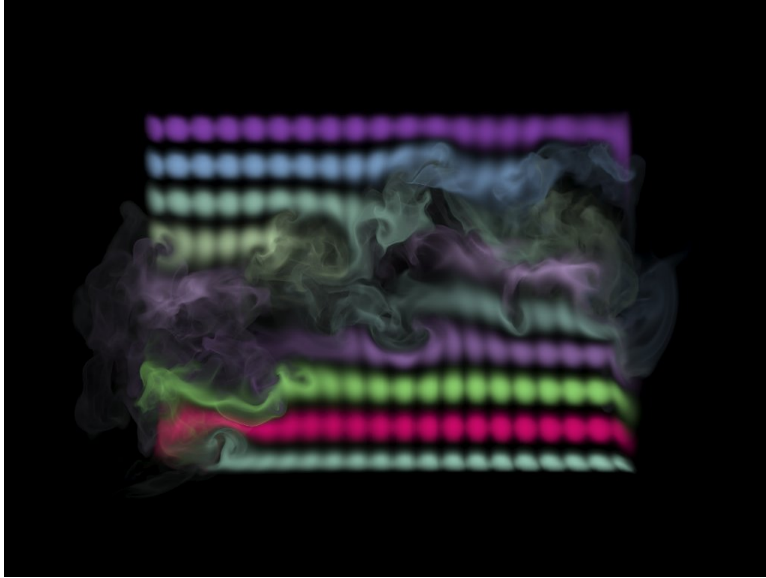


Figure 7.6: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The fine detail in the density field is now resolved by using a finer raymarching step (1/10-th the grid resolution).

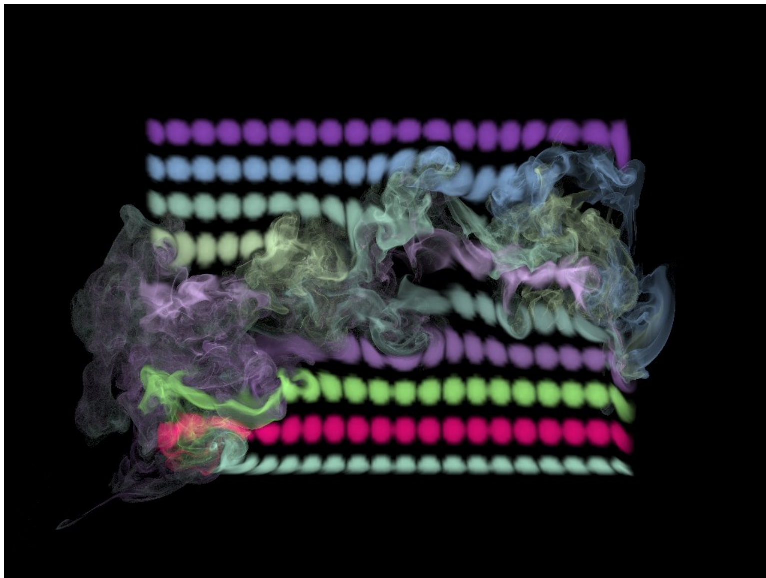


Figure 7.7: Density distribution after 60 frames of gridless advection. The fine detail in the density field is resolved by using a fine raymarching step.

lation 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 7.8.

We can evaluate the relative performance of various options, e.g. how many gridless steps to take, using the graph in figure 7.9, 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 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).

Figure 7.8: 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.

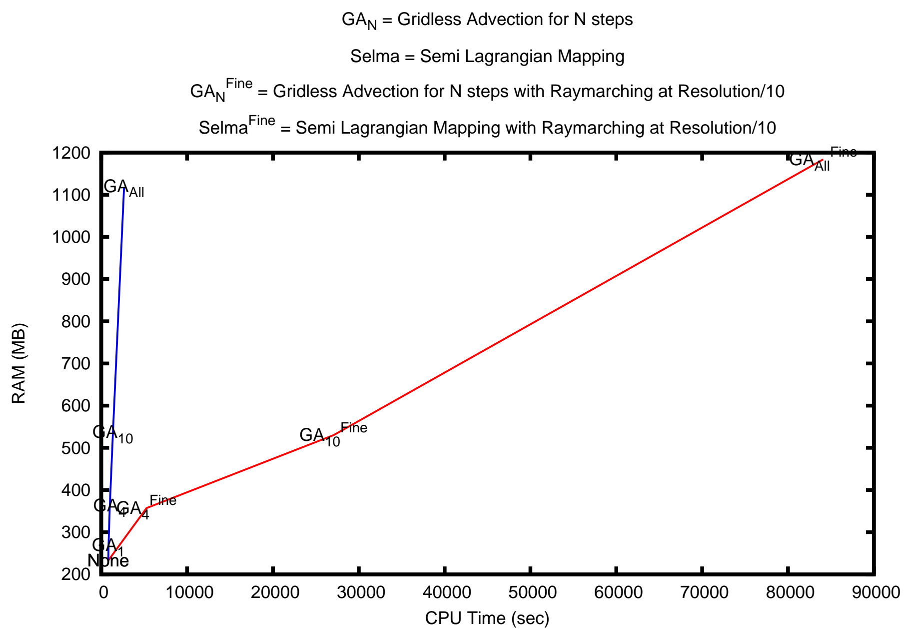


Figure 7.9: Performace of gridless advection as the number of advection frames grows. The steep blue line is gridless advection rendered with the raymarch step equal to the grid resolution. The red line is a raymarch step equal to one-tenth of the grid resolution. These results are not from a production-optimized renderer, so time and memory values should be taken as relative measures only.

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 7.2 through 7.7, 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

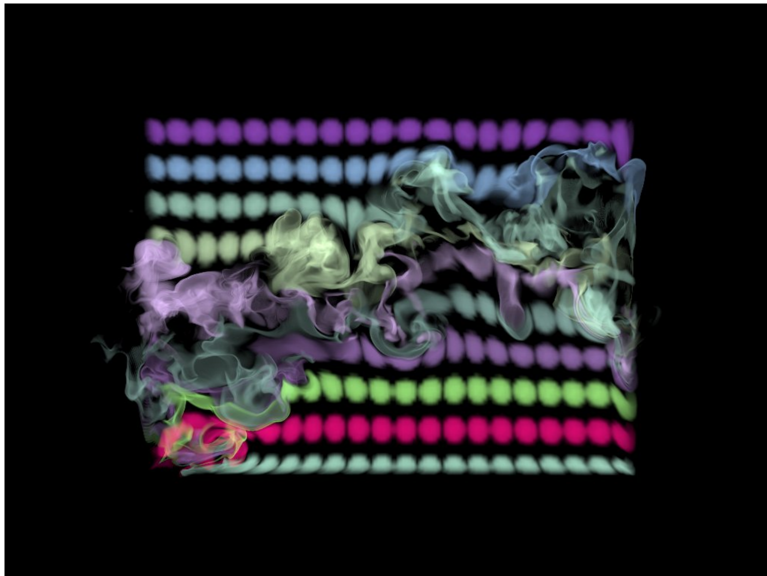


Figure 8.1: Density distribution after 60 frames of SELMA advection. The fine detail in the density field is resolved by using a fine raymarching step.

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.

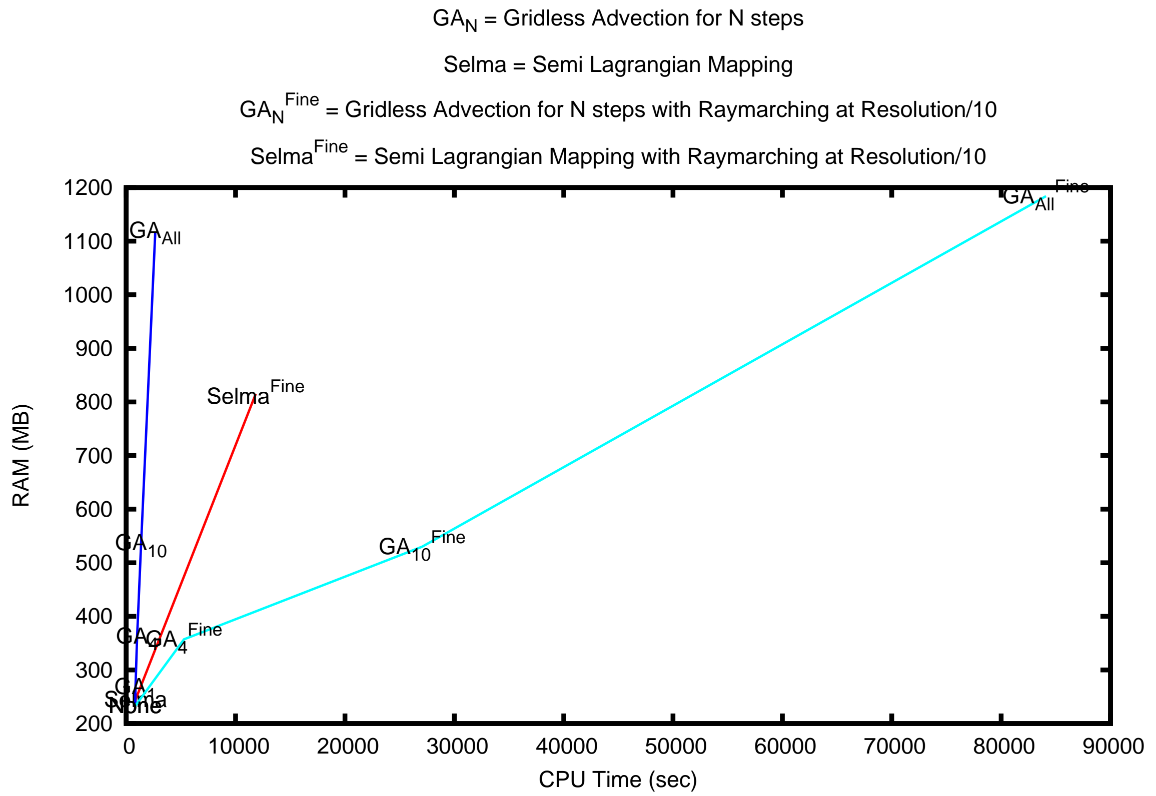


Figure 8.2: Comparison of the performace of Gridless Advection and SELMA.

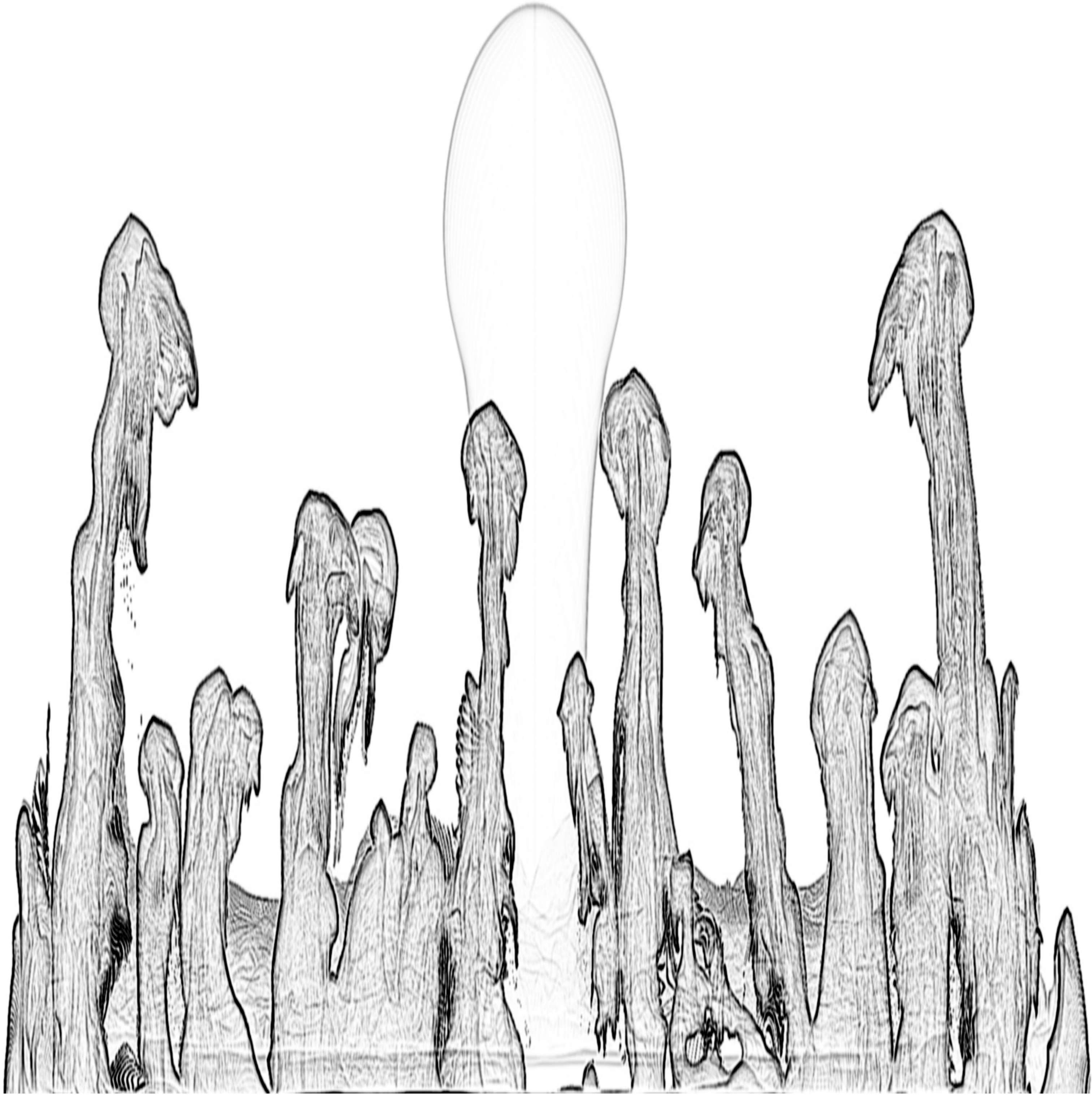
Figure 8.3: Example of SELMA used in the production of *The A-Team* to apply a simulated turbulence field to a modeled cloud volume as an aircraft passes through.

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.



Index

Ahab, iv
attribute transfer, 32

boundary conditions, 44
bunny, 18, 23, 26

cfd, 39
cloud, 1, 11, 12
cloud modeling, 11
color, 2
compositing, 1
cracktastic, 35, 67
cumulous cloud, 12, 13, 23
cutting geometry, 35

density, 2, 3, 9, 11, 12, 14, 15, 23, 39–41, 44–48, 51–53, 56, 57, 61–63
displacement, 14, 16
domain, 3
dust, 1

explosion, 35
extinction coefficient, 3

Felt, iv, v, 1–3, 9, 11, 12, 16, 17, 19, 23, 29, 31, 40, 44, 62
Felt script, 3, 4, 7, 16, 17, 19, 31, 36, 37, 40, 41, 44, 62
field, 3
fire, 1
fluids, 39
fracture, 35

gridless advection, iv, v, 8, 9, 23, 24, 39, 41, 44, 48, 51–59, 61–63

Heaviside step function, 36
hog, iv

levelset, 12, 14–20, 29, 32, 35–37, 44
levelset knife, 35

matrixfield, 3

nacelle, 29
nacelle algorithm, 29
notation, 3
Nuke, 8

opacity, 2, 3

parameter control, 23
Photoshop, 8
productions, 1
pyroclastic, 14

raymarching, 2
reflecting boundary, 44
resolution independence, 7
Rhythm and Hues Studios, i, iv, 1, 11

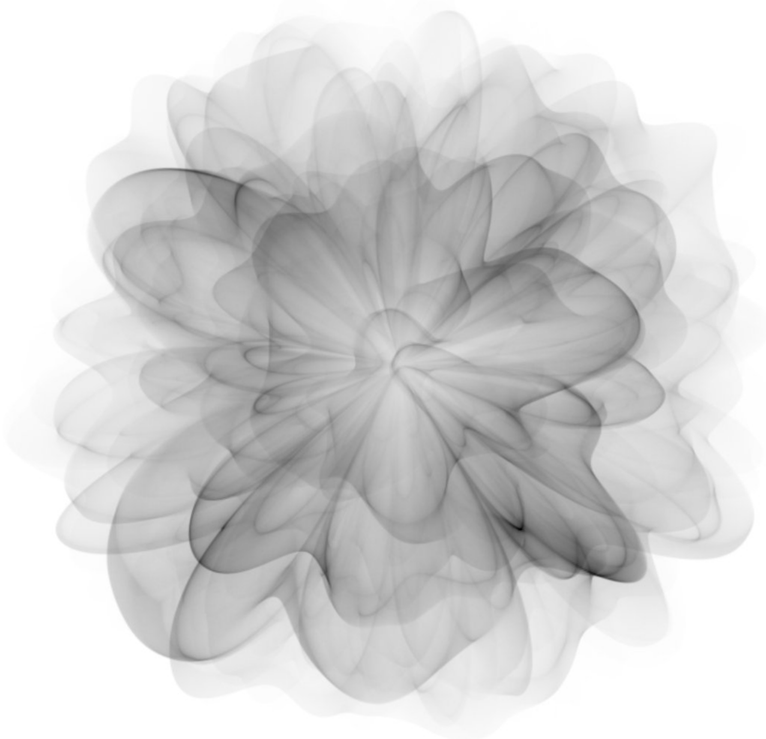
scalarcache, 4, 5, 20, 31
scalarfield, 3–5, 7, 12, 14, 16, 17, 19, 23, 31, 41, 48
SELMA, iv, v, 8, 51, 57, 61–63, 65
semi-lagrangian mapping, 61
smoke, 1
splash, 1

Taylor expansion, 30
The A-Team, 11, 23, 24, 58, 63, 65
transmissivity, 3

vectorcache, 4, 62
vectorfield, 3–5, 12, 16, 17, 20, 31, 32, 41, 62

warping, 29

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



Mantra Volume Rendering

Andrew Clinton

SIDE EFFECTS
SOFTWARE

Table of Contents

1 History of Mantra Volume Rendering....3
1.1 Motivating Factors....3
1.2 Limitations....3
1.3 Volume Renderer Design....4
2 Volume Geometry....5
2.1 Volume Procedural Interface....5
2.2 Rendering Primitives for Volumes....6
2.3 Voxel Grid Storage....7
3 Rendering Algorithms....9
3.1 Acceleration....9
3.2 Ray Marching....10
3.3 Microvoxels....11
3.3.1 Splitting and Measuring....12
3.3.2 Shading Grid Generation....13
3.3.3 Sampling....14
3.3.4 Motion Blur....16
4 Shading Algorithm....18
4.1 Parameter Binding....18
4.2 Shading Context....19
4.3 Shader Writing....19
5 References....24

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          getWeightedBoxes(RefArray<BoundingBox> &boxes,
                                           FloatArray &weights,
                                           float radius,
                                           float dbound) 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 getWeightedBoxes() 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

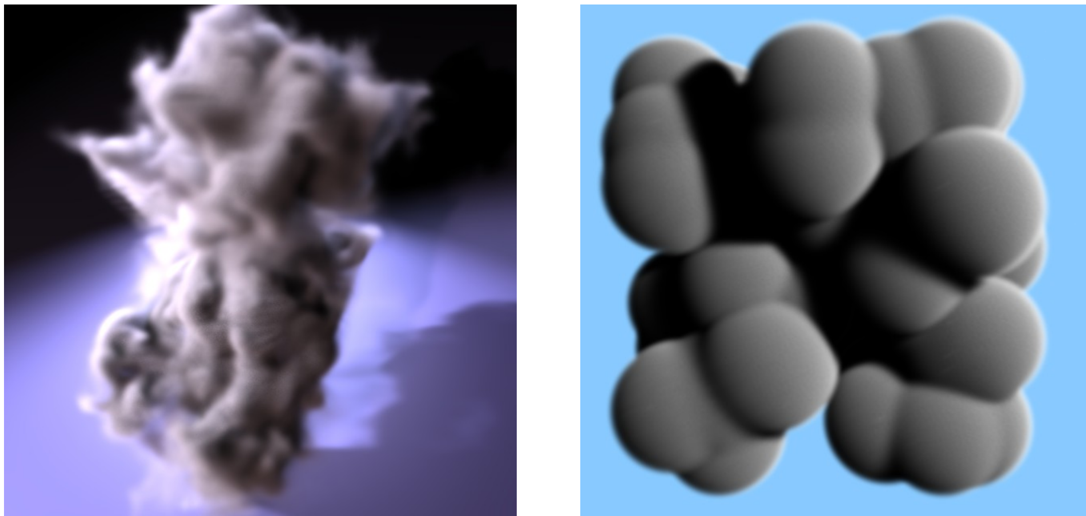


Figure 2: Voxel grid (left) and metaballs (right)

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 provide the trivial filtered evaluation operation for evaluation of fields. The generation of boxes for acceleration makes use of a voxel mipmap to hierarchically determine blocks of occupied voxels that can be represented by a single bounding box. This minimization of boxes helps to reduce the complexity of the eventual acceleration data structure. Boxes are also enlarged to account for the known filter width that is used for evaluation, and for the displacement and velocity bound.

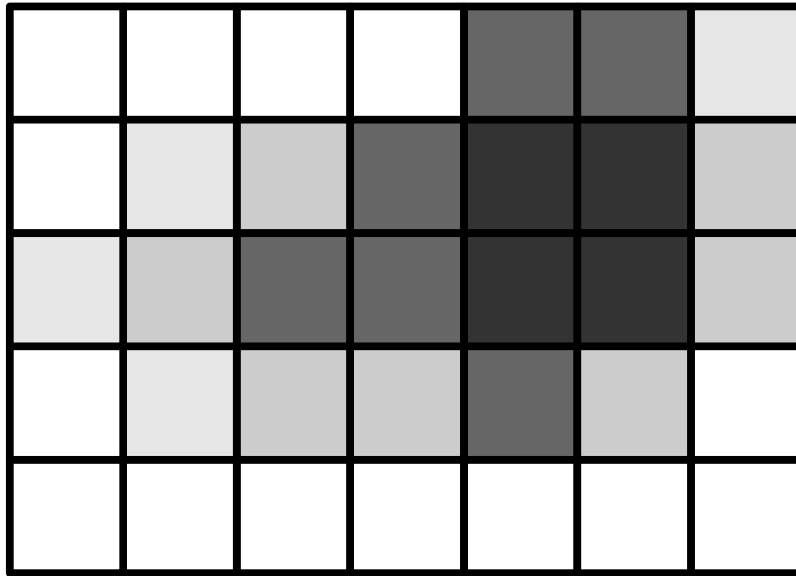


Figure 3: Example voxel field, with white empty voxels

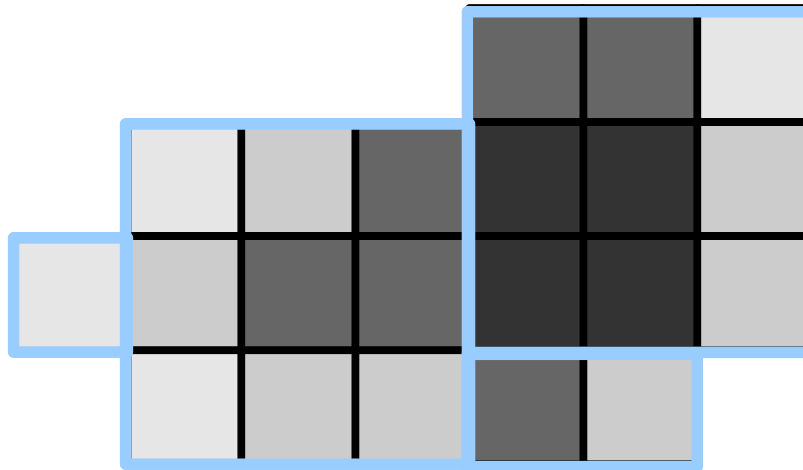


Figure 4: Possible clustering of 2D voxels (gray) into representative boxes (blue)

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.

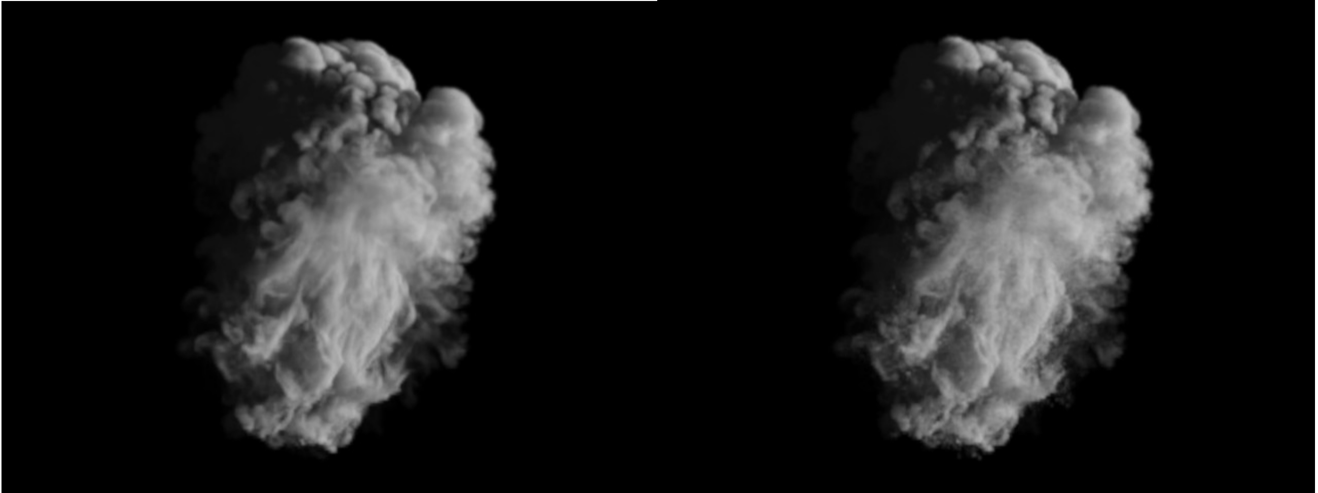


Figure 5: 5-bit ordered dither *Figure 6: 1-bit ordered dither*

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 Acceleration

The volume API provides a list of bounding boxes for the volume to describe the space that is occupied by the volume. These boxes are similar to the boxes that would be provided for surfaces, and are used to construct an acceleration data structure that is analogous to the one used for ray tracing acceleration for surface intersections.

Mantra uses a KD-Tree data structure 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 identical 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 defining the volume. 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.

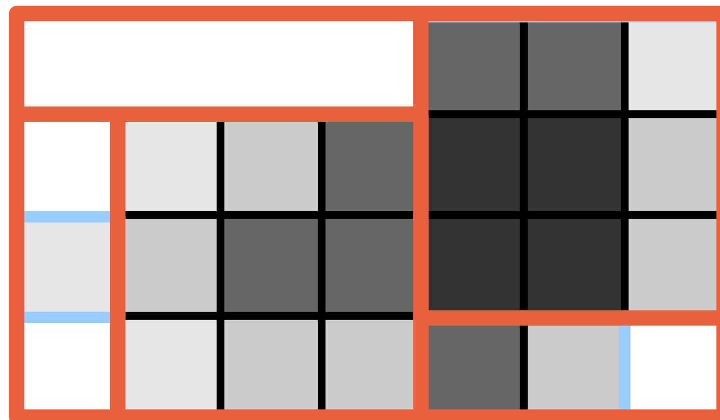


Figure 7: KD-Tree partitions (red) constructed on representative boxes (blue)

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 will contribute little to the image.

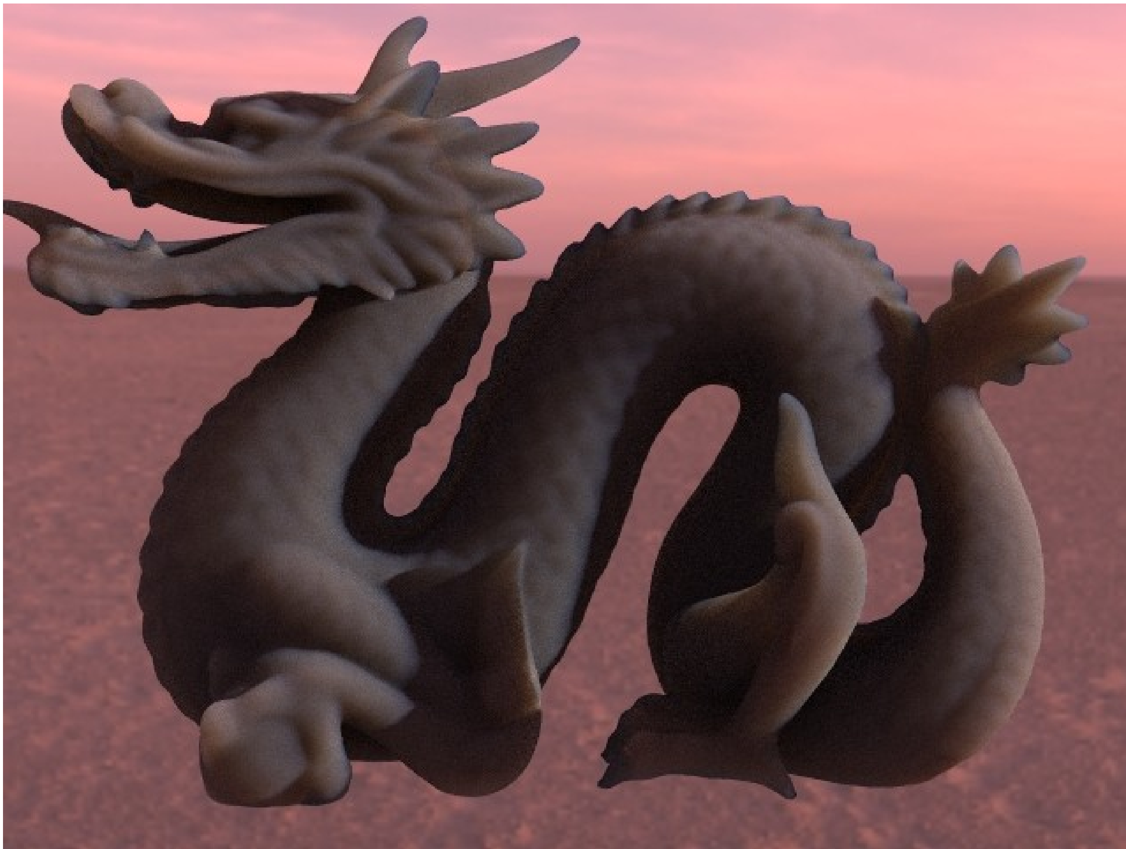
For volumes that are known to have a uniform density, samples can be distributed ideally based on the known density. Volume extinction follows an exponential falloff curve, meaning that if samples are distributed according to the exponential function they will each have equal weight. Samples can be distributed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

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

Now randomly distribute c(x) between 0 and 1 to generate exponentially distributed distances.



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 points where the shader acts are defined differently – in the case of micropolygon rendering, shaders are normally executed on subdivided meshes.

In the case of 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 format of 3D grids.

3.3.1 Splitting and Measuring

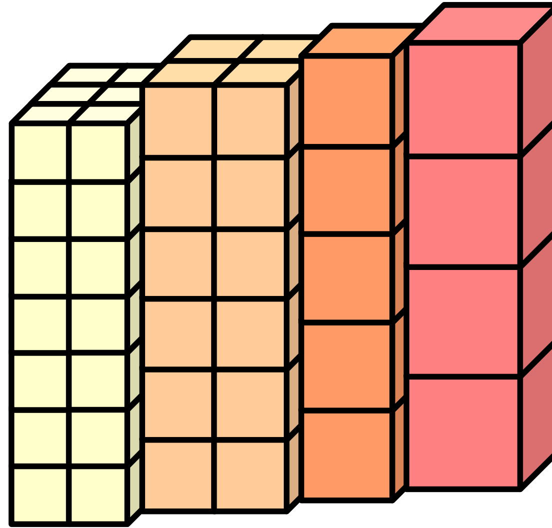


Figure 9: View-dependent dicing - viewer on left

A micropolygon renderer makes use of one or more metrics to measure 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

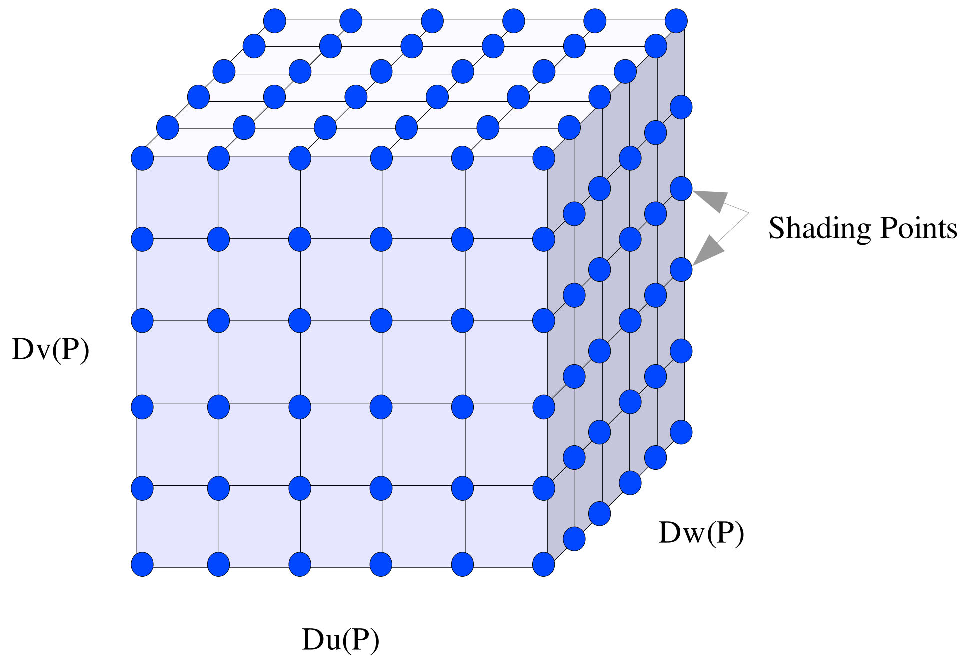


Figure 10: Cube of 216 shading points

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 with adjacent shading grids.

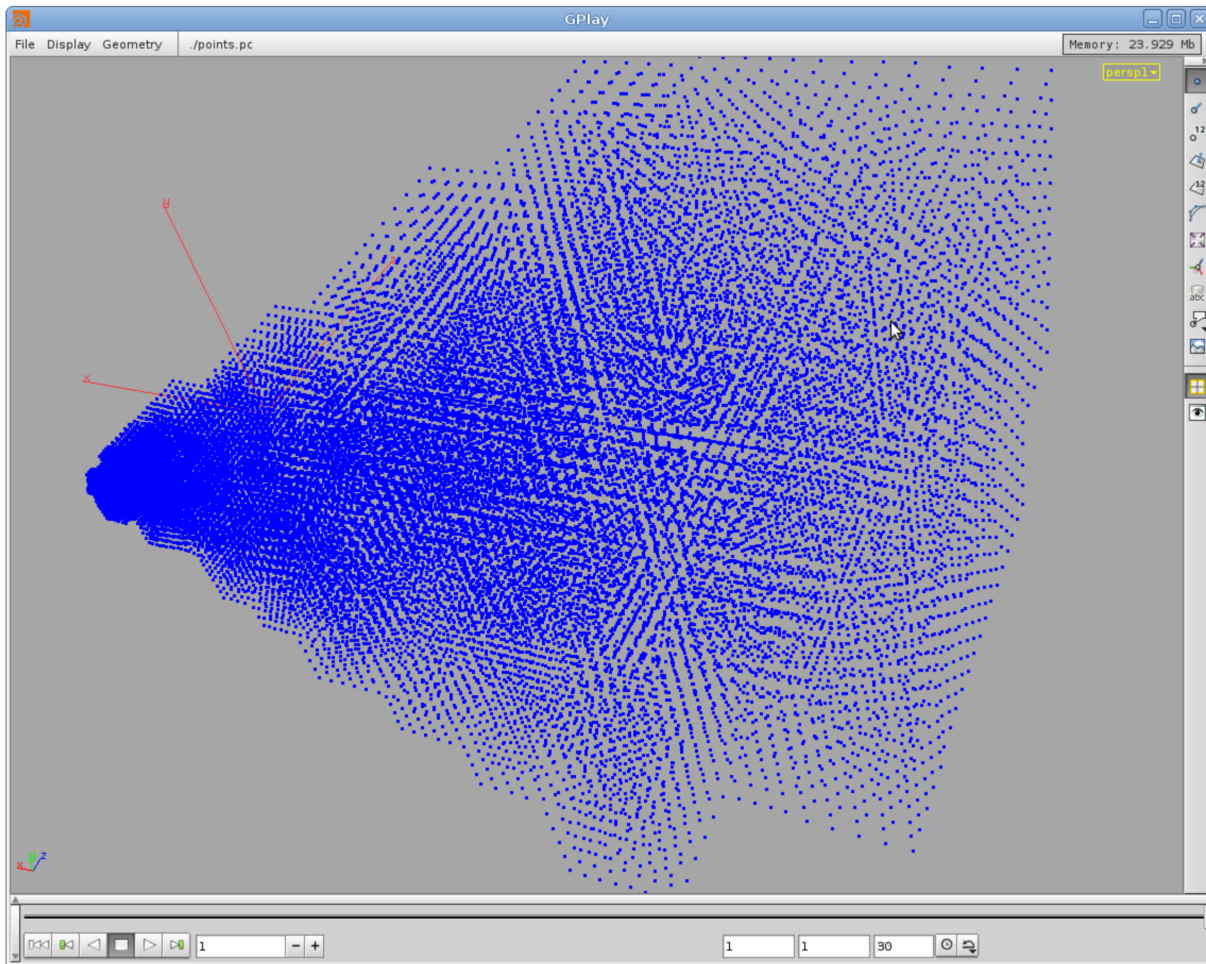


Figure 11: Actual cubes of shading points generated by a render. The viewing frustum originates on the left of the image

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 12: Render of a motion blurred volume

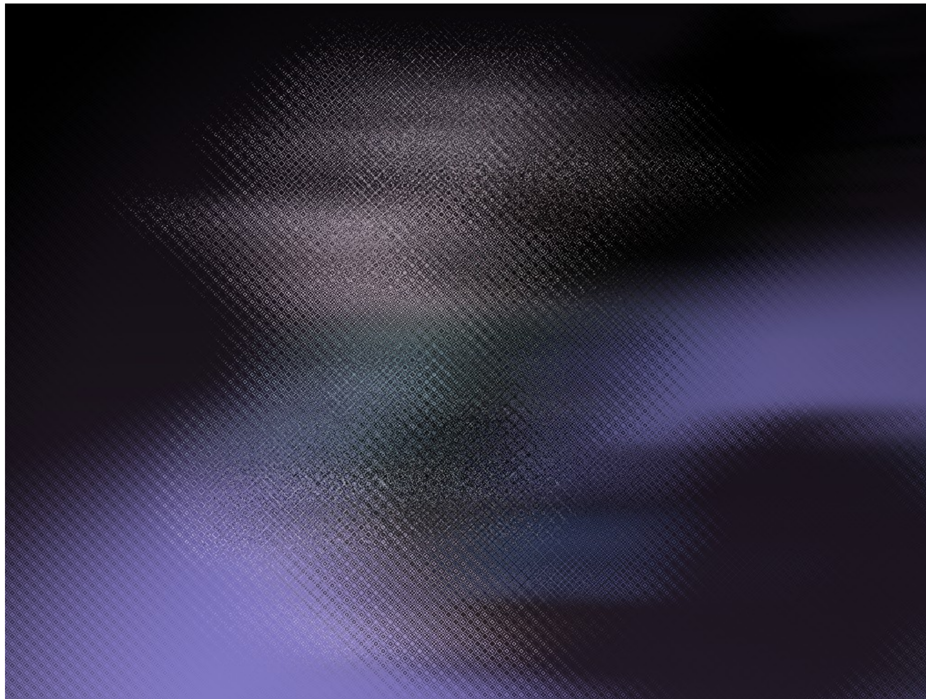


Figure 13: Sub-pixel sampling patterns for a motion blurred volume, 16x16 pixel samples

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 be applied on the generated shading grids while sampling continues to execute on the 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.

Decoupled shading offers one significant advantage. 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().

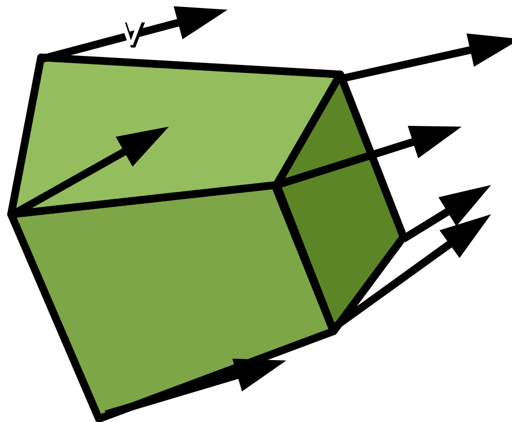


Figure 14: Velocity blurred voxel

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.

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 new global variable, dPdz, indicates the distance within the volume that should be composited by the shader.
- A new 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.

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.

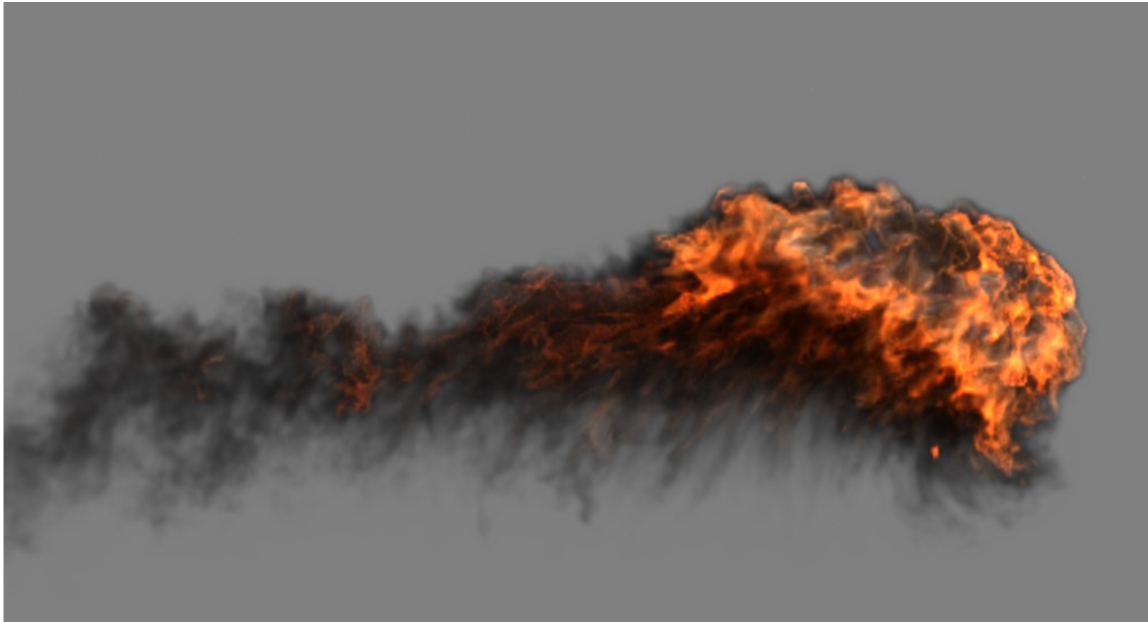


Figure 15: Fireball effect generated with "Pyro" shader

Mantra also supplies 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.

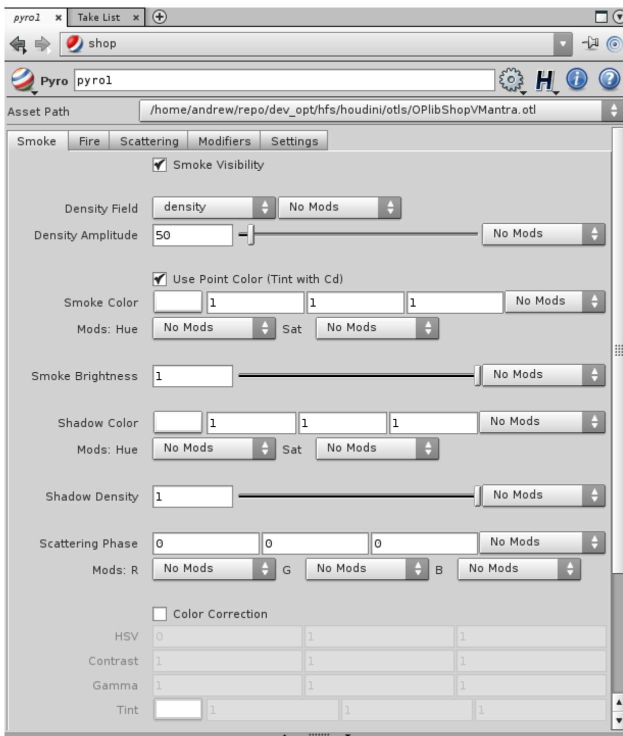


Figure 16: Pyro smoke tab - choose the fields to use for rendering smoke and configure modifiers

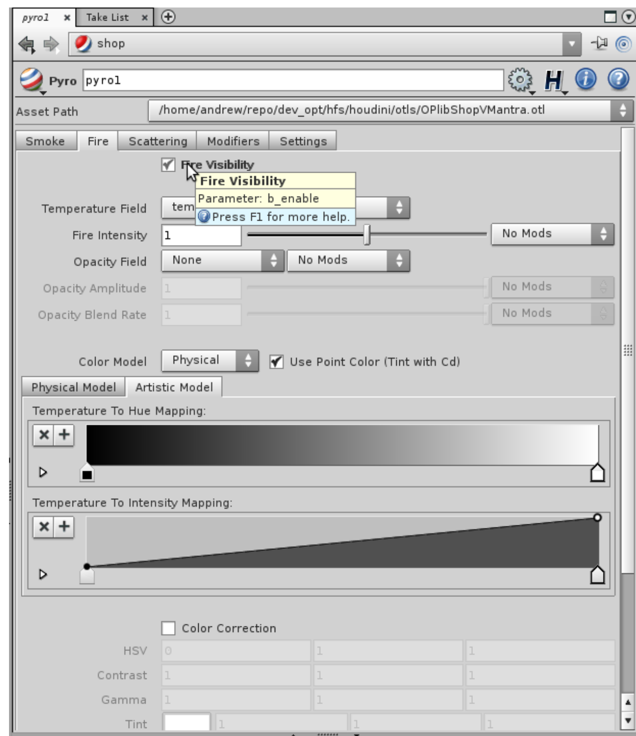


Figure 17: Pyro fire tab - choose the fields to use for rendering fire (emission) and configure modifiers

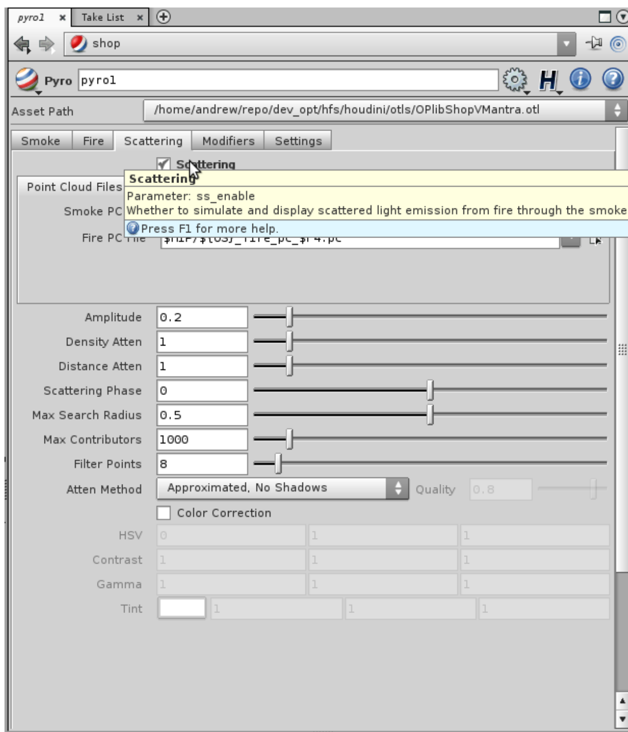


Figure 18: Pyro scattering tab - configures multiple scattering calculation using point clouds

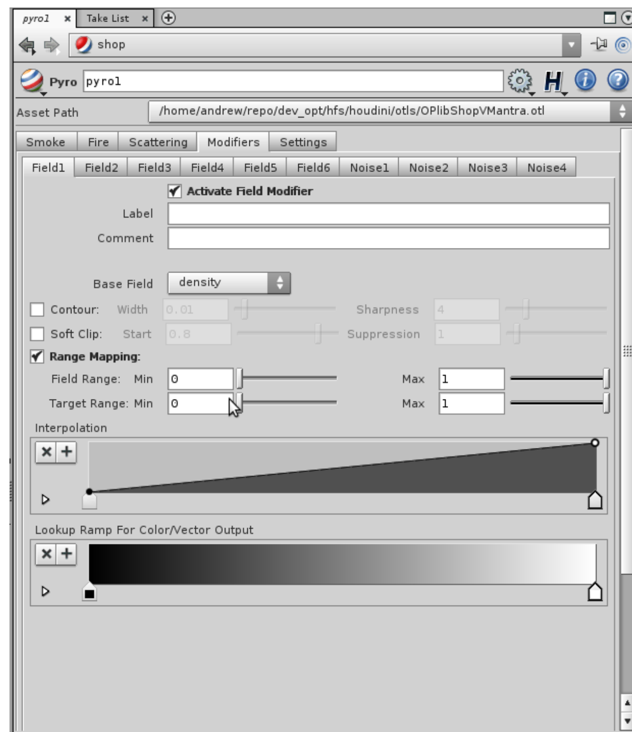


Figure 19: Pyro modifiers - configures modifiers (for example, noise) that can be used to adjust fields for rendering

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. *2I*, 4, 95-102.
[4] Clinton, A., Elendt, M. Rendering volumes with microvoxels (1999). SIGGRAPH 2009 Talks.

Volumetric Modeling and Rendering

Devon Penney
PDI/DreamWorks Animation

Contents

Introduction 3
Volumetrics Library 3
  Motivation 3
  Grid Spaces 4
    index 4
    unit 4
    world 5
  API Design 5
    `AbstractField` 5
    `ActualField` 6
    `FieldProxy` 6
    `Indexer` 6
    `Interpolator` 7
    `File` 7
  Levels of Control 7
  RLE Compression 8
Rendering 9
  Motivation 9
  API Design 9
    `RenderSettings` 9
    `RenderContext` 9
    `RenderComponent` 10
    `View` 10
    `Volume` 11
    `Shader` 11
    `Output` 11
    `Render` 12
  Empty Space Traversal 13
  Multiple Volumes 13
  Light Caching 14
  Motion Blur 18
    Algorithm 18
    Internal Motion Blur 20
    2D Motion Blur 22
References 24

Introduction

Prior to 2008, there was significant infrastructure for rendering and processing voxel grids at DreamWorks Animation, but these solutions were end result focused, written in the heat of production. This prompted a reworking of all volumetric modeling and rendering tools with the hope that a clean interface would enable more people to write useful volumetric operators and shaders. This process involved a redesign and implementation of the voxel grid container library, as well as a new standalone volume renderer. This overhaul has resulted in an explosion of the number of volumetric tools, and the number of effects that use them. The results can be seen in recent films such as *How to Train Your Dragon* and *Shrek Forever After*.

Volumetrics Library

Motivation

We used our prior experiences in voxel grid library design to develop a replacement called "Multifield" (MF). The cornerstone of MF is a clean C++ API that utilizes common design paradigms, and provides python bindings to allow for easy pipeline integration. It can store and manipulate a variety of grid types such as MAC grids, and float, half and double precision buffers, and frustum-shaped grids, all of which was functionally split among several similar libraries before. Lastly, we required a fast and memory efficient sparse representation of the voxel data. A previous library used an octree representation, which suffered from slow random lookups despite a favorable memory footprint. Thus, we decided to implement run length encoded volumes as a faster replacement.

Grid Spaces

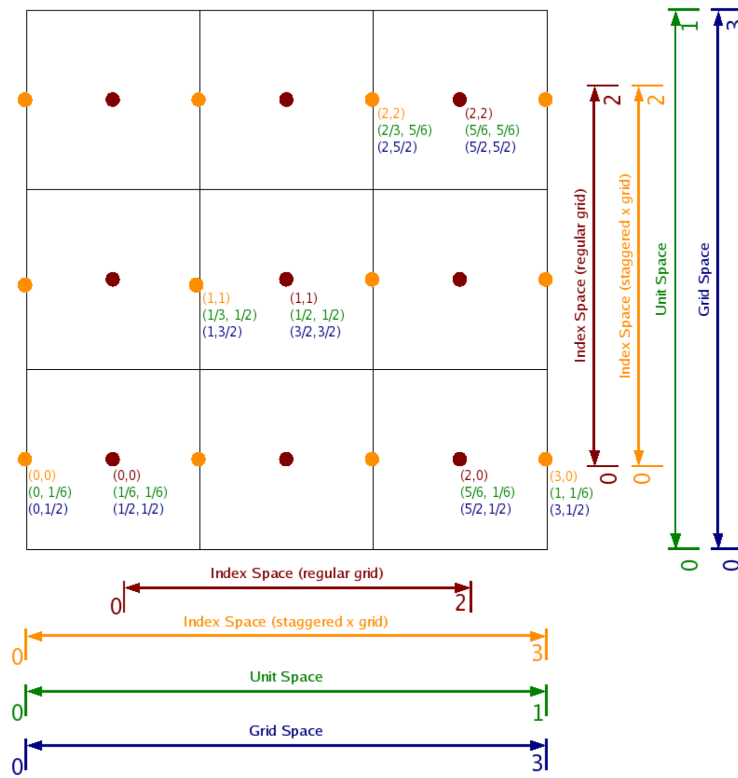


Figure 1. A 2D example of grid spaces for a regular cell-centered grid and a staggered grid.

Different algorithms necessitate operation in different grid spaces, which are ways of referring to the values stored on a grid. We have four spaces defined on MF grids (also called fields): index, grid, unit and world. Conversions between spaces are defined with methods such as `indexToWorld` and `worldToUnit` which enable algorithms to work in their natural space while maintaining code readability. In addition, we support interpolation in all of the grid spaces in the form of methods such as `sampleWorld` and `sampleUnit`.

index

Index space is defined as 0 to the number of voxels in each dimension minus 1, like an array in C or C++. This corresponds to the index of a grid cell, similar to the (x,y) coordinate of a pixel in an image. This space is useful when doing image processing-like operations on volumes, such as convolution (i.e. blurring).

unit

Unit space is parameterized as 0 to 1 in each of the three dimensions, and is often referred to as "local" space in a similar fashion to geometry. This is useful for operations where you need to work on the grid in a manner that is resolution independent. One example is what we call "barndooring" for volumes where you attenuate grid values away from the center of

a grid on each face. In this case, you express the extents of the falloff in terms of a percentage of the grid, which is exactly unit space coordinates.

world

World space defines the location, orientation and size of a volume in space. When a volume is rendered in a shot, interpolation is done using world space coordinates. This transformation is usually controlled by a 4x4 matrix which converts world space positions to unit space and vice-versa. In addition, we support frustum transformations, which replace the 4x4 transform by a routine that gives voxels a truncated pyramidal shape.

API Design

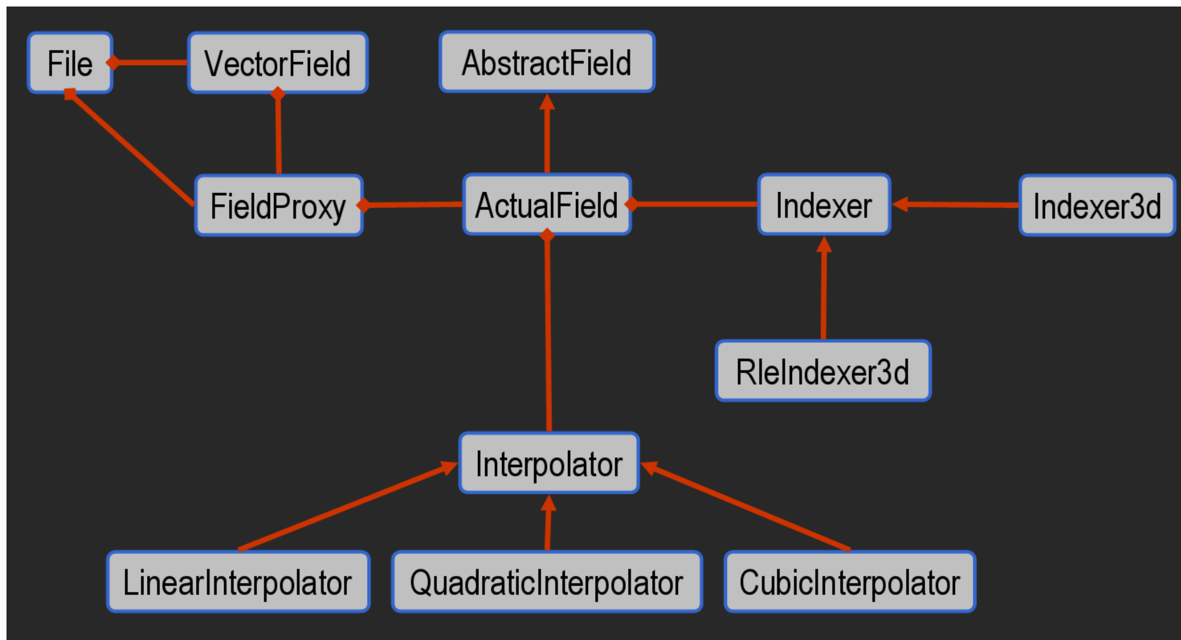


Figure 2. A simplified UML diagram for the MF library.

The core of the MF library is a voxel grid container that we call a field, which is ostensibly a wrapper around a 3d array. Also, there is a File object that represents a collection of grids with a mapping between string names and fields. Below are the key classes involved with the MF library.

`AbstractField`

This is the base class that all field classes inherit from. It defines all of the behaviors of a field that are agnostic to the data type stored within the grid. It contains the 4x4 transform used to convert grids from unit (local) space to world space coordinates. In addition, there are many abstract methods common to grids, thus ensuring a common interface for different containers to inherit from. Lastly, it has the functionality necessary for frustum grid transformations. Important methods include:

`Vector3 indexToUnit(Vector3 input)` - converts a point from index to unit space.

`Vector3 unitToWorld(Vector3 input)` - converts a point from unit to world space.

`Matrix getTransform()` - returns the 4x4 transform that converts unit space coordinates to world space.

`abstract void compress()` - compress a grid as implemented in subclasses.

`unsigned int sizeX()` - get the number of voxels in the x dimension.

ActualField

`ActualField` is a subclass of `AbstractField`, and is templated according to the grid data type it is storing (T). It contains the data for the grid in an array of the template type. In addition to inheriting all the functionality of `AbstractField`, it defines getters and setters for dealing with the grid. Important methods include:

`T getValue(int x, int y, int z)` – get the value at a index space location.

`void setValue(int x, int y, int z, T val)` - set the value at an index.

`T sampleWorld(Vector3 worldPos)` - interpolate to derive a value at a world space location. Variants of this method exist for all grid spaces.

`T* getPtr()` - retrieves the underlying array.

FieldProxy

A `FieldProxy` acts as a proxy object for an instance of an `ActualField`. It wraps every method available in an `ActualField`, and allows for a type agnostic way of dealing with grid data. This entails a speed penalty due to the fact that each method must resolve the stored data type at run time by using a switch statement to iterate over all supported grid types. While this speed penalty causes a noticeable hit in performance, it allows developers to write code that supports arbitrary bit depth float fields without any extra work.

Indexer

`Indexer` is a superclass for a group of objects that map (x,y,z) tuples for integer-valued index space positions to linear indices pointing to the voxel's value in an array. The base class does a scan line ordered mapping into a dense 3d grid. We subclass it to define the necessary transformation to support run length encoded volumes. Additional subclasses could be derived to define other compression schemes, or different voxel storage ordering mechanisms such as Hilbert curves. Important methods include:

`int getIndex(int x, int y, int z)` - maps an index space position to a linear index.

`int getNumDefined()` - returns the number of defined voxels. For sparse grids, this is significantly less than the total number of voxels (`sizeX*sizeY*sizeZ`).

Interpolator

`Interolator` is a base class for a module that can interpolate grid values. Subclasses implement various algorithms such as tricubic, triquadratic, and trilinear interpolation. An `ActualField` has a member Interpolator instance, which using the strategy pattern can be modulated at run time, thus allowing for a dynamically changing interpolation scheme. We found that triquadratic interpolation offers the best tradeoff between speed and quality for many uses including rendering. It has a single abstract method that must be implemented:

> `T interp(int x, int y, int z, Vector3 pos)` – This method takes an index and position within a cell and returns the interpolated value.

File

The `File` object offers a mapping between string identifiers and `FieldProxy` objects. This is what is stored and loaded from disk, and is the main interface for manipulating a collection of grids. Important methods include:

> `FieldProxy getField(string name)` - Gets a field given its name.

> `void addField(string name, FieldProxy field)` - Adds a field with a given name.

Levels of Control

MF is designed with the notion that it should be possible to write extremely optimized volumetric operators for experienced programmers, yet it should also be easy for inexperienced artists to write code without worrying about details or optimizations for simple production-specific tools. This "level of control" architecture is a foundation of the API design, and manages the balance between code that is fast to execute, and code that is fast to write.

The `FieldProxy` class is one feature of the "levels of control" architecture. It allows for a grid to be loaded from disk, and manipulated in a manner that is agnostic to the stored data type. All operations are exposed as manipulating grids that contain double precision floating point numbers, but values are eventually cast to the stored type when appropriate. This is useful since it is often advantageous to keep some data at half precision whereas other things are better stored as floats. Such optimizations can allow for higher resolution grids and more detail in volumes. This class is the preferred interface for production-specific C++ tools since it is fastest and easiest interface to deal with fields. To contrast this, R&D developers will often use `ActualField` objects when the overhead of `FieldProxy` seriously impacts performance.

Python bindings represent the highest level of control available in our system. We wrap the `mf::File` object as a dictionary that maps string field names to Numpy arrays. The numpy arrays refer to memory allocated by `ActualField` objects. This allows for artists to utilize the extensive toolset available in Numpy and Scipy to manipulate voxel grids. While

performance concerns often arise when extensive looping is required, this workflow significantly decreases the time for new tools to be written.

Grid accessors are used to query grids for values at voxel indices and to interpolate for arbitrary positions inside the volume. We utilized the "levels of control" principle with such methods by allowing users to circumvent bounds checking and other slowdowns in cases where it isn't needed. In addition, `FieldProxy` objects are not used when the grid type is known (for instance with a simulator), which gives an additional speed boost.

`ActualField::getUncompValueFast(x,y,z)` `ActualField::getPtr()`	Must know the stored data type, no bounds checking, assumes grid is uncompressed	Highest performance, lowest ease of use
`FieldProxy::getValue(x,y,z)`	Bounds checking (reflect, wrap, extend edge, etc), agnostic to data type, works with compression	
python Numpy arrays	Python interface	Lowest performance, easiest to use

Figure 3. Comparison of methods for accessing a field value.

RLE Compression

Run length encoding (RLE) is a compression scheme that tracks sequences of repetitious values called "runs". For volumes, we use RLE to compress values that are sufficiently close to zero [Houston et. al. 2004]. For a grid of size (X,Y,Z), we build a table of size X*Y where each entry stores a list of the runs in z. A run is composed of its type, length and data index. A run's type is categorized as either "defined" (non zero) or "undefined" (close to zero). Defined values are stored in a linear array and undefined values are not stored at all. The linear index in the run is used to determine where the "defined" voxel value is stored in the linear array. The storage is reduced to O(X*Y+D) where D is the number of defined voxels as compared to O(X*Y*Z) for a dense grid. The complexity of a random lookup is O(average number of runs in one scan line in Z) due to the necessary binary search.

RLE compression is exposed through two methods on our field class, `compress()` and `uncompress()`. Methods for accessing and setting grid values utilize a binary search instead of the typical 3D array indexing as is common with dense voxel grids. To build compressed grids, we offer a `FieldWriter`, which applies compression as the grid is being built, and doesn't require building the entire dense 3D grid. This works by maintaining a list of (voxel index, value) pairs, which are then compressed into a grid data structure without ever allocating for the dense voxel grid.

We found that there are several key difficulties when using RLE encoded volumes in a production setting. It is difficult to have an efficient way of building sparse grids, as opposed to encoding an already dense grid. The method our library implements is slow,

and requires that set voxels can't be accessed until the entire grid is created. This eliminates the ability to build incrementally updated grids, which incurs a significant limitation on the usage. Also, RLE grids have memory limitations since it needs to store the X*Y grid of runs as well as a data structure to store the run information. In addition, once a grid is RLE compressed, there is a very limited set of operations one can perform to modify the grid. If any voxel goes from being defined to undefined or vice versa, it requires recompressing, which is an expensive operation proportional to the grid resolution. Thus, RLE compressed grids are essentially read only for most cases.

Considering these limitations, we found utility in using RLE for rendering large numbers of grids simultaneously. In this case, each grid is RLE compressed as it is read into memory utilizing a fast dense grid to RLE conversion process. This increases the effective number of grids and grid resolution we were able to render for shots where it isn't feasible to do many separate render passes. Also, with most well-behaved volumes, the number of runs in a single scan line in Z is rather low, which means the runtime for a single random lookup is ostensibly constant. This mitigates the damage to render time interpolation performance, which requires a significant amount of random lookups.

Rendering

Motivation

We decided to implement a standalone renderer instead of trying to shoehorn the replacement into an existing solution. This allowed for maximal flexibility and freedom to develop an ideal solution. We designed a flexible shading system that made it easy to write new shaders with a minimal amount of code, thus hiding the complex details of ray marching. In addition, we made sure the new rendering solution scaled with complexity and was not plagued by problems with memory usage since it was imperative that we could render as large of a grid as possible. Lastly, we directly supported rending multiple volumes and 3D and 2D motion blur thus allowing for a feature set unaccomplished by a single prior solution.

API Design

The foremost goal of the API was to separate ray marching, volume sampling, volume shading, and output into separate components. This design allows for a large level of flexibility for shader development as well future low level improvements.

`RenderSettings`

This contains values for all of the initialization variables to the renderer. Examples are the ray marching step size, image size, toggle for motion blur, whether to use light caching or not, etc. It is used to initialize state in the various components of the renderer.

`RenderContext`

This is the primary construct used by objects such as shaders to retrieve values during ray marching. It has methods such as:

`Vector3 getWorldPt()` - returns the current world space location being sampled.

`double getFieldValue(FieldId id)` - queries a field for the current step position. FieldId is a unique identifier for a field in a volume.

`Vector3 getVectorFieldValue(FieldId id)` - queries a vector field for the current step position.

`Vector2 getPixelXY()` - returns the pixel coordinate for the current ray.

`Vector3 getRayDir()` - returns the world space direction for the current ray.

`double getStepSize()` - returns the current step size for ray marching.

This class is populated by the `Render` object, which is responsible for ray marching.

RenderComponent

`RenderComponent` is the base class for most objects in the renderer. It has very little functionality itself, but imposes a structured interface by which objects can be utilized within the context of ray marching. Most importantly, this class defines several abstract methods which are called by the `Render` class (described below) at specific points during the render.

`void preFrame(const RenderSettings &s)` – called before the frame starts being rendered, and is used to initialize data used by the component.

`void postFrame(const RenderSettings &s)` - called after the frame is finished, and is used to cleanup member data of the component.

`void preRay(const RenderContext &ctx)` - called before ray marching is initiated for a given ray.

`void postRay(const RenderContext &ctx)` - called after a ray is finished marching.

`void preStep(const RenderContext &ctx)` - called before a step along a ray is taken.

`void postStep(const RenderContext &ctx)` - called after a step along a ray is taken.

The following three classes all derive from `RenderComponent`

View

`View` is the superclass for anything which rays can be cast from. Subclasses must define transformations from world space to the view's local space as well as local space to raster space for the output image. There are two implemented subclasses, one for a render camera and another for lights. Ray casting through lights is used for depth map generation. The most important methods are:

`Vector3 getWorldSpaceRayDir(const double pixelX, const double pixelY)` - gets the ray direction for a pixel.

`Vector3 getWorldSpaceRayStart(const double pixelX, const double pixelY)` – gets the starting position for a ray given a pixel.

Volume

`Volume` subclasses are responsible for taking world space positions and producing field values such as density, temperature, or any arbitrary field. They don't necessarily need to be grids, but the most common one is the `MfVolume`, which contains an MF File object as described in the first section. Their most important methods are:

`void preStep(const RenderContext &ctx)` - Given the world position referred to for the current step, we cache the index space position to reduce matrix multiplication when interpolating multiple fields.

`double getStepFieldValue(FieldId id, const RenderContext &ctx)` - Using the index space position derived in preStep, we interpolate to return a grid value.

`bool isOccupied(const RenderContext &ctx)` - This is used for empty space traversal to determine if the current step has non zero density.

`Vector3 worldToUnit(Vector3 worldPos)` - converts a position in world space to unit space for a volume.

Shader

`Shader` instances are responsible for using the `RenderContext` to get values obtained from the `Volume` to produce a color and opacity for a given step through a volume. This is the most commonly overridden class, and they are written by effects developers for shot-specific shading solutions. It has two methods of note:

`Vector3 calcLighting(const RenderContext &ctx)` – The parent class defines a light loop here, but it can be overridden. Abstracting out lighting calculations into their own method enables the ability for light caching.

`Vector4 shade(const RenderContext &ctx)` – This applies the Beer-Lambert law based on the step size and calls `calcLighting(…)` to return a color and opacity for a volume segment.

Output

`Output` subclasses accumulate values from the `RenderContext` at either a per-step or per pixel rate, and outputs the results to various files. Typical output targets include raster images, camera depth maps, velocity images, and deep shadow maps. Their important methods are:

`void accumPixelSample(RenderContext &context)` - Called once per pixel to accumulate values. This is used to output files such as raster images.

`void accumStepSample(RenderContext &context)` - Called once per step to accumulate values. This is used to output files such as deep shadow maps.

Render

The `Render` object is the glue that puts all of the previous classes together. It iterates over pixels, determines which rays to march, and performs ray marching. Ray marching has the following pseudocode that uses the aforementioned classes to produce an accumulated color and opacity.

```
Vector4 marchRay(Ray theRay)
    double t0, t1
    if (!volume.intersects(theRay, &t0, &t1))
        call postRay on all components
        return black
    accumColor = black

    call preRay on all components
    for currT in [t0, t1]

        // early ray termination
        if accumColor is totally opaque
            postRay on all components
            return accumColor

        // get the position along the ray
        worldPosition = view.pointOnRay(theRay, currT)
        context.setWorldPos(worldPosition)
        call preStep on all components

        // empty space traversal
        if !volume.isOccupied(context))
            postStep on all components
            continue

        // shading and volume sampling occurs in shade(...)
        currStepColor = shader.shade(context)
        if (currStepColor has no opacity)
            postStep on all components
            continue

        // integration
        accumColor = accumColor OVER currStepColor
        context.setAccumColor(accumColor)

        // accumulate values in output modules
        call accumStepSample on all Output objects
        postStep on all components

    postRay on all components
    return accumColor
```

Empty Space Traversal

Empty space traversal is initialized by a preprocess in the `Volume` object's `preFrame`. We take the density field, and generate a boolean grid where each cell is one if it or any neighbor has positive density and zero otherwise. This amounts to dilation on the density field, which can become slow for large volumes, thus making caching a prudent idea. At render time, we take the world space position of a sample and convert it to an integer index space position. We query the empty space grid at this location without interpolation to determine whether the cell or any neighbor has density. This allows for quick trivial rejection of empty space while ray marching.

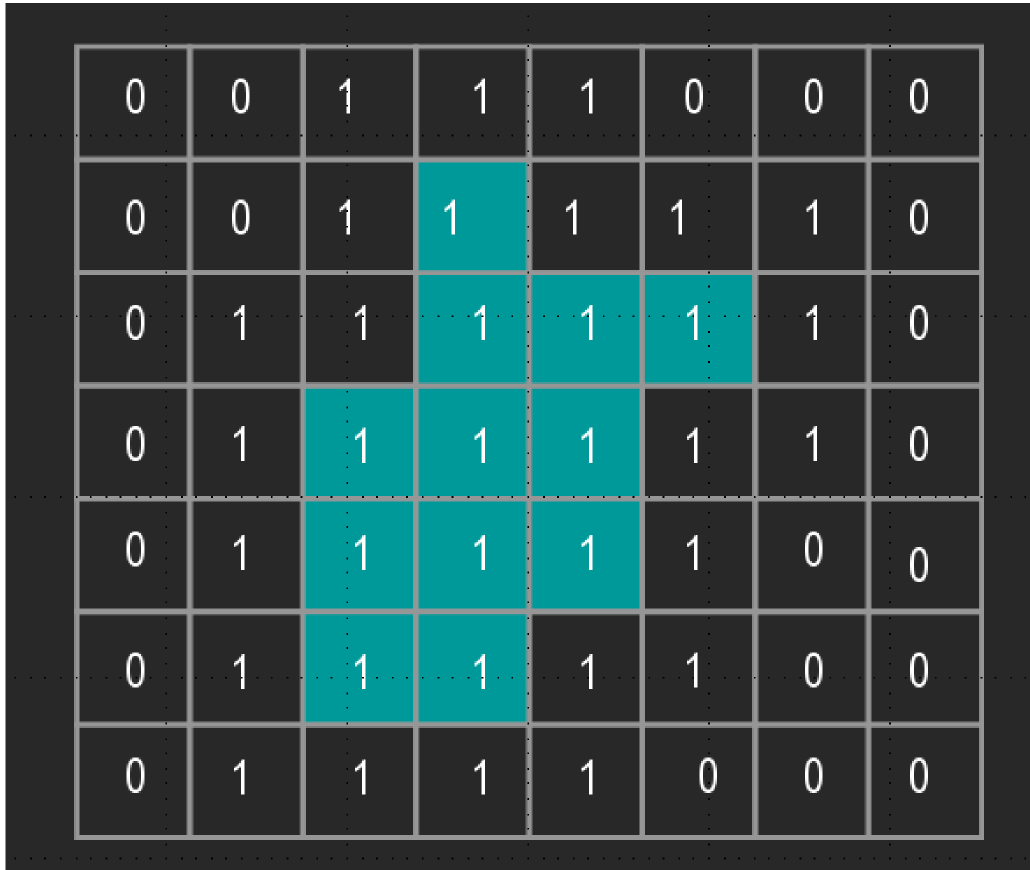


Figure 4. The teal volume has the corresponding empty space traversal grid represented by the ones and zeros.

Multiple Volumes

Rendering of multiple volumes is handled by introducing a new class called the `VolumeGroup`. It stores a list of all the volumes being rendered for the current frame, and in `preRay` will generate a list of all the volumes intersected. The `Render` object queries the `VolumeGroup` at each step for the currently rendered volume.

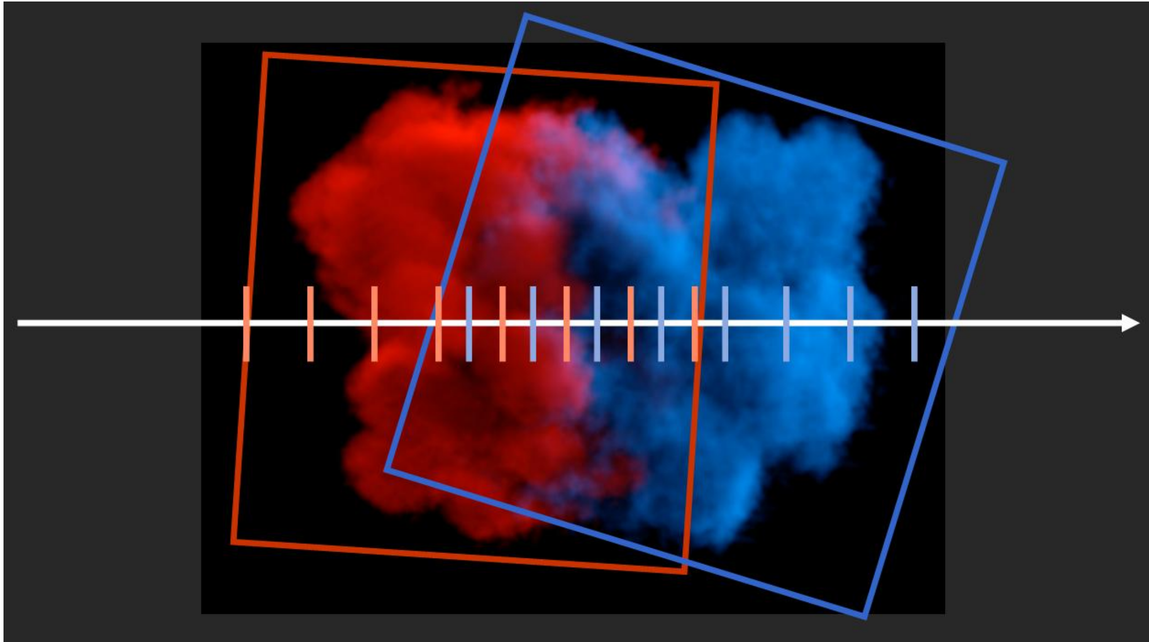


Figure 5. This example of rendering multiple volumes is discussed below.

Consider ray marching the white ray shown above. In regions where the two volumes overlap, we cut the step size in half, and alternate the volumes by using the `VolumeGroup`'s `preStep` callback. Even though we are ray marching at twice the sample rate as before, we compute Beer-Lambert with the same step size that is used in non overlapping regions. This effectively interleaves the samples to give the appearance of the two volumes overlapping in space. Note that in the general case, the spatial step size is divided by the number of overlapping volumes for a given region.

Light Caching

The time spent during shading is the bottleneck of volume rendering, so much so that it often far overshadows optimizations such as faster interpolation and empty space traversal. To mediate this, we use a grid-based light caching algorithm to reduce the number of shading samples. It is implemented by introducing the `LightingWrapper` superclass. `LightingWrapper` contains a pointer to the current shader, and has a single method to evaluate lighting for a given step. By default, `LightingWrapper` directly calls the `Shader`'s `calcLighting` method.

We have a subclass for light caching called `CacheLightingWrapper`. In `preFrame`, we initialize the a 16 bit vector field for color and an auxiliary grid for neighborhood calculations. The auxiliary grid is 1 when all neighboring cells for a given cell have valid lighting values and 0 otherwise. The bounding box for the grid is the same as the one being rendered, but often the resolution will be less. We expose a dial as a multiplier on the input grid resolution for the cache. Lower lighting grid resolutions result in faster renders at the cost of more rendering artifacts.

In calcLighting, we either interpolate to determine the current lighting based on neighboring cells, or calculate lighting at grid cells and interpolate for the result. This amounts to the following pseudocode:

```
Vector3 calcLighting(Vector3 sampleLocation)
    idx = lightingCacheVolume.worldToIndex(sampleLocation)
    if (!neighborGrid.getValue(idx))
        update colorGrid with lighting at neighbors the cell at "idx"
        neighborGrid.setValue(idx, true)
    return colorGrid.sample(sampleLocation)
```

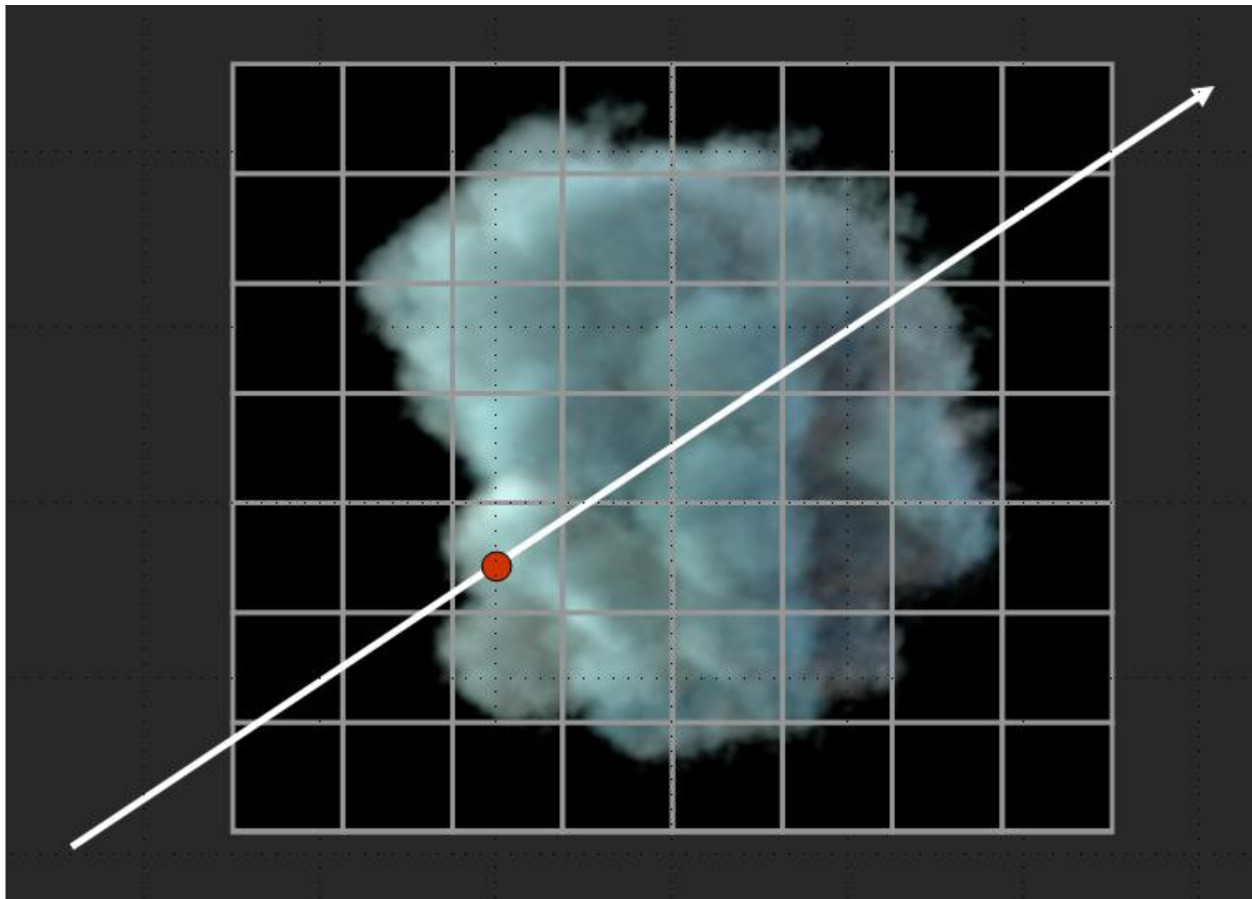


Figure 6. Ray marching with a light cache. We are calculating the lighting of the volume at the red dot's location.

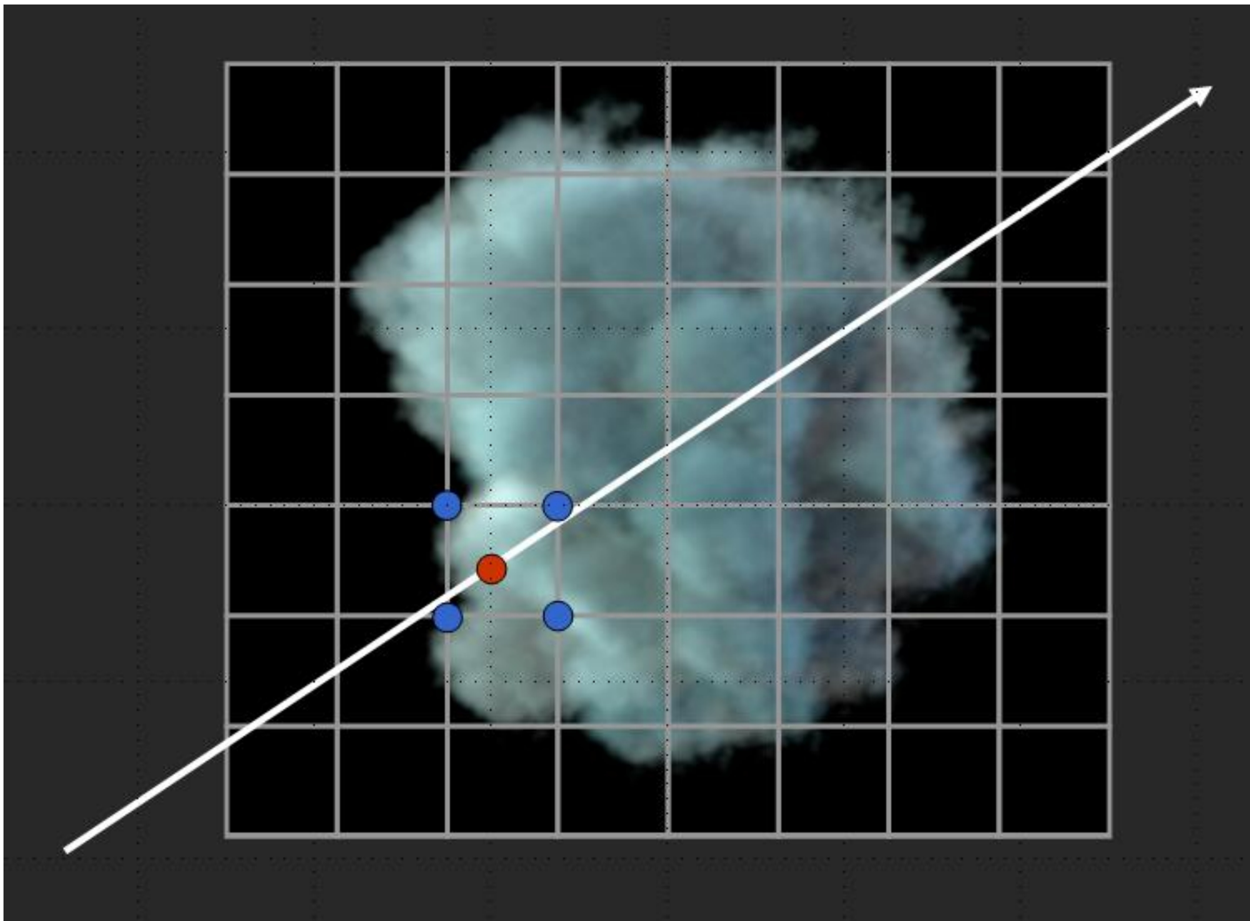


Figure 7. The cache is empty, so lighting is calculated at neighboring grid points and an interpolated result is returned.

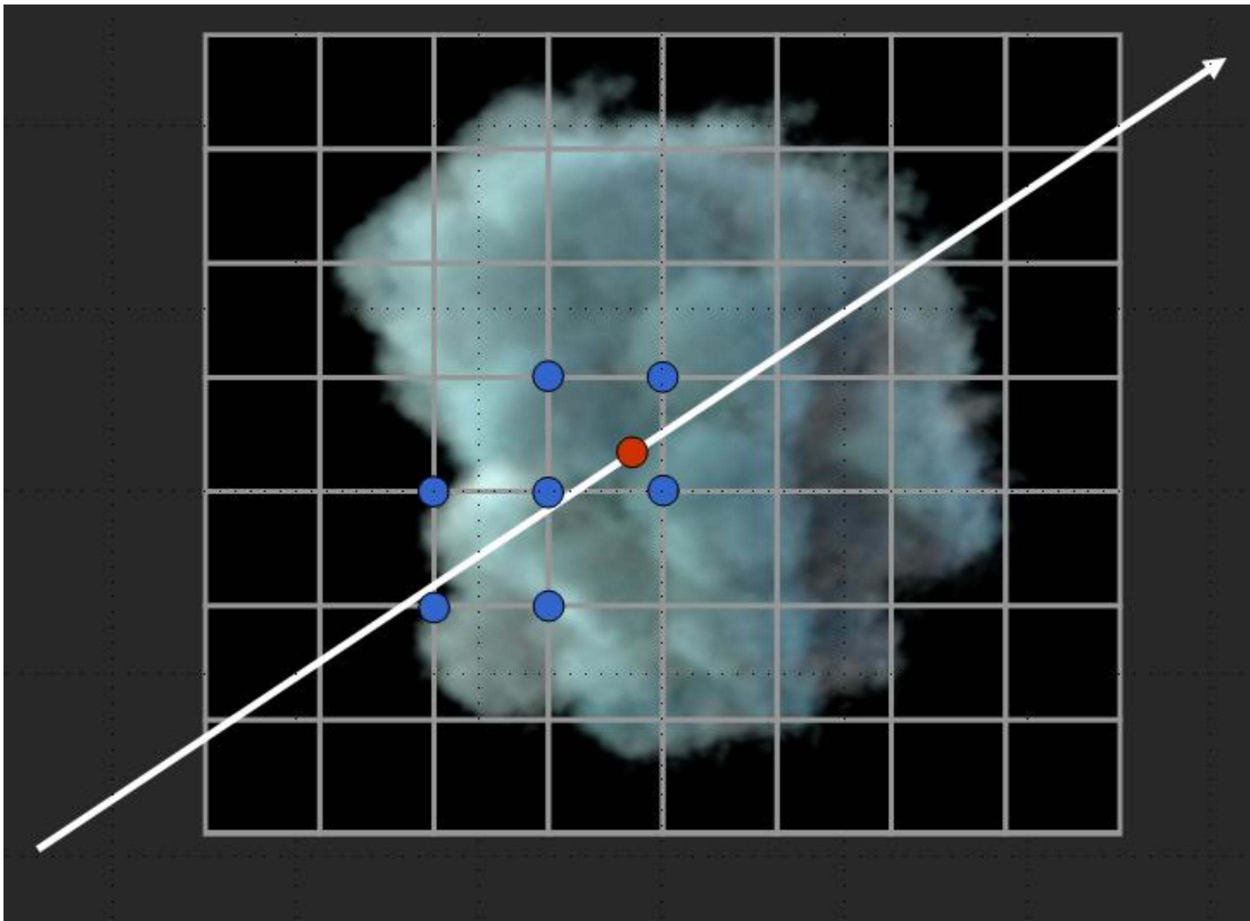


Figure 8. At the next ray step, only three new lighting calculations are needed since the bottom left one is in the cache.

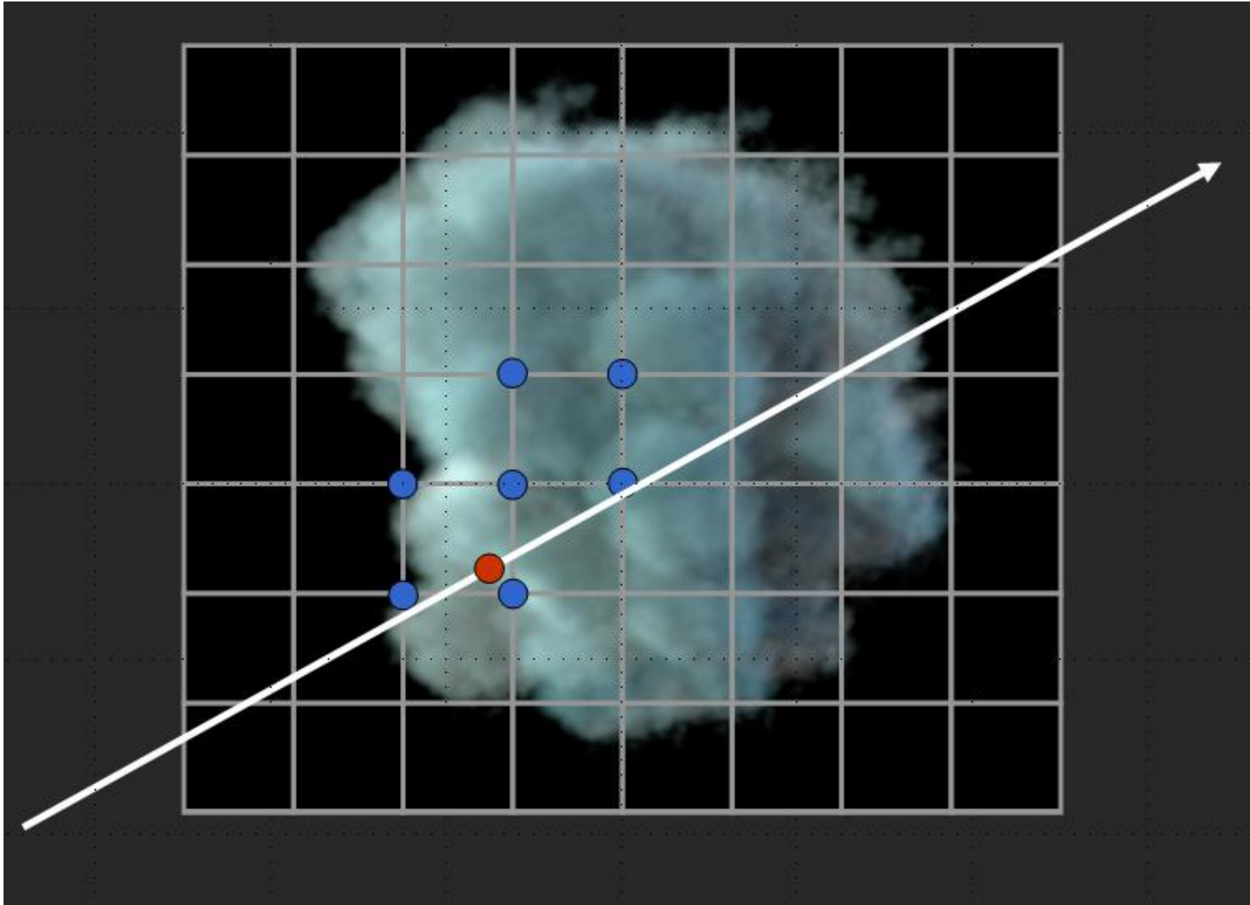


Figure 9. Consider ray marching a different ray. No lighting calculations are made at the red dot, and a lighting value is returned by interpolating based on cached values.

Motion Blur

Algorithm

In a simplified model without color, the goal of motion blur is to solve for the accumulated opacity across the shutter open interval, $[t_s,t_e]$ for the camera space depth range $[z_s,z_i]$ describing the intersection with the volume.

$$\alpha_i = \int_{t_s}^{t_e} \left(1 - e^{\left(- \int_{z_s}^{z_i} f(z,t) dz \right)} \right) dt$$

Figure 10. This integral describes opacity accumulation with motion blur as defined above. f(z,t) is the density function of the volume at depth z and time t.

We use a distributed ray tracing algorithm for motion blur by integrating the results of ray marching a predetermined number of rays, each with a time distributed on the shutter interval. This approach avoids the excessive memory consumption associated with other techniques such as particle smearing. However, this method suffers from poor performance due to an excess of shading calculations and interpolations.

First, we define three categories for motion blur contributions. First is camera motion, which is motion blur due to movement of the viewing camera. Second is container motion, which is motion blur from the movement of the transform stored on the volume itself. Third is internal motion blur, which is motion due to the stored per-voxel velocity. We allow each of the components of motion blur to be individually toggled since there are situations when container motion blur is baked into the per-voxel velocities, as well as other situations.

Define the frame being rendered as having time t_f, and the shutter on the interval $[t_s,t_e]$ where $t_s \leq t_f \leq t_e$. We cast n rays to march, each of which is assigned a time t_r where t_r is distributed between t_s and t_e. Each ray is marched on the interval $[z_s,z_e]$ expressed in camera space depth units as intersected against the motion blur envelope (described below). Values for accumulated color and opacity are derived and stored for each of the n rays for each z_i in $[z_s,z_e]$ depending on the ray marching sample. Next, for each z_i, we average the value across all the rays and store the result.

We must determine the motion blur envelope, which is a bounding box to intersect against in order to determining z_s and z_e, the camera space depth range for ray marching. This bounding box is different from the volume's itself since it is stretched according to the motion of the volume container, voxel velocities, and the camera motion. This is accomplished by sampling points within the volume and transforming them from t_f to t_a for t_a sampled on the shutter interval. We take the max and min of all these sample locations to determine the axis aligned bounding box.

For each step along a ray, we derive an offset transformation that allows us to sample volumes at t_r. This is done at each step to allow for jittering of the time between t_{n-1} and t_{n+1} to smooth out results and reduce biasing. To accomplish this, we must compose the transformation contribution from the three aforementioned motion blur types. Camera and container motion blur are handled in a nearly identical fashion by computing a 4x4 transformation matrix that transforms points at t_f to t_r. Internal motion blur uses an advection scheme [Kim and Ko 2007] as described in the next section.

Internal Motion Blur

We utilize the methods illustrated in Eulerian Motion Blur to account for internal motion of volumes in motion blur calculations. This is done using the following pseudocode to sample density t_r given density and velocity at t_f.

```
double densityAtTimeFToTimeR(Vector3 worldPositionAtTimeF, time r)
    velocityAtF = densityGrid.sampleWorld(worldPositionAtTimeF)

    // advect backward along velocity at time tf to determine vr
    velocityAtR = densityGrid.sampleWorld(worldPositionAtTimeF -
            (tr - tf) * velocityAtF);

          // advect backward along velocity at t_r to determine density
          return densityGrid.sampleWorld(worldPositionAtTimeF - (tr - tf) *
            velocityAtR);
```

The algorithm can be visualized with the following example.

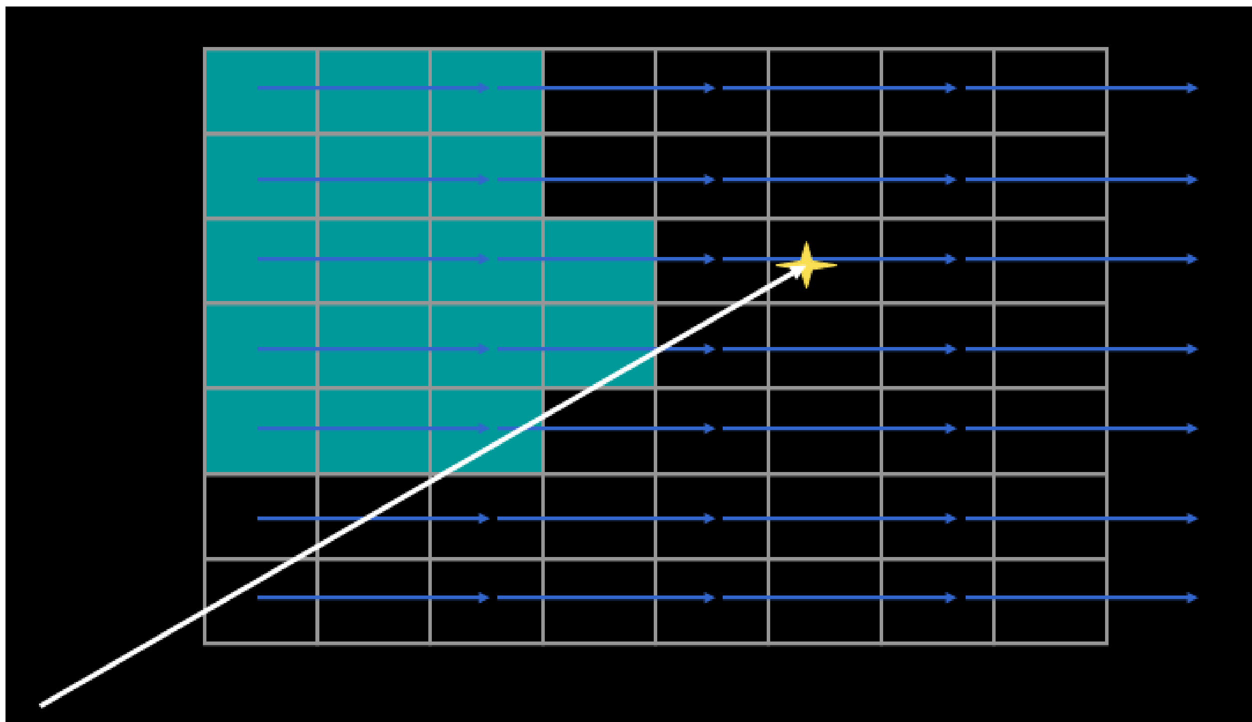


Figure 11. The goal is to estimate the density at the yellow star at time t_r given velocity and density defined at t_f.

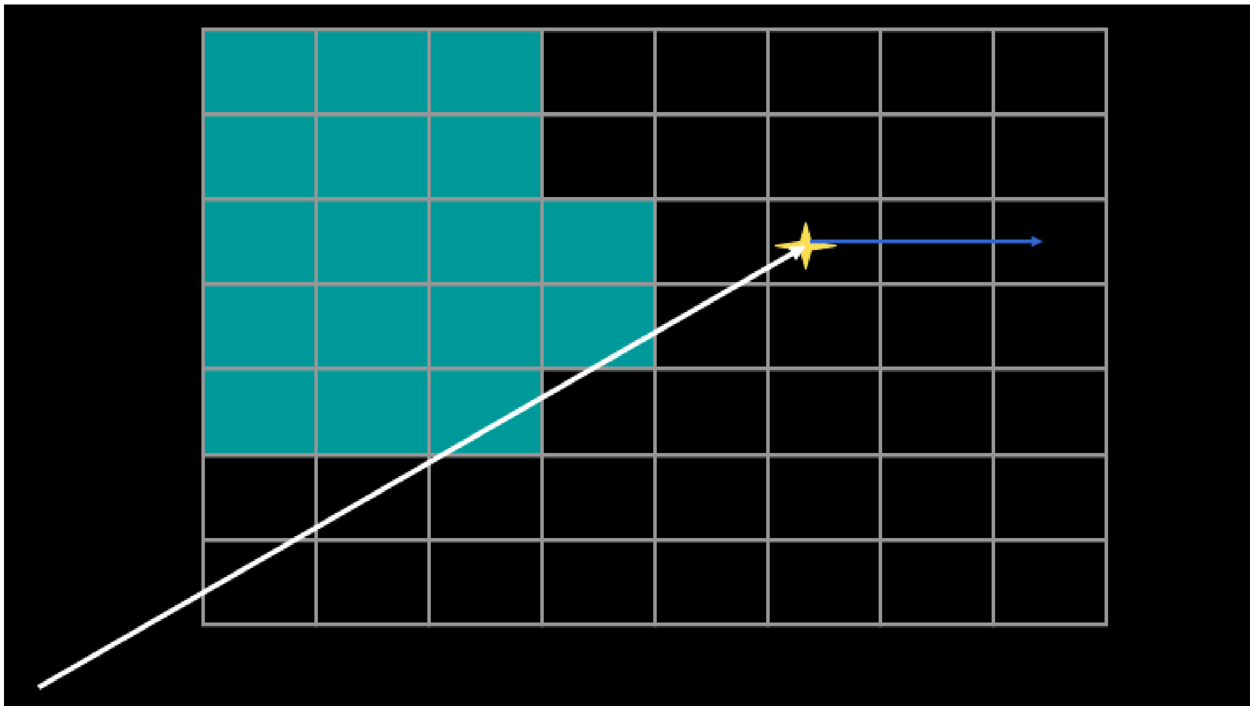


Figure 12. Sample the velocity v_f at the yellow point.

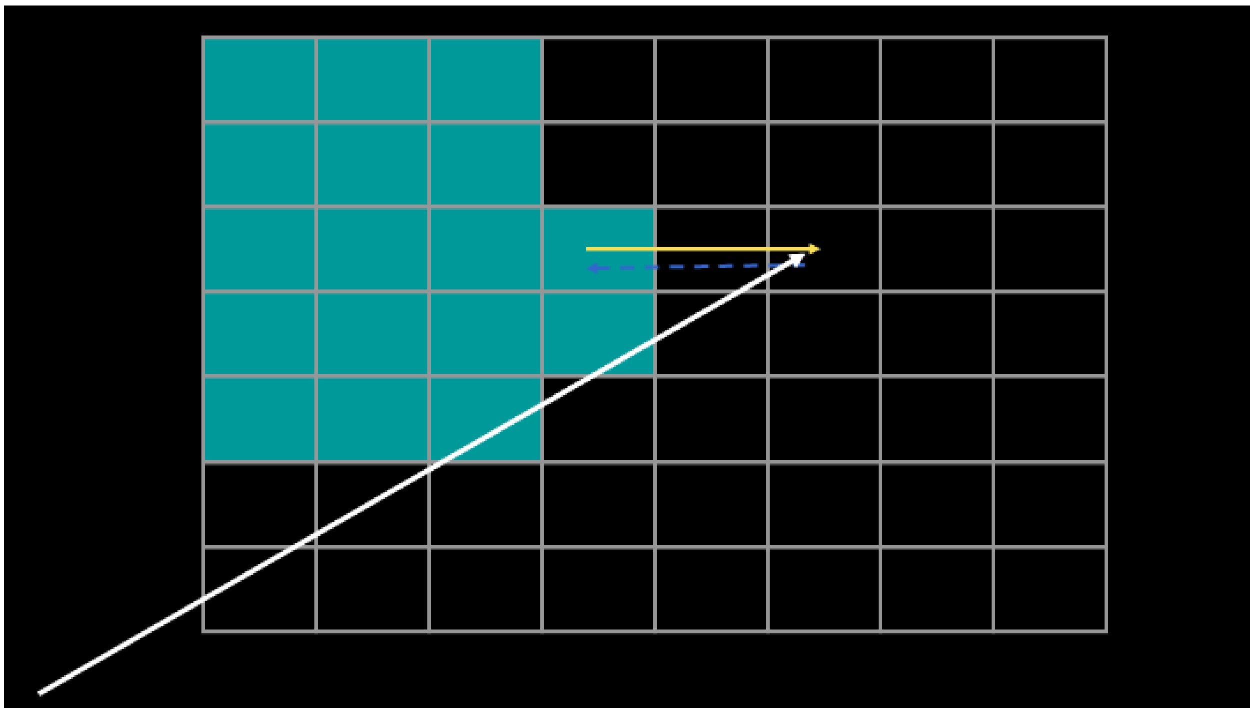


Figure 13. Backtrack v_f to estimate the value for v_r denoted by the the yellow vector.

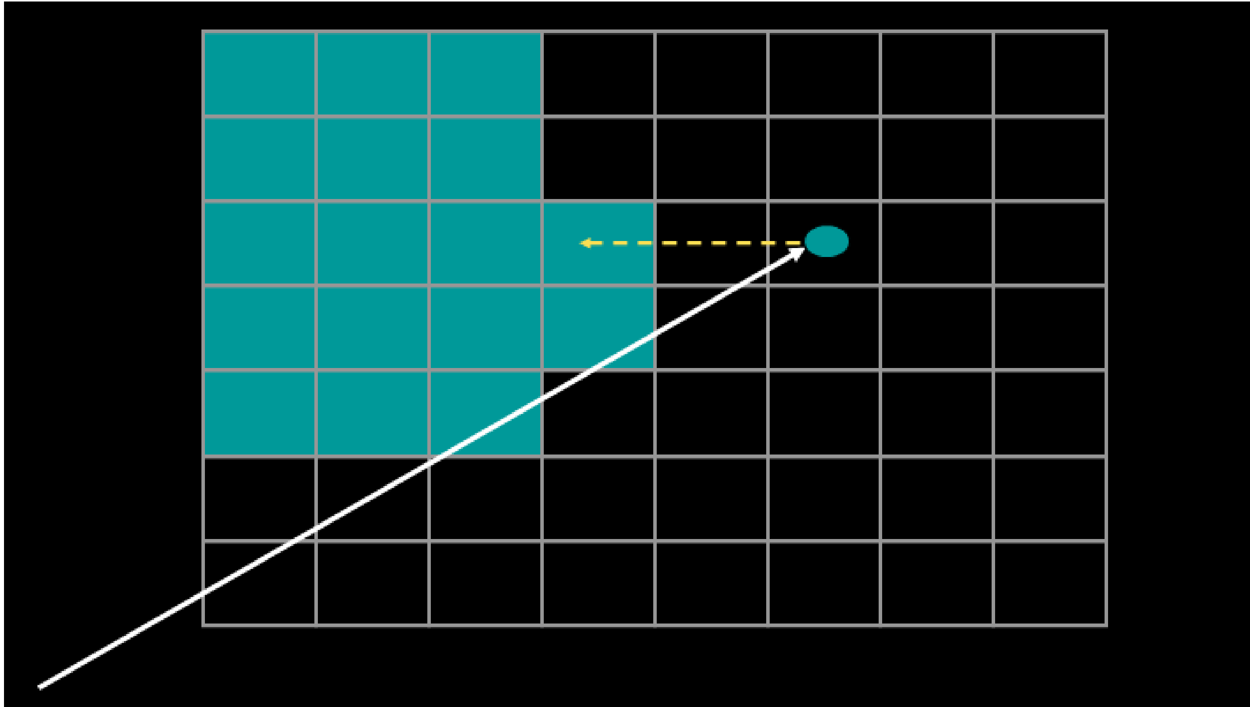


Figure 14. Backtrack v_r to estimate the density at time t_r.

A inherent limitation of this method is that it requires a dense velocity field since it requires sampling it outside of the density region. This is not always the case if you are rendering grids that result from point, curve or surface rasterization. In those cases, the velocity field is only defined where the density is. For cases like this, one must smear the velocity field to encompass the motion of the volume. Alternatively, the motion blur can be embedded directly into the grid by smearing density and color according to velocity.

2D Motion Blur

For large volumes that have a high degree of screen-space motion, 2D motion blur can be a huge time saver since it is significantly faster than the aforementioned algorithm. Unfortunately, 2D motion blur is unrealistic to use when volume layers are being composited with geometry that has complex overlapping motion. In cases where it is imperative that volumes and geometry are composited correctly together, 3d motion blur needs to be combined with either render time compositing of surfaces and volumes or a deep image type of representation [Clinton Elendt 2009].

$$V_s = worldToScreen\left(\frac{\sum_i V_w * \tau_i)}{\sum_i \tau_i}\right)$$

Figure 15. The equation for determining the screen space velocity as described below.

To render with 2D motion, we render a velocity image that encodes screen space velocities of the volume at each pixel. The screen space velocity for a pixel is derived by taking an average of velocities, V_i for each step i along a ray weighted by the transmittance, (T_i). Thus, velocities that are "covered up" farther along a ray contribute less to the screen space velocity. While this doesn't have a physical basis, it offers better results than a mean or max for the 2D velocity.

References

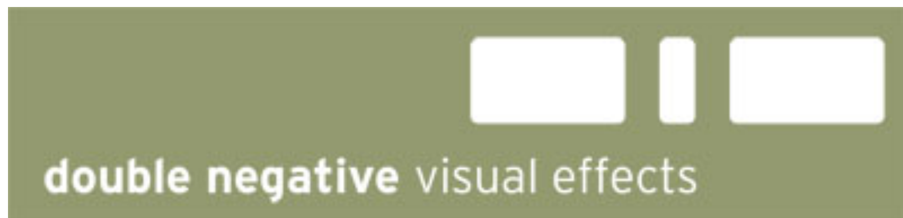
Houston, B., Wiebe, M. & C. Batty. (2004) "RLE Sparse Level Sets." Proceedings of the SIGGRAPH 2004 Conference on Sketches & Applications. ACM Press. [PDF] (Supplemental Materials [PDF])

Kim, D., and Ko, H.-S. 2007. Eulerian motion blur. In Natural Phenomena, Eurographics Association, Prague, Czech Republic, D. Ebert and S. Merillou, Eds., 39–46.

Clinton, A., and Elendt, M. 2009. Rendering volumes with microvoxels. In *SIGGRAPH Talks*, ACM.

DNB

Jeff Clifford & Gavin Graham



History of DNB 4

Initial Aims 4

Later Aims & Features 4

Design of DNB 5

The DNB Voxel Grid 6

Fluid Data Formats 6

Generation of Data 7

Clipping *7*

Fluid Shading *8*

Filling of Voxels (The Preprocessor) 8

Back Interpolation *8*

Hermite-Spline Interpolation *9*

Overlapping Data *10*

Holdouts *10*

Camera Motion Blur *10*

Fluid Motion Blur *10*

The Renderer 10

Voxel Shading *10*

Voxel Attributes *11*

Light-Loop *11*

Self-Shadowing *11*

Ray March *11*

Opacity Normalization *12*

The Rasterizer 12

Memory & Threading 13

Slices *13*

Multi-Threading 13

Other Features 15

Secondary Outputs 15

DNB Holdouts 15

Limitations 15

DNB Shaders 16

Comparison of Shaders 16

Shaders Parameters and UI 17

Fluid Shaders 18

Particle Shaders 20

Voxel Shaders 21

Light Shaders 25

Shaders in Production 27

Inkheart Levelset Shader 27

Hellboy 2 / Harry Potter / Sorcerer's Apprentice Fire Shaders 28

2012 Smoke Shaders 29

2012 Particle Shaders 31

Standard Particle Shading Practice in DNB 31

Instancing Shader Initial Approach 31

Instancing Shader Optimised Approach 32

Bibliography 34

History of DNB

DNB (**D**ouble **N**egative Renderer **B**) was developed at the end of 2004 for use in production on Batman Begins. At the time it was felt that commercial renderers did not provide a fast enough solution for true 3D volumetric rendering and that an in-house solution would benefit Double Negative.

Version 1 of DNB proved successful in production and as a result it was decided to keep developing DNB as part of Double Negative's FX pipeline for all future shows. To date DNB has been used on over 20 feature films including Stardust, 2012, Prince of Persia and The Sorcerer's Apprentice. Over this time there have been 4 major versions with Version 5 now just starting to be used on Harry Potter and The Deathly Hallows.

DNB was designed to fit into the 3D pipeline very much like RenderMan. The renderer itself is a standalone application and the interface to using it and its shaders is provided via a Maya plugin and UI.

The following sections aim to explain the reasoning behind DNB's design and some of the decisions chosen.

Initial Aims

At first it was important for DNB to have the following:

• Low memory footprint
• Simple design
• Fast rendering
• Controllable via shaders
• Self-shadowing

Users wanted to be able to render Maya fluids quickly and easily and at the same time, with limited memory on the renderfarm, it was important to design DNB to be smart with what was available.

Later Aims & Features

After each project on which DNB has been used, the TDs have provided plenty of feedback and as a result more features have been added. In this way DNB is very much a production evolved renderer, with more features always being added but the basic simple architecture staying the same.

The idea behind DNB is that it is very much a useable renderer, with its working process easy enough to explain to all TDs, and accessible enough to allow them to write their own shaders for it.

By the time of DNB v4 the following features were available:

• Camera motion blur
• Fluid motion blur
• Isosurface normals

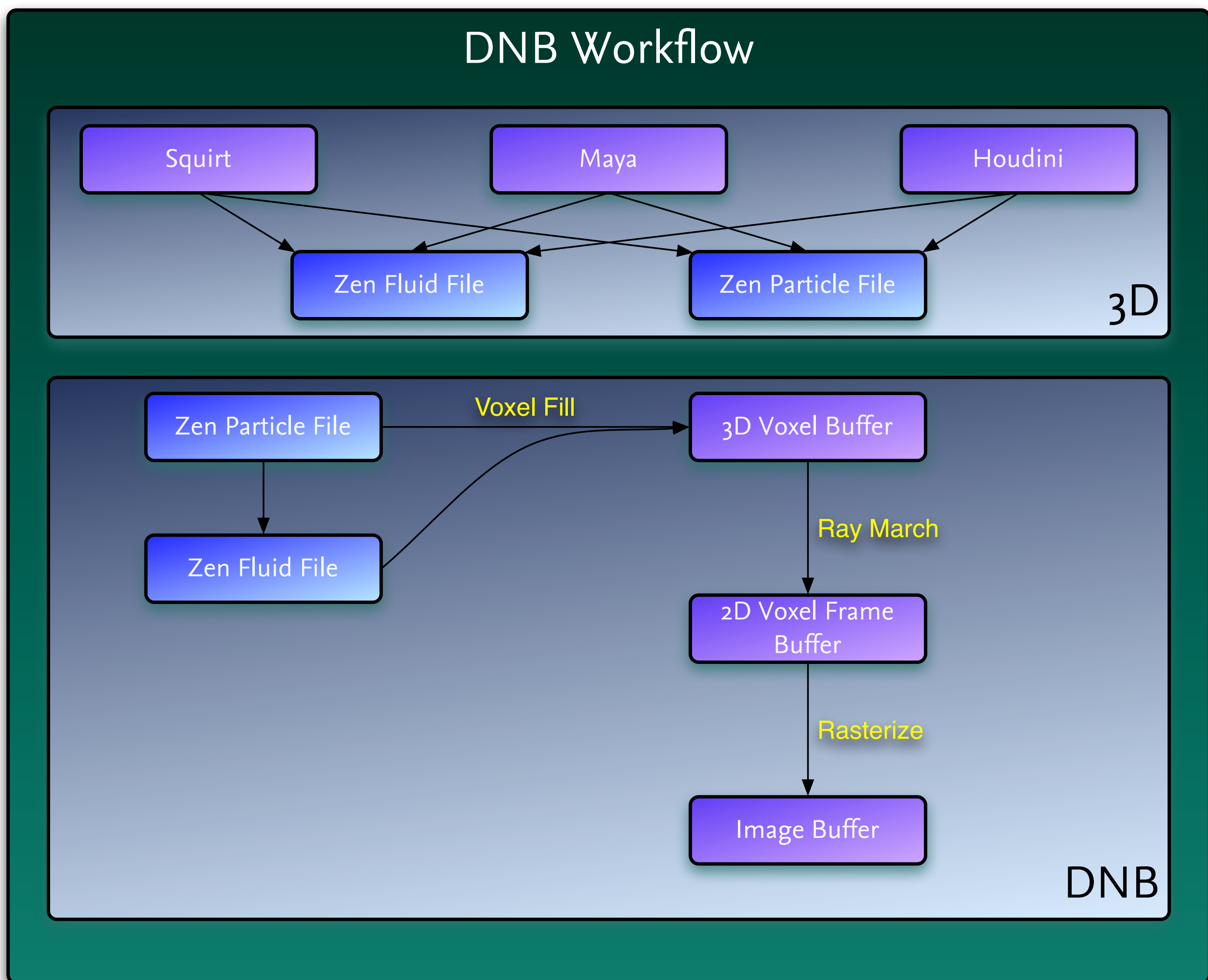
• User attributes
• Memory management
• Multiple threading
• Single scattering
• Multiple scattering

The following notes discuss v4 of the renderer.

Design of DNB

The design of DNB has essentially stayed the same over the various versions with more features and optimizations added on the way.

The following flowchart shows the workflow of DNB:

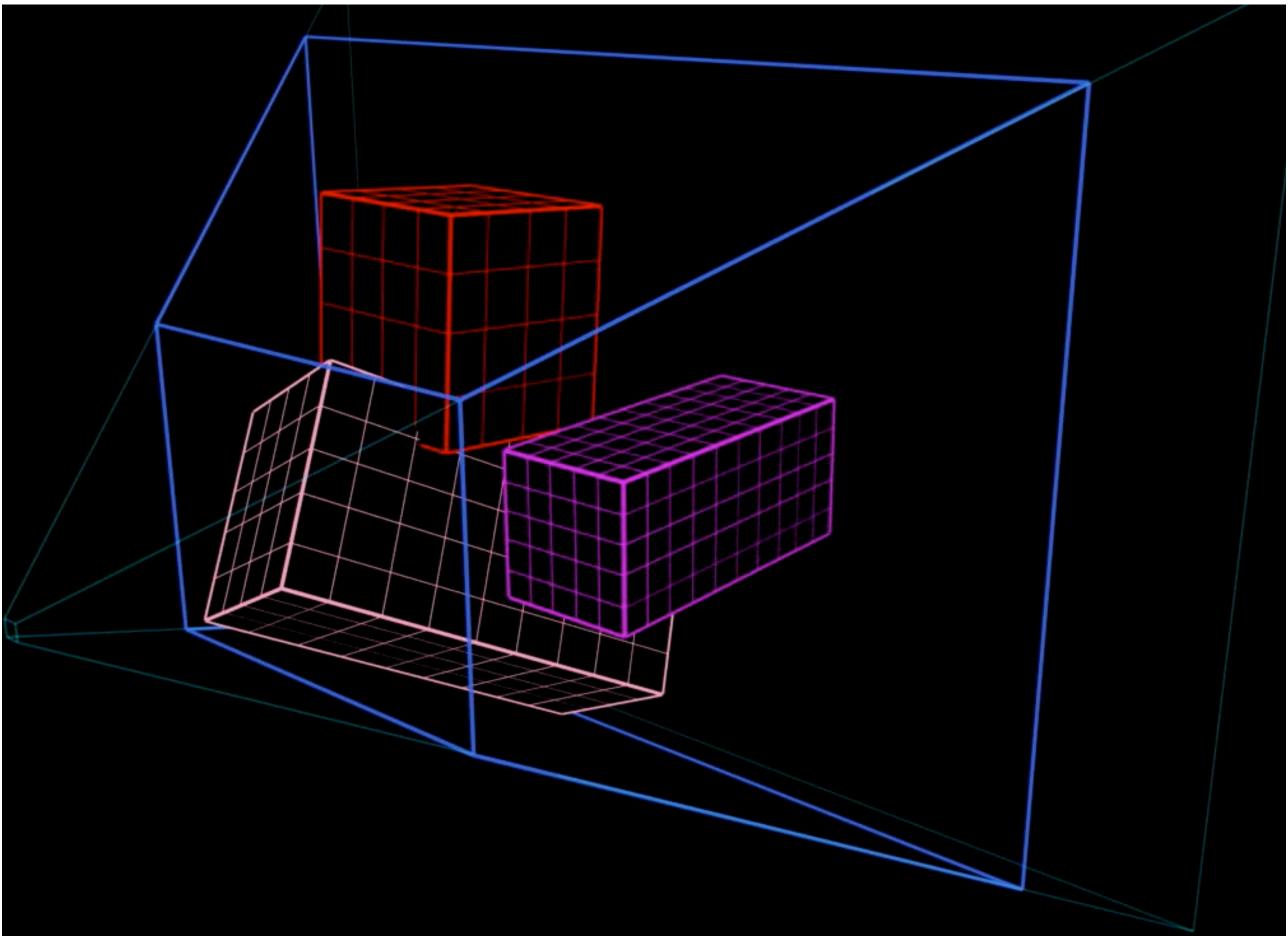


Basically, users generate data using the 3D application of their choice. DNB then takes this data at rendertime and fills its voxel structure with it. From that point DNB renders the image along its voxel z rays before rasterizing an image. The following sections explain the process in more detail.

We stick to discussing using fluid sims here although DNB can also fill from particle data sets. This is because splatting of particles into voxels is covered in the other course notes. It is also because DNB's most common data source is fluids. However, in the shader section loading additional particle data at voxel shading time is discussed.

The DNB Voxel Grid

DNB essentially treats the Fluid sim data as its "primitive". At render time DNB calculates the bounding boxes of all the sims it will render and then creates a frustum aligned voxel grid so that all sims are just enclosed. DNB's voxel grid will only extend as far as the camera frustum. DNB's grid therefore varies on a frame by frame basis. This means that as fluids move to or away from the camera DNB adjusts its grid so that the user specified detail is maintained in the render.



Three fluids in a camera aligned frustum - the thick blue lines indicate the tightly bound volume that DNB will subdivide according to the required voxel resolution

The z spacing of DNB's voxel grid is designed to be regular intervals in camera space so that the sampling rate doesn't vary.

Fluid Data Formats

Initially when DNB rendered Maya fluid sims the data was exported in Maya's PDB data format. This was fine for a while but had some disadvantages - the files were very large and we were restricted by how the data had to be stored. We then moved to Tweak Film's

Open Source GTO format. This allowed for greater flexibility in the types of data we could store - e.g. some data could be stored as *half* rather than *float*. This in turn resulted in smaller files and quicker reading of data. Finally, Double Negative moved to using its own data format ZEN (written by Jon Stroud in 2005 - he claims he discovered it in the arctic!) This allowed us to develop our own Fluid Data Protocol, the result of which meant we could access just parts of data in a file, automatically use LODs and compress the data using a discrete wavelet transform for greater file compression.

Generation of Data

Data passed to DNB now comes from a range of sources. These sources will generate two types of ZEN data - either zenVoxel data (fluids) or zenParticle data (particle sets). Initially just Maya Fluid sims were rendered but these days the following sources all create data that DNB is capable of rendering:

- Maya fluids
- Maya particle sets
- Squirt fluids
- Squirt particle sets
- Squirt levelsets
- Houdini particle sets
- Houdini volumes
- Procedural (rendertime) data

Squirt (initially written by Marcus Nordenstam and now developed by Ian Masters and Dan Bailey) is Double Negative's in-house Fluid Dynamics Simulator. It is by far the most common source of data for DNB due to its capabilities and generation of either fluid or particle data (or both simultaneously) by its solvers.

To aid in the viewing of such data, a set of in-house GPU viewer tools is used to view ZEN files in apps like Maya or via command line. There are also tools for splatting particle ZEN files into fluid ZEN files if desired. Further command line apps allow merging of fluids, extracting statistics about attribute values, retiming etc. These additional tools give the TDs extra flexibility in how they generate their data and quickly reaching the kind of look they are after.

Clipping

It is worth mentioning a few optimizations DNB can do to reduce the amount of clipping checks that need to be done. Clipping and how to do it properly in a renderer is extensively covered in much literature. We'd recommend Jim Blinn's books on the subject (see Bibliography).

As DNB renders data that comes in 3D grids it is worth calculating for each data set whether any of the 8 corner points clip the camera frustum. This allows us on a per frame basis to quickly reject data sets that are completely outside the camera frustum. It also means we can see if the entire data set is contained within the frustum and shrink DNB's voxel grid to just contain it providing for a higher quality render.

Fluid Shading

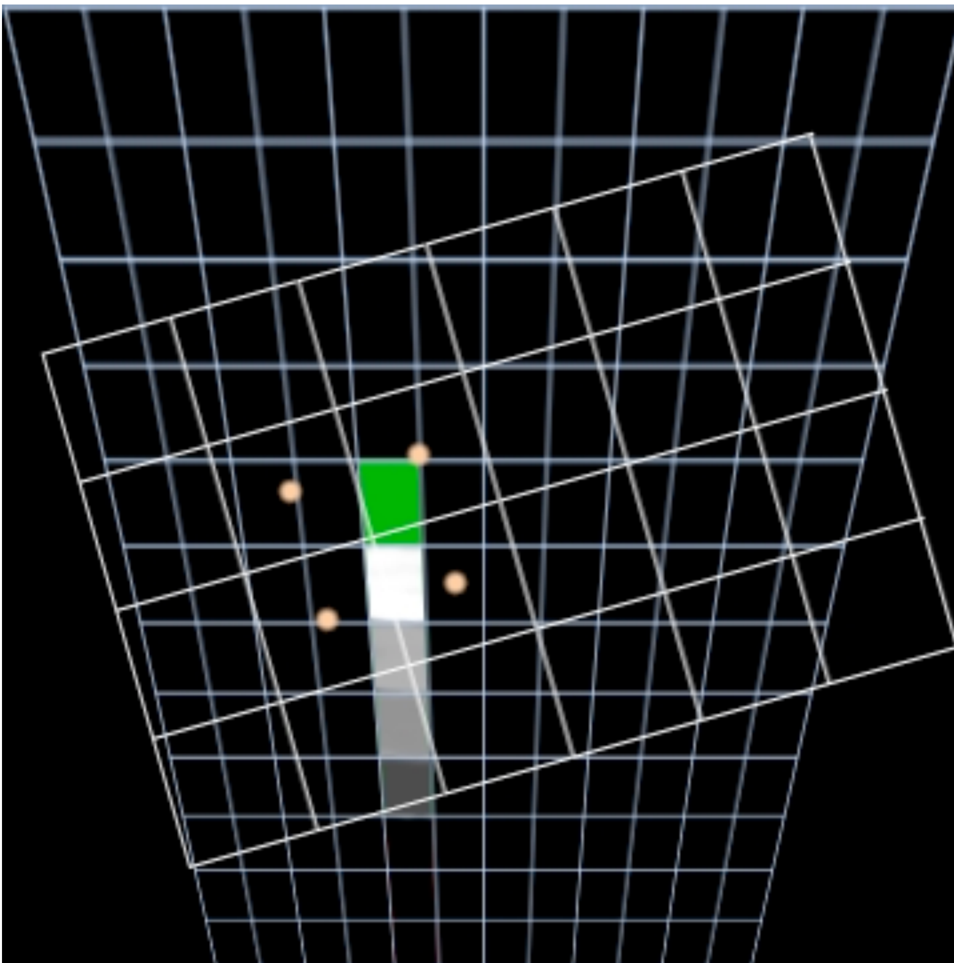
Once data is loading into DNB's memory from a ZEN fluid file it is possible to use a fluid shader on that data. This allows TDs more control over the output look of their render by giving them access to change their data at this point of the rendering stage. Please see the Fluid Shader section for more info.

Filling of Voxels (The Preprocessor)

In DNB a lot of work is done at the same time as filling its voxel structure so that the rendering phase requires less computation. The filling of the voxel structure is a main task though and this section explains how it is done in DNB.

Back Interpolation

DNB takes a slightly different approach to how it fills its voxels to that of other renderers. Instead of splatting the data into DNB's frustum-based voxel grid, DNB iterates over the centre points of each of its voxels and then does a backwards looking interpolation back into the fluid data sets. By looking up the exact value for the centre point of each voxel we avoid aliasing and we then have the data nicely aligned along DNB's voxel's z-rays for ray marching later.



Raymarching along a slice, the current (green) voxel looks up the neighbouring fluid grid cells and interpolates

There are two choices to the interpolation type used to fill DNB's voxels, each with their own advantages and disadvantages:

Tri-Linear	Hermite-Spline
Uses less memory	More memory required for storing gradients
Rougher looking results	Smoother looking results
No gradient data generated	Gradient data generated which is used by renderer later

Although the Hermite-Spline interpolation is also slightly slower than the Tri-Linear interpolation it tends to be used by TDs all the time now due to the additional benefit of using the gradient data to generate an isosurface normal for each voxel. This can then be used for various lighting effects at shading time.

Hermite-Spline Interpolation

Before any interpolation can be performed DNB needs to work out if a voxel is within the 3D space of each fluid sim. Because the sims are simply 3D orthogonal grids, by doing a quick dot product with each of the fluid data's orthogonal vectors it is possible to check not only if the voxel lies within the bounding box of the sim but what the local cell is. The local cell indices are what is needed for the interpolation lookup.

Before solving the interpolation DNB requires the gradients of the attribute it is solving for. Therefore, DNB first of all differentiates each fluid cell's attributes in the local x, y & z directions to use in the Hermite-Spline calculation.

To calculate the Hermite-Spline Interpolation in the 1D case there are four terms rather than two in the Tri-linear case. If DNB is calculating the value v at point s and the attribute values are known at v_1 and v_2 and their gradients t_1 and t_2 known, then the four Hermite basis functions can be calculated and added together to find the interpolated value.

$$h_1(s) = 2s^3 - 3s^2 + 1$$
$$h_2(s) = -2s^3 + 3s^2$$
$$h_3(s) = s^3 - 2s^2 + s$$
$$h_4(s) = s^3 - s^2$$

$$v(s) = v_1 h_1 + v_2 h_2 + t_1 h_3 + t_2 h_4$$

Once the attribute has been solved for the exact centre position of the DNB voxel, if the attribute happens to be opacity then the opacity gradient is also calculated and used to form DNB's isosurface normal for that DNB voxel.

Overlapping Data

An advantage of how DNB fills its voxels is in the way it deals with overlapping data. When DNB fills a particular voxel it loops over all data sets and fills that voxel with data from all of them. If more than one set of data fills a voxel then for each attribute the data needs to combine in the desired way. For opacity this is a simple summation and colour combines according to opacity ratio. Other attributes, like age and phi, can be controlled via the voxel shader to determine how best to combine the data for what the user wants to achieve in the render.

Holdouts

In DNB, holdouts are dealt with at the filling stage. Holdouts are supplied to DNB as Deep Shadowmaps (either from another DNB render or Renderman render). Such a file format means that voxels can be partially heldout if required. It also allows for optimizations to the render time for when a voxel is completely held out, as this means DNB doesn't need to progress further along the z-ray it is currently on. This saves on needless interpolation and filling. A further optimization comes from recording the totally heldout voxel so that at rendering time an early exit along that particular ray is also possible.

Camera Motion Blur

The blurring of data due to the relative translation and rotation of camera and data objects whilst the shutter is open is dealt with in the Preprocessor in DNB rather than at the rendering stage. A simple multi-sample technique is used where for each motion sample another fill call is made with the lookup being in a different location. An average of the number of samples is then taken resulting in voxels that have been blur-filled. Whilst this multi-sampling is time-consuming at this stage, it has the benefit of meaning that still only one voxel shade call needs to be done per voxel gaining us time at the later stage which is often more computationally intensive.

Fluid Motion Blur

There is another type of blurring required when a fast moving fluid requires blurring internally. To do this the fluid data provides velocity data for each voxel. DNB then works out how the fluid density changes for each motion sample based on these vectors.

The Renderer

The renderer part of DNB deals with the shading of voxels after they have been filled by the preprocessor. It is also responsible for the raymarch along DNB's voxel structure z rays.

Voxel Shading

For each voxel DNB calls the voxel shader. The voxel shader then calculates the output colour and opacity for that voxel. The voxel shader usually does a light-loop to shade the voxel. Examples of shaders can be found in the next section. The voxel shader will have access to a variety of base class calls so that performing tasks like self-shadowing and doing diffuse and specular lighting using the iso-surface normal are simple calls.

Voxel Attributes

In DNB each voxel has a range of attributes that can be set and then used in the voxel shade call. The following struct shows what is available to the user:

```
struct voxel_struct
{
	bool totally_heldout;
	float fuel;
	float temperature;
	float user_attr_float_1;
	float user_attr_float_2;
	vector3f velocity;
	vector3f texture;
	vector3f user_attr_vector_1;
	vector3f user_attr_vector_2;
	rgbf colour;
	rgbf opacity;
	rgbf holdout_transparency;
	rgbf mean_scatter_angle;
	rgbf scatter;
	vector3f isosurface_normal;
	vector3f centre_pt;
}
```

As well as common attributes like colour, opacity, velocity etc there are some user slots that store other attributes (e.g. age) or can be used for calculation purposes.

Light-Loop

Within a voxel shade call usually a `light_primary()` call is made which triggers all lights that are lighting the voxel for the primary image output to calculate their contribution to the voxel being shaded. This loop effectively calls the light shader for each of those lights which allows for lighting calculations to be done and other tasks like deep shadowmap lookups to occur.

Self-Shadowing

In DNB self-shadowing from directional and spot lights is achieved by the use of deep shadowmaps using Pixar's .dshd type files[1].

In the light-loop the amount of light reaching a voxel from each light is calculated by looking up the deep shadowmap for that light. This call is provided automatically via the lightShader base class. TDs can override the call to vary the lookup with multipliers or ramps if they so wish.

Ray March

As all voxels are shaded along a z-ray DNB performs an "Under" ray march to calculate the overall value to assign to the voxel frame buffer. The x & y resolution of the voxel frame buffer is the same as DNB's voxel grid but it has no z depth.
The ray march continues until the furthest away z voxel is reached, or if the opacity threshold is reached. An opacity threshold of 0.99 is typically used as an optimization to prevent the renderer needlessly carrying on with the ray march geometrical progression when the value is tending towards 1.0.

This Under is calculated as follows:

$$O_{ray_n} = O_{ray_{n-1}} + Oi_n * ((1.0 - O_{ho}) - O_{ray_{n-1}})$$

For the nth voxel along the ray. A partial holdout is taken care of with O_{ho}. O_{in} is the output shaded opacity.

The equation is the same as the above for the colour calculation but with the opacity multiplied against it as part of the raymarch.

Opacity Normalization

One aspect of DNB that has been very important from its inception is the ability for TDs to perform look development using a low resolution voxel grid and then switch to using a high resolution grid at beauty render time, but still be assured that the overall opacity of the render will be maintained. Essentially TDs want to be able to change the number of z voxels used and still achieve the same look.

In DNB this is achieved via opacity normalization. Essentially we are sampling the same data using fewer or more z samples than before. Therefore, DNB needs to adjust the per voxel opacity to compensate correctly so that after raymarching the same total opacity is achieved. The answer to this problem is actually to solve the geometric progression of the raymarch and not to simply scale the opacity of each voxel according to the change in the number of voxels (i.e. raymarching through 2 voxels of 0.5 opacity is not the same as ray marching through 4 voxels of 0.25 voxels each - even if some textbooks do suggest this!).

The Rasterizer

DNB’s rasterization engine simply converts the DNB voxel frame buffers to image buffers of the desired resolution via linear interpolation. Typically this results in an RGBA image for the primary render and further RGBA images for any secondary outputs.

Additionally, TDs can opt to have Z data rendered out. They can choose a near, far or depth threshold to be written out for each Z ray and this will also be rasterized into the output image in a separate Z channel.

Memory & Threading

Now that we have explained the various stages in a DNB render, the optimizations to memory and threading that have been implemented can be discussed.

Slices

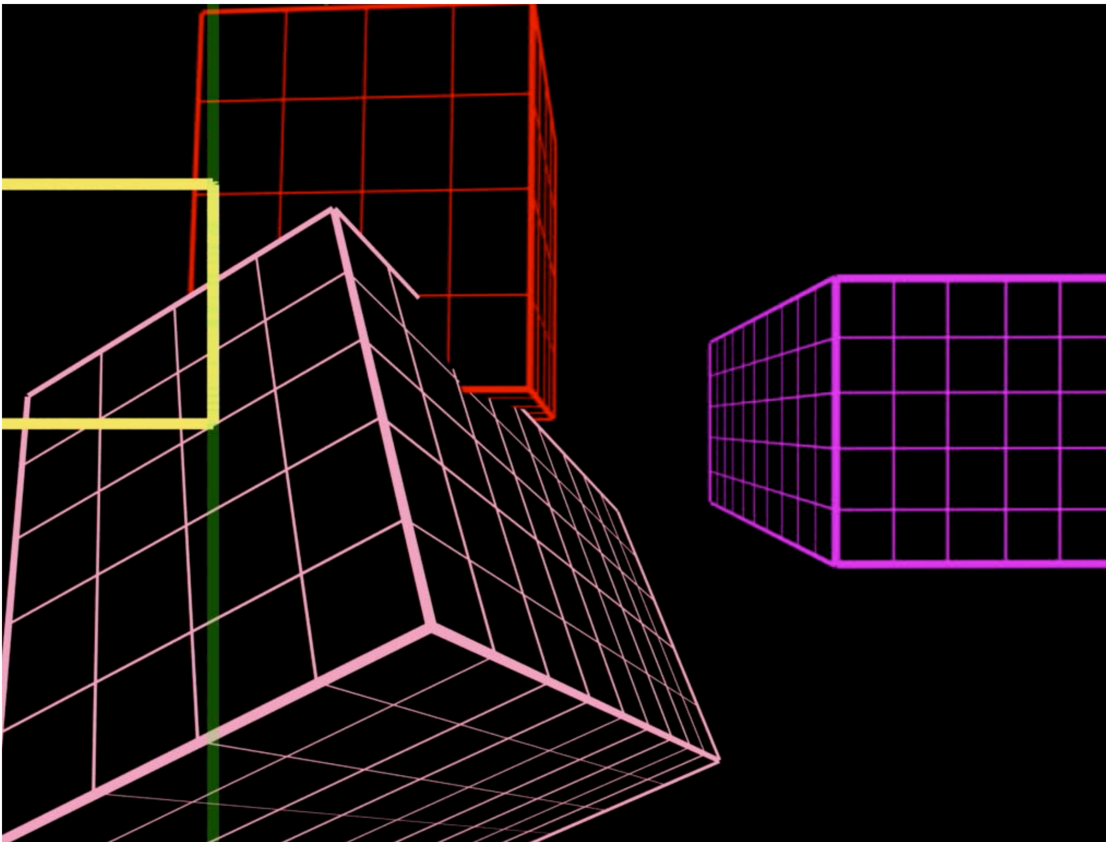
Over time the amount of memory available to users has increased (DNB v1 users typically had 512MB-1GB available, with DNB v4 the amount has increased to at least 16GB) - and as anyone in production knows, TDs will consume this additional amount of memory with ease! At the same time the data sets in terms of numbers to render and individual size have increased. With the film 2012, the fluid sims would be up to 100MVoxels in size and there would be hundreds of sims per frame. Consequently DNB would sometimes need to squeeze over 100GB of data per frame into under 16GB.

To do this DNB was designed to render slices of voxels at a time with the user deciding how much memory each DNB voxel slice should use. A DNB slice is DNB's full voxel grid divided into vertical strips. Each of these strips has a full y & z number of voxels and a reduced number of x voxels.

By using slices DNB can perform further memory reductions by just loading those data sets that contribute to each slice and then unloading them once done as soon as possible. With ZEN files this is taken further by loading only those cells within the data set which contribute to a slice - something that was not possible with PDB or GTO files.

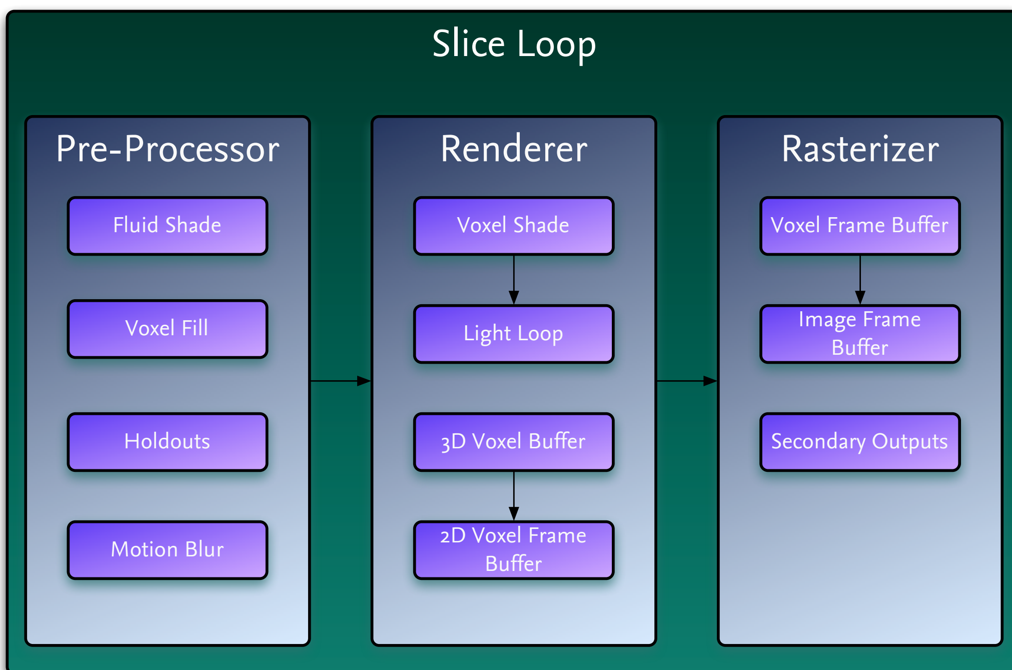
Multi-Threading

Over the years as well as memory quantities increasing, the number of cpu cores has increased. DNB has been optimized to take advantage of this and is coded to do both the preprocess filling and rendering in multiple threads in a simple stable way.



The green line delineates the right hand edge of a single slice for DNB to initially render. The pink and red fluids that overlap this slice will be loaded into memory, purple will not yet be read. Then depending on threading settings, this slice could be subdivided vertically as shown by the Yellow chunk.

Taking advantage of the DNB slice design it is possible to easily divide the vertical slice so that each cpu thread processes a different section of voxels in the slice. As these threads won't write to the same data it means they can both fill and render simultaneously with a linear improvement in rendering speed as a result.



Other Features

Secondary Outputs

In DNB additional voxel frame buffers are available for rendering secondary outputs into. Each additional secondary output is an RGBA buffer. Access to these is provided by the voxel shader where the opacity Oi[n] and colour Ci[n] values set are in a vector array of length n, the number of secondary outputs. This is shown in the shader examples in the next section. DNB's rasterizer then converts the secondaries to images automatically.

DNB Holdouts

As well as rendering images, DNB is able to render deep shadowmaps from the point of view of the camera. Such a render is the same as a beauty render but DNB's voxels are rasterized into a 3D deep shadowmap rather than a 2D image, with just the opacity being set per voxel. A separate `shadow_shade()` call per voxel can be performed at render time in this case to eliminate colour calculations. Such renders can then be used for holdouts in PRMan.

Limitations

We have discussed various features of DNB that it is designed to do well. However, some things are either difficult or very slow to do with DNB and it is worth discussing them and how to avoid such issues.

Multiple Scattering has not been mentioned much in this course. The main reason for this is that such an effect when implemented properly is very expensive computationally. Although DNB has implemented multiple scattering based on generating scatter maps from lights and from the camera, from our experience we found the render time taken to be unsatisfactory. TDs also found that proper multiple scattering reduced their control over where they wanted more or less scatter to occur. As a result "faking" multiple scatter by simply light bleeding the colour in DNB's voxels gave the TDs a much faster and more controllable result.

Whilst DNB has implemented slices to reduce memory consumption this does mean voxel shaders are limited to the extent of lookups in the local neighbourhood in the x direction. To overcome this, users may set a value in voxel shaders to provide a number of x voxels to overlap between slices so that there are always enough voxels available to a particular algorithm. However, in the extreme case for algorithms that needed access to many or all voxels, this overlap could extend to trying to hold the entire voxel structure in memory which would exceed the memory limits DNB currently works under, and negate the advantages of DNB's slice-based approach.

DNB Shaders

As previously mentioned DNB has been designed to be very open and flexible, shader-wise. In this section we'll discuss the different types of DNB shaders and provide examples of their capabilities.

Shaders in DNB are written in C++ and are designed so that they derive from a base class that provides a lot of base class functions and parameters already.

Shaders are then compiled into dynamic libraries that are loaded and executed by DNB during a render. New shaders are distributed and maintained on a per-show basis, but when they reach a certain level of usefulness/global appeal, they become part of the sitewide DNB distribution.

Comparison of Shaders

Over time DNB's shader set has expanded to four different types. This is to allow the TDs as much control as possible over their renders. It also allows them to override the DNB base shaders and create their own specific type shaders - e.g. their own spotlight shader.

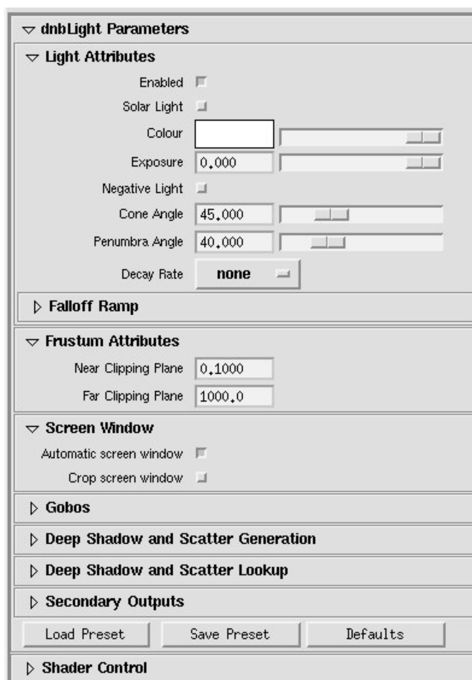
Fluid Shaders	Particle Shaders	Voxel Shaders	Light Shaders
Called once per fluid data set	Called once per particle data set	Called once per voxel	Called in voxel shader's light loop
Act on entire fluid data	Act on entire particle data	Act on current voxel	Act on lights queried during light loop
Have access to all fluid attributes	Have access to all particle data attributes	Have access to neighbouring voxel's data	Have access to all light parameters
Can create extra attributes	Can create extra attributes	Responsible for setting voxel's output colour & opacity	Provide deep shadowmap lookups
Can generate data procedurally	Can instance more particles or fluid data	Call light loop	Provide access to gobos
Used to set opaqueness of fluid	Can cull particles according to LOD	Can calculate secondary outputs	Provide environment map lookups

Shaders Parameters and UI

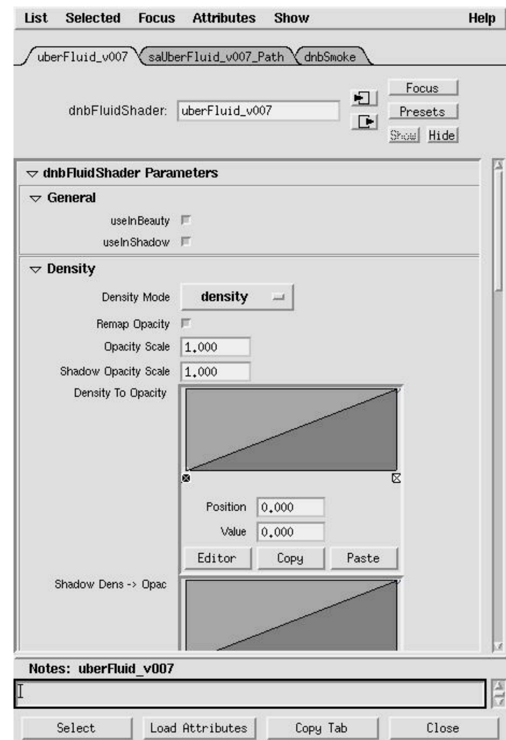
Shader parameters are listed in an associated xml file for each shader written:

```
<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>
       ..
</category>
..
</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.



spotlight shader UI



swiss army knife fluid shader UI (small section)

Fluid Shaders

As previously discussed fluid shaders act directly on the data as provided by the data 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 would cause three different looking fluids to be rendered in the same scene.

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.

The densityFluidShader:

The following code 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 is equivalent to an opacity value of 1.0.

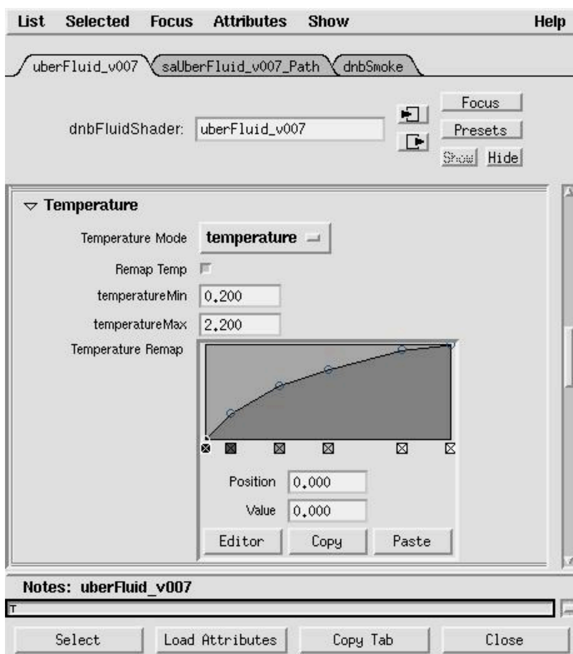
```
// The shade function can be used to do alterations to the
// fluid data (i.e. the Maya/squirt fluid data) before the render occurs.
// It is usually faster to do some operations here rather than in the
// voxel shader since fluid resolution is generally << voxel resolution

bool densityFluidShader::shade(fluid_struct& fluid_structure)
{
   // Lets convert density to opacity according to shader parameter curves
   const float opaque_density = (m_renderInfo->render_pass == "camera") ?
fluid_structure.camera_opaque_density : fluid_structure.shadow_opaque_density;

   vector<float>::iterator density_iter = fluid_structure.hd_attributes.density.begin();
   const vector<float>::iterator density_end_iter =
fluid_structure.hd_attributes.density.end();

   while(density_iter != density_end_iter)
   {
      *density_iter /= opaque_density;

      if(*density_iter > 1.0)
         *density_iter = 1.0;
      if(m_renderInfo->render_pass == "camera")
         *density_iter = DensToOpac.linear_lookup(*density_iter);
      else
         *density_iter = ShadowDensToOpac.linear_lookup(*density_iter);

      ++density_iter;
   }
   return true;
}
```

Here follows some code snippets from a similar fluid shader as it processes a particular (temperature) attribute with more typical fluid shader controls:

```
class uberFluid: public dnbFluidShader {
private:
   int              temperatureMode;
   bool     remapTemperature;
   float    temperatureMin;
   float    temperatureMax;
   ramp     tempRemap;
}
```

```
bool uberFluid::shade(fluid_struct& fluid_structure) {
   vector<float>::iterator temp_iter = fluid_structure.hd_attributes.temperature.begin();
   const vector<float>::iterator temp_end_iter =
fluid_structure.hd_attributes.temperature.end();
   while(temp_iter != temp_end_iter) {
      if (remapTemperature) {
         float tempNorm = linstep( temperatureMin, temperatureMax, *temp_iter) ;
         *temp_iter =  tempRemap.linear_lookup( tempNorm );
      }
      ++voxel_index ;
      ++temp_iter;
   }
}
```

which translates to the Maya UI as:



fluid shader snapshot with temperature remapping

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 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. Also available are four generic slots (two for float attributes and two for vectors) where a user can specify any custom attribute to pass across to the voxel shader, where they could be used to multiply against opacity via ramps or get sent out as secondary outputs.

Particle Shaders

Particle shaders in DNB work in a slightly different fashion to fluid shaders - the renderer lets the user load whatever arbitrary attributes from the particle file the user sees fit, then it is up to the shader writer to populate a struct which contains the standard attributes that the voxel shader expects. Essentially, there are less restrictions put on what will be read in here compared to fluid loading since particles tend to get fairly arbitrary attributes created in different packages, and fluid simulations usually deal with quite specific attributes. But this is of course at the expense of greater shader complexity!
So a typical particle shader (looking at a single attribute) might allow for things like:

```
void simpleParticle::setParticleAttributes()
{
   if (m_age_attribute==0)
      setOptionalAttribute( "age" );
   else setOptionalAttribute( "ageNormalized" );
}

bool simpleParticle::init()
{
   if (m_preprocessAge)    m_renderInfo->preprocess_data.user_attr_float_1 = true;
   return true ;
}
bool simpleParticle::pre_shade( std::vector< dnbParticle >& particles )
{
   if ( !m_hasAgeNormalized ) particles[i].AddAttribute( "ageNormalized", 0.0f );
   if (m_age_attribute==0)
      particle.AttributeOffset( "age", m_ageNormalizedOffset );
   else particle.AttributeOffset( "ageNormalized", m_ageNormalizedOffset );
   return true;
}

// It is safe to assume that this function will only be called within the bounding sphere
of each particle
bool simpleParticle::shade( const dnbParticle& particle, const GML::Point3f&
localPosition, particle_attributes_struct &attributes)
{
   // Sort out Age
   float ageNormalized = 0 ;
   if ( ageScaleEnabled || m_ageOpacityRemapEnabled || age_density_remap_enabled ||
m_preprocessAge) {
      ageNormalized = particle.GetFloatAttribute( m_ageNormalizedOffset ) ;
      ageNormalized = linstep ( age_normaliser.x, age_normaliser.y, ageNormalized ) ;
      ageNormalized = age_norm_remap_advanced.linear_lookup ( ageNormalized ) ;
   }
   if ( preprocess_data.user_attr_float_1 ) {
      attributes.user_attr_float_1 = ageNormalized ;
   }

   return true;
}
```

Basic particle shaders instance either a radial blob of density or a sphere of noise on to each particle as discussed elsewhere in the course with regards to other renderers - a shader that allows for fluid instancing will be discussed here in the production focussed section.

Voxel Shaders

Voxel shaders are at the heart of the DNB rendering process and this is where all the hard work gets done. The dnbVoxelShader base class provides a lot of functionality and a very simple shader can be written in just a few lines:

```
// Note this shader uses the base class' light_primary call
// (which may be overloaded by your own version instead if you like)

// The aim of this call is to set Ci[0] and Oi[0]

bool generalVoxelShader::shade(
	vector<voxel_struct>* voxel_structure,
	int voxel_index,
	const ms& E_ms,
	vector< IMG::RGBf >& Ci,
	vector< IMG::RGBf >& Oi)
{
	const voxel_struct& cur_voxel = (*voxel_structure)[voxel_index];

	const GML::Point4f cam_pt(cur_voxel.centre_pt);

	// Set colour initially to zero in case we don't shade this voxel
	Ci[0] = IMG::RGBf(0.0, 0.0, 0.0);

	// Opacity is unchanged
	Oi[0] = cur_voxel.opacity;

	// Light loop
	if(GML::fgtz(Oi[0].r) || GML::fgtz(Oi[0].g) || GML::fgtz(Oi[0].b))
	{
		if( !light_primary(Ci[0],Oi[0],cur_voxel,cam_pt,E_ms) )
			return false;
		Ci[0] *= UseColourData ? cur_voxel.colour : VoxelColour;
	}

	return true;
}
```

This shader will automatically light the voxel correctly via the base class `light_primary()` call. A param also decides whether to use the colour contained in the voxel data or some user defined (via shader parameters) colour.

The IsosurfaceVoxelShader:

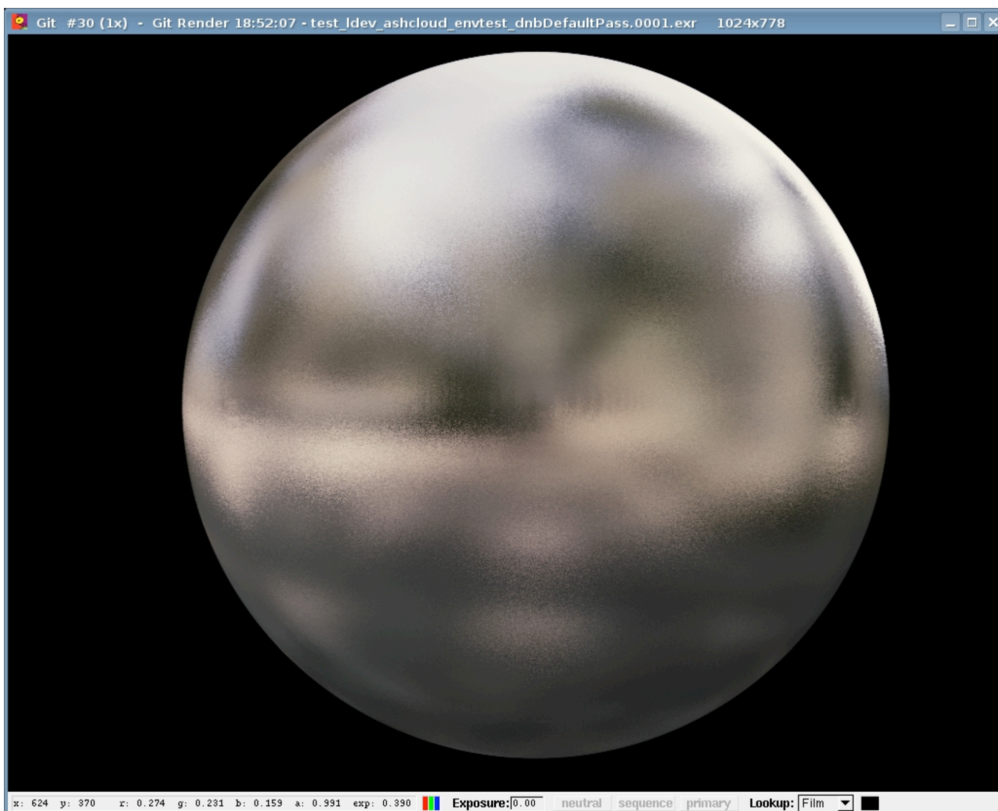
The following example is more complicated and uses DNB's isosurface normal voxel attributes to enable DNB to lookup an environment texture map. The overall effect is to light the fluid with the environment map and where the size of the isosurface normal vectors is greatest this causes a surface to appear as the lookup is strongest - essentially an isosurface occurs.

```
bool isosurfaceVoxelShader::shade(
   vector<voxel_struct>* voxel_structure,
   int voxel_index,
   const ms& E_ms,
   vector< IMG::RGBf >& Ci,
   vector< IMG::RGBf >& Oi)
{
   voxel_struct& cur_voxel = (*voxel_structure)[voxel_index];
```

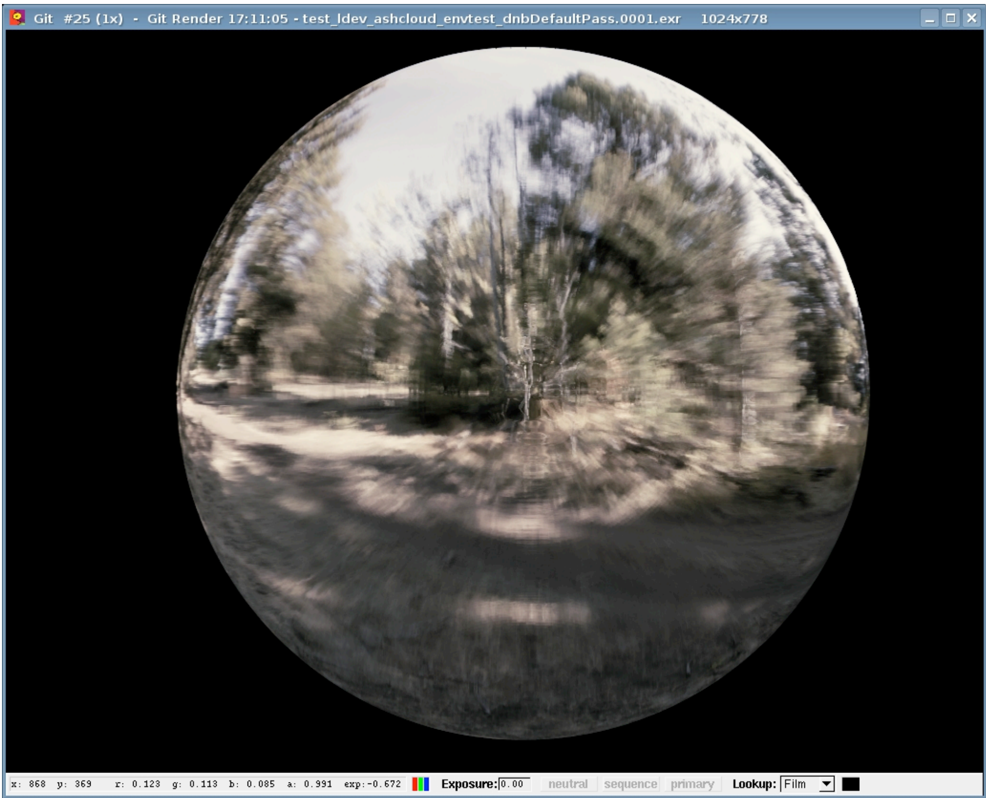
```
   const GML::Point4f cam_pt(cur_voxel.centre_pt);

   // Set colour initially to zero in case we don't shade this voxel
   Ci[0] = IMG::RGBf(0.0, 0.0, 0.0);

   // Opacity is unchanged
   Oi[0] = cur_voxel.opacity;

   // Light loop
   if(GML::fgtz(Oi[0].r) || GML::fgtz(Oi[0].g) || GML::fgtz(Oi[0].b))
   {
      // Lets iterate over the local voxels to average the iso-surface normal

      const GML::Vector3f orig_isosurface_normal = cur_voxel.isosurface_normal;

      if( LookupTechnique != NONE )
      {
         GML::Vector3f meaned_isosurface_normal;  // Initially (0.0, 0.0, 0.0)

         vector< int > neighbours( m_n_total_lookups );

         vector< int >::iterator i_neighbours = neighbours.begin();
         const vector< int >::iterator i_neighbours_end = neighbours.end();

         if( LookupTechnique == RANDOM )
         {
            m_renderInfo->seed_random_generator( static_cast<unsigned int>
( voxel_index ) );
            while( i_neighbours != i_neighbours_end )
            {
               // get random values from this thread's random number generator
               const int x = static_cast< int >( m_renderInfo->random_value() *
m_x_range_plus_one - m_x_half_range_plus_half );
               const int y = static_cast< int >( m_renderInfo->random_value() *
m_y_range_plus_one - m_y_half_range_plus_half );
               const int z = static_cast< int >( m_renderInfo->random_value() *
m_z_range_plus_one - m_z_half_range_plus_half );

               *i_neighbours = voxel_index + x * voxel_x_step() + y * voxel_y_step() + z *
voxel_z_step();

               ++i_neighbours;
            }
         }
         else if( LookupTechnique == REGULAR )
         {
            for(int x = -XLookupRange; x <= XLookupRange; x += XRegularStepping)
            {
               for(int y = -YLookupRange; y <= YLookupRange; y += YRegularStepping)
               {
                  for(int z = -ZLookupRange; z <= ZLookupRange; z += ZRegularStepping)
                  {
                     *i_neighbours = voxel_index + x * voxel_x_step() + y * voxel_y_step()
+ z * voxel_z_step();

                     ++i_neighbours;
                  }
               }
            }
         }

         i_neighbours = neighbours.begin();

         while( i_neighbours != i_neighbours_end )
```

```
        {
          meaned_isosurface_normal += (*voxel_structure)
[ *i_neighbours ].isosurface_normal;

          ++i_neighbours;
        }

        meaned_isosurface_normal /= static_cast<float>( m_n_total_lookups );

        // Modify vector for light shader calls

        cur_voxel.isosurface_normal = meaned_isosurface_normal;
      }

      if( !light_primary(Ci[0],Oi[0],cur_voxel,cam_pt,E_ms) )
        return false;

      // Set vector back so that other voxel lookups are correct

      cur_voxel.isosurface_normal = orig_isosurface_normal;

      Ci[0] *= UseColourData ? cur_voxel.colour : VoxelColour;
    }

    return true;
}
```

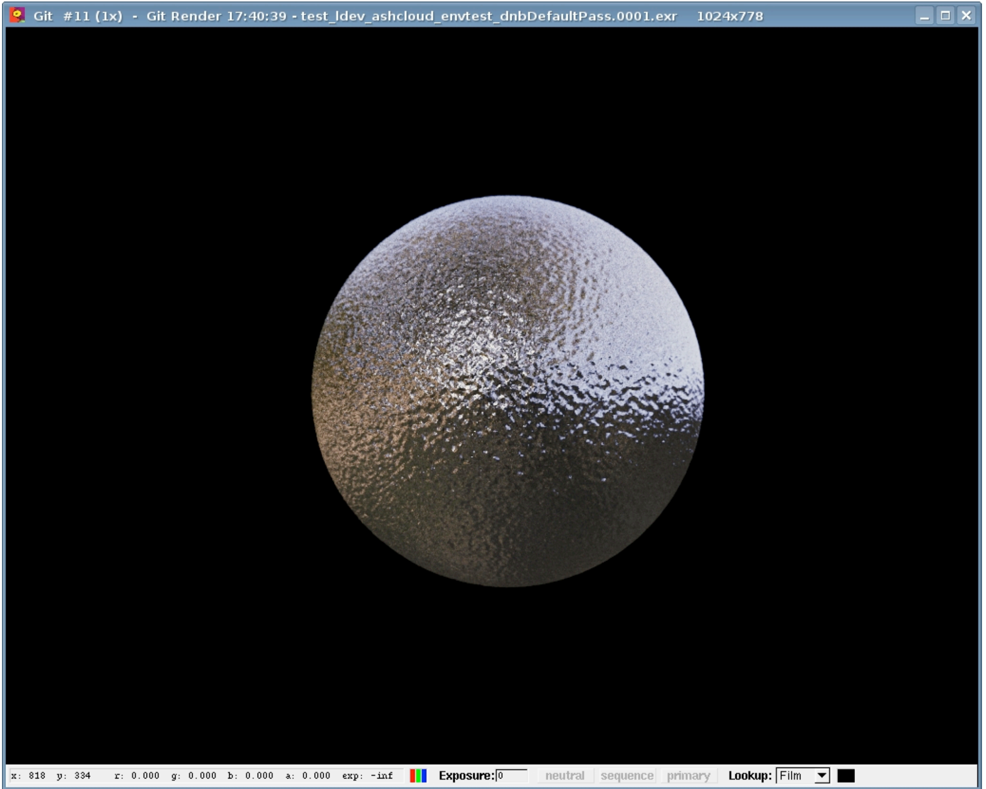
IsosurfaceShader shader results:



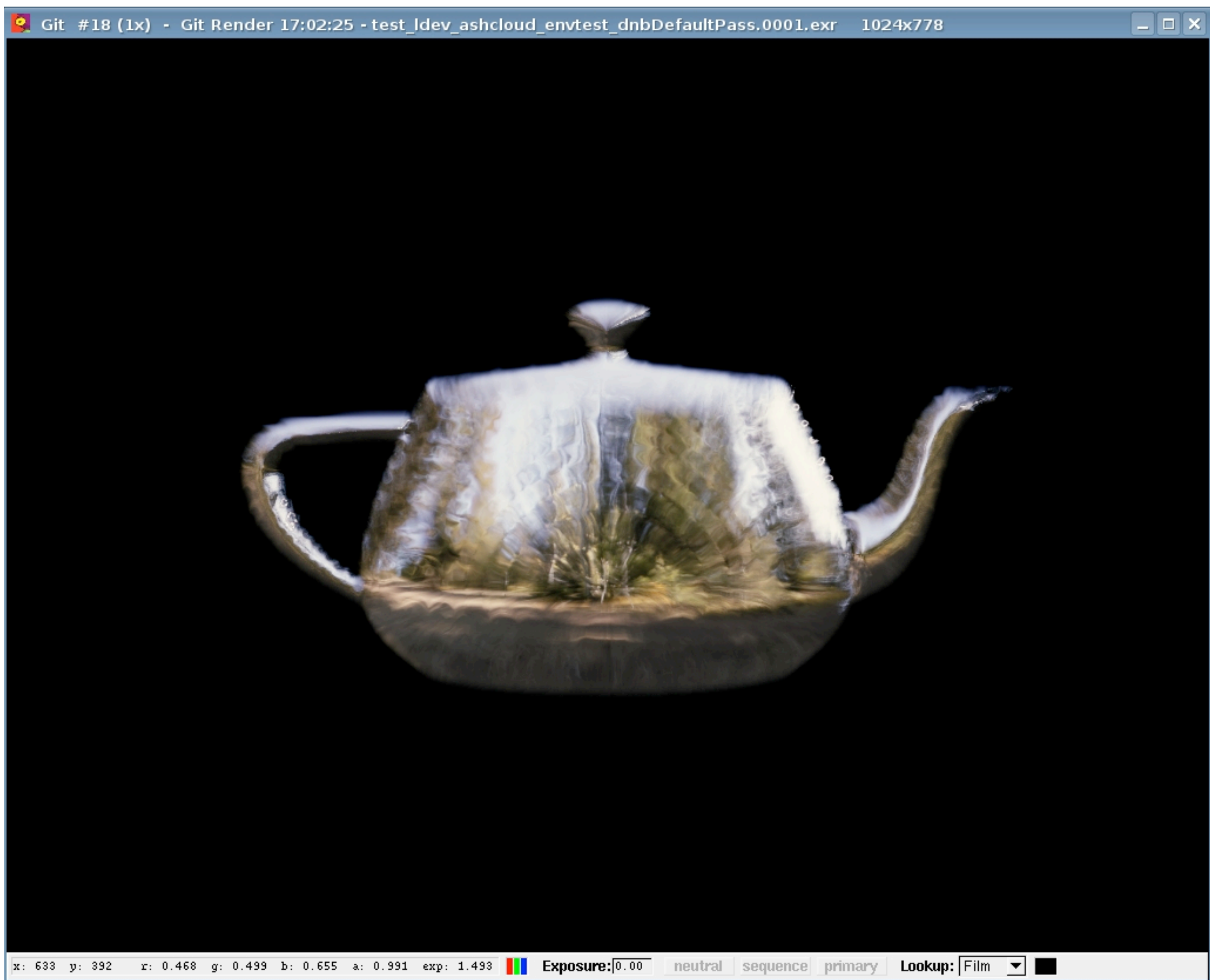
Using a random lookup of 100 nearby voxels produces a blurred look.



In this render a regular grid lookup was used instead with the nearest surrounding voxels used to generate a meaned isosurface normal. As a result the image is less blurred and with a less dithered look.



A more interesting result is shown here due to the voxel isosurface normals being varied so as to effectively bump map the fluid!



Finally we have the compulsory teapot render! No Siggraph Course should be without one!

Light Shaders

Light Shaders not only describe the shading effects for a light but they literally describe the full light. In fact, within Maya, DNB shaders are loaded and then become DNB Light nodes - effectively replacing the use of Maya lights. As a result, at render time DNB has access to all light attributes as well as shader parameters.

The base lighting classes in DNB look after the effect of each light on the voxel shader when it does its `light_primary()` call. But each light could override the functions from the base class if it wanted. DNB has native support for the following types of lights: spot light, point light, ambient light, directional light. In practice, the spot light is used in 95% of renders, and the main thing the light shader is relied on for is to do shadowmap lookups.

In theory, one could modify anything in the voxel structure in the light shader. In practice, it just modifies the colour. An important distinction is that this isn't the shader that gets run when generating the shadowmap - that is run through all the same processes as a beauty render, just from the point of view of the light and with a deep shadowmap rather than image being generated.

Though DNB gives the ability to TDs to modify most aspects of the process as they see fit, we have seen very little need to customize light shaders in production - but custom controls for falloff on a pointlight were added on a recent show to match a setup that was look-developed in compositing.

```
bool specialLight::shade(
   IMG::RGBf& Cl, IMG::RGBf& Os,
   GML::Point3f& Ps, GML::Vector3f& L,
   IMG::RGBf& Cl_shadow, ms& L_ms,
   GML::Vector3f& /*N*/)
{
   Cl = colour_intensity * pFalloff(L);
   if( !gobos( Cl, Os, Ps, L, Cl_shadow, L_ms ) )
      return false;
   return true;
}

const float specialLight::pFalloff(const DNB::Vector3f& L)
{
   float decay = 1.0;
   float fadeout = linstep( m_near_clip, m_far_clip, L.length() ) ;

   if(m_shading_light.decay_rate != none)
   {
      const float length_L_squared = fadeout * fadeout;

      if(m_shading_light.decay_rate == linear)
         decay /= sqrt(length_L_squared);
      else if(m_shading_light.decay_rate == quadratic)
         decay /= length_L_squared;
      else if(m_shading_light.decay_rate == cubic)
         decay /= length_L_squared * sqrt(length_L_squared);
      else
         cerr << "dnbLightShader:  Unknown decay rate" << endl;
   }

   return decay - 1;
}
```

The spotlight `deepshadow()` lookup has also been overridden to do a blurred shadow lookup to get softer edges to shadows, and give the appearance of less blocky shadows when using a low resolution shadowmap source.

Shaders in Production

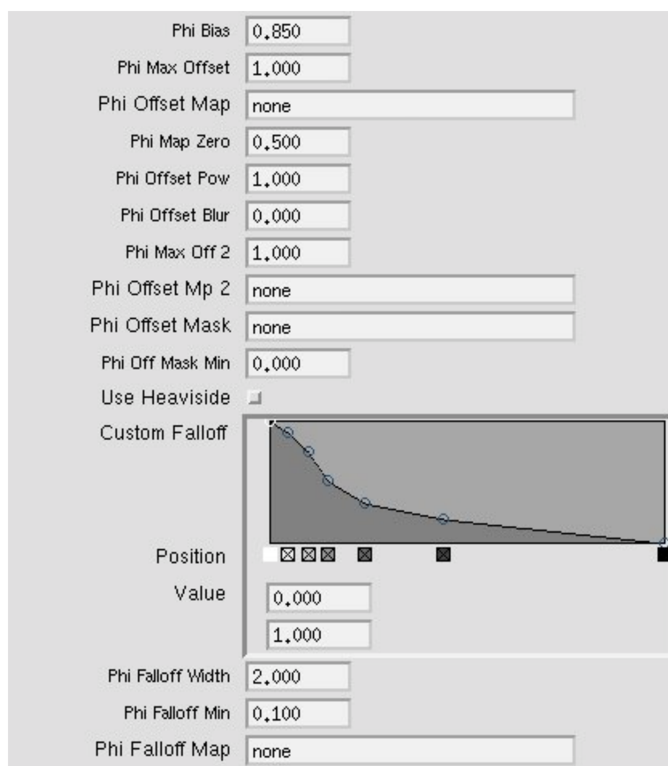
Throughout the last 6 years of DNB's use, it has been found that what most TDs desire is the flexibility to produce either a completely correct beauty "out of the box", which means providing maximum levels of control over any parameters that could improve the look of the final image, or else to be able to send as many useful arbitrary outputs to the compositing stage, so that the right look could be found there without any need for re-rendering. 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 our belief that this aspect of the renderer is just as important as some of the more intricate aspects of the renderer internals. Due to many of these being written in production by TDs who did not always have a strong programming background, much importance was placed on the renderer being open to programmable extensions without significant understanding of the underlying mechanisms being always required.

Inkheart Levelset Shader

On a project featuring a character made entirely out of smoke, it became necessary to swap out a rigid character from inside smoke sims that were supposed to envelope him as when he moved quickly, as limbs could suddenly become apparent 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 (a user attribute, not one supported by DNB as standard) per fluid and convert the levelset information to density at rendertime in the shader.

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



part of the levelset shader parameters UI

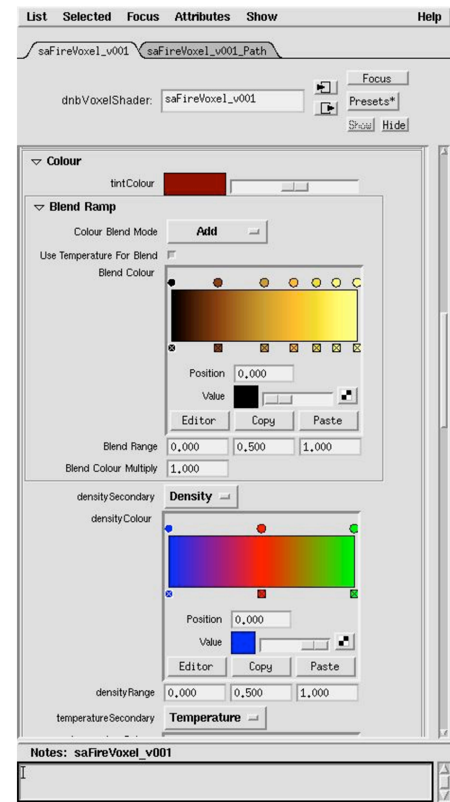
Being a fluid shader made it possible to generate separate levelsets for hands, arms, body, head, and custom texture maps for each. By rigidly constraining the levelsets to the animation rig, resolution could be tailored for each body part based on shot requirements.

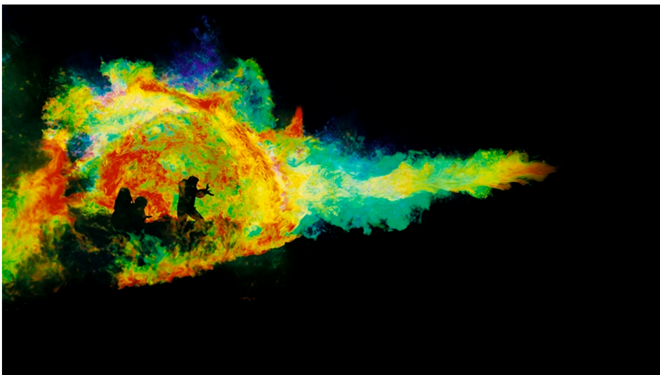
It was found to be a powerful additional tool in DNB's volume rendering arsenal, created by TDs without having to change the renderer's underlying code or architecture.

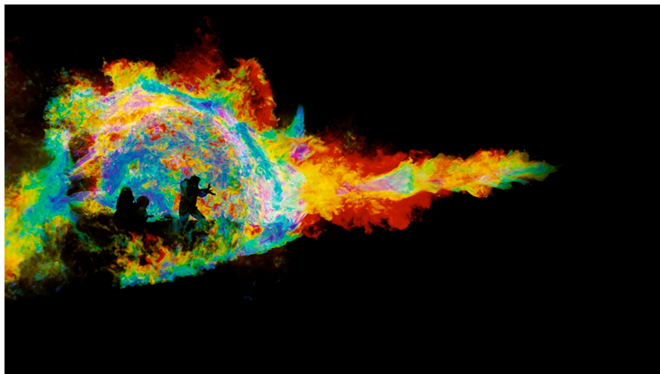
Hellboy 2 / Harry Potter / Sorcerer's Apprentice Fire Shaders

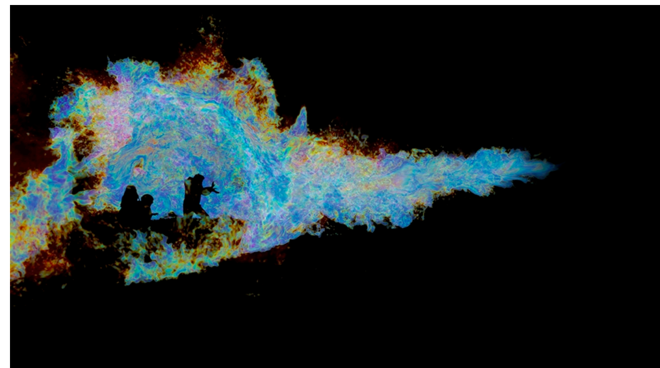
Fire was an effect that first came up for DNB on Hellboy 2, then reappeared again and again on subsequent shows. A particular setup has evolved using our "uber" fluid and voxel shaders that seemed to give compositors flexibility to create various different looks for the fire without needing resims or 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.









Fire passes, clockwise from bottom left: density remap, temperature remap, age remap, typical raw beauty colour output, composite produced using the remapped secondary outputs

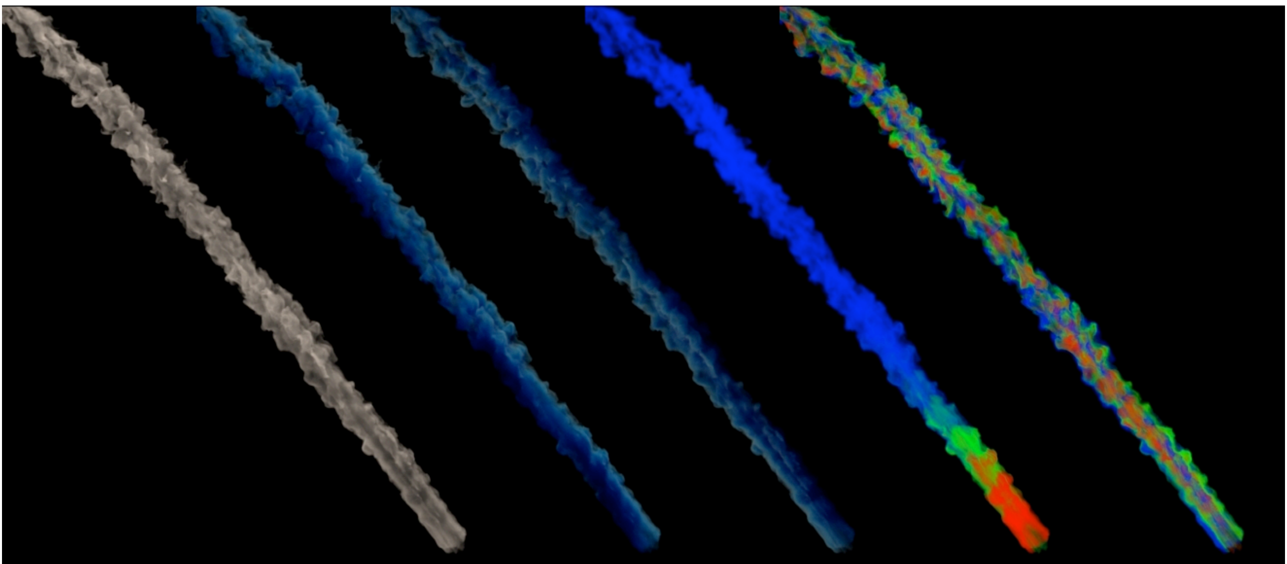
2012 Smoke Shaders

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, a variant of 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 common trick used across these smoke types was to render shadowmaps with a ('orange') multiplier on each colour channel of the opacity - so the shadow opacity multiplier red channel had a density similar to the beauty opacity, the green channel had a lower value (typically 30% the red value) so more light penetrates the medium, and the blue channel had a very small value (perhaps 5% of the red) such that most light penetrated very deeply, or virtually all the way through the medium from the light's point of view.

When these shadowmaps were looked up in the beauty pass, a cyan looking result was achieved, where the red channel could be used for quite a harsh rim light, the green gave the average "typical" light falloff, and the blue channel could be used for filling in detail so no areas had to be completely in shadow from any light.

The beauty output would use 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 so that compositors had the power to make the same lighting decisions without needing a re-render.



Smoke shader output: beauty, key light, fill light, temperature, texture coordinate driven noise.

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, checks must be put in place to ensure particular attributes are needed - the voxel shader passes information back to the fluid shaders via the renderInfo structure, the fluid shaders then check for these signals and activate the attributes accordingly.

So the voxel shader might contain in the init() function

```
if ( ((noiseCoords==TEXTURE) && ((noiseMix>0)||(noiseBlendMode>0))) || (texBlendMode>0))
{
	cerr << "uberVoxel: preprocessing texture" << endl ;
	m_renderInfo->preprocess_data.texture = true;
}
```

which is then picked up on by the fluid shader's pre_shade() call - this is executed after the fluid and voxel inits, but before any shading occurs:

```
doTexture = (m_renderInfo->preprocess_data.texture) || (noiseSpace==2) ||
(colourMode==3) ;
// possible that texture is needed even when voxel shader hasn't asked
// since fluid shader can use it to drive density and colour too
if (doTexture) {
	if (!fluid_structure.contains_texture) {
		cerr << "uberFluid: creating texture" << endl ;
		if(!createFluidAttribute(fluid_structure, "texture"))
				return false;
		}
	else cerr << "uberFluid: texture found - processing." << endl ;
}
```

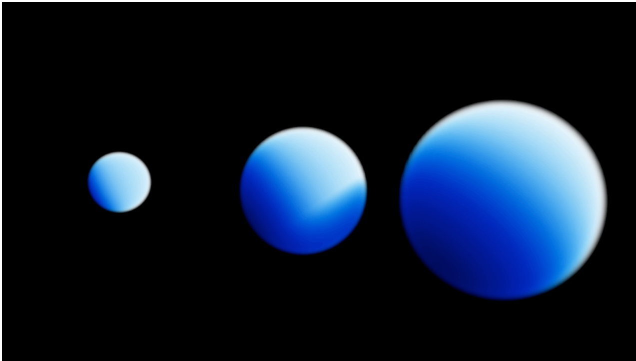
and then we can iterate over the texture attribute and fill it from disk in the fluid shader's shade() as has been shown for other attributes:

```
if (doTexture) texture_iter = fluid_structure.hd_attributes.texture.begin();
//etc
```

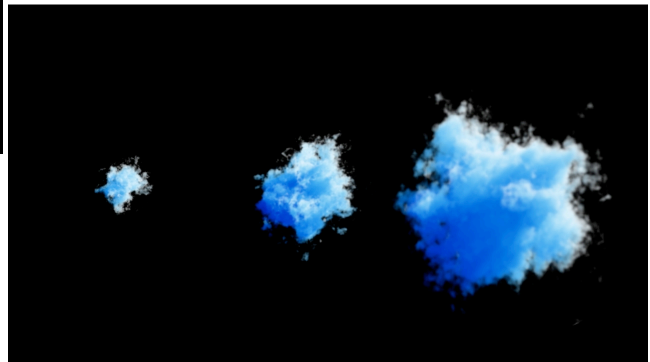
2012 Particle Shaders

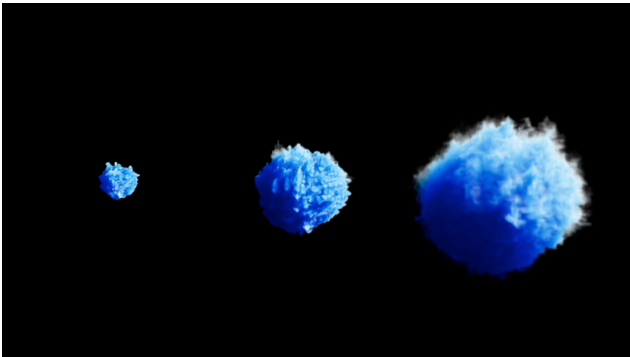
Standard Particle Shading Practice in DNB

As is typical with many other volume renderers, when DNB wants to render particles it gives the user the option of sticking some sort of radially fading soft blob onto each point, or generating noise in the volume around the point.



Particle render types, clockwise from top left: radial density falloff, procedural fluffy noise, 2012 style instanced fluids





On 2012 we found the need to go a lot further than what was provided to the user out of the box, so we did the typical DNB workflow of writing a shader which did roughly what we wanted, and then handed the code 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.

Instancing Shader Initial Approach

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 was written to allocate density buffers for the storage of the fluid data e.g.

```
   size_t          m_res;
   size_t          m_res2d;
   size_t          m_res3d;
   float*          m_noiseBuffers;

init() {
   m_noiseBuffers = new float[ m_fluidBufferCount * m_res3d ];
   for (as many fluid buffers as you want):
      float* densityBuffer = NULL;
      densityBuffer = new float[ density_size ];
      zenVoxelReader.readFullGrid( fAttr, densityBuffer, 2 ) ;
      // and a bit of other code to get the zen file read in
      int index = m_res2d*z + m_res*y + x;
      float densValue = densityBuffer[index];
      densValue = m_fluid_density_ramp.linear_lookup( m_densityMultiplier * densValue ) ;
      m_noiseBuffers[currentBuffer*m_res3d + index] = densValue ;
      delete[] densityBuffer;
}

shade() {
   // nearest_neighbour_lookup, trilinear_interpolation also supported
   size_t localIndex = static_cast<size_t>( noisePos.z + 0.5f )*m_res2d +
         static_cast<size_t>( noisePos.y + 0.5f )*m_res +
         static_cast<size_t>( noisePos.x + 0.5f );
   size_t absIndex = bufferIndex*m_res3d + localIndex;
   density = m_noiseBuffers[ absIndex ];
}
```

but of course with this naive approach we ran into such obvious problems as massive memory consumption when trying to allocate many buffers (one for each frame of the fluid we wanted to instance, and we were trying to instance many fluids). But as proof of concept went, this was a great way for us to work out all the mechanisms involved in the setup, how to setup our particles, and so on.

Instancing Shader Optimised Approach

At this stage we could either live with our limitations (on most shows this would have been fine, but not on the epic to beat all epics that was 2012) or look to somebody who knew what they were doing - enter R&D, and statically allocated density buffers which could be shared across particle systems, and deallocated on demand, to optimise memory use.

```
struct InstanceInfo {
   float* data;
   int instanceCount;
   InstanceInfo() :
      data(NULL),
      instanceCount(0)     {     };
   ~InstanceInfo() {delete[] data;     };
};
class Particle : public dnbParticleShader {
private:
   std::vector< InstanceInfo* > m_fluidInstances;
   static std::map< std::string, InstanceInfo* > s_instancedFluids;
   bool initFluidInstance( const std::string& filename, int bufferId );
      // what used to happen in-line in init()
   bool populateBuffer( const std::string& filename, float* instanceData );
   std::vector<std::string> m_instancedFilenames;
}
```

```
//Each instance of a particle shader creates its own structures..
initFluidInstance( const std::string& filename, int bufferId ) {
   InstanceInfo*& instance = s_instancedFluids[ filename ];
   if ( !instance )
      instance = new InstanceInfo;
   m_fluidInstances[bufferId] = instance;
   m_instancedFilenames[bufferId] = filename;
   if ( instance->data ) {
   // if the data is already loaded, increment the instance count and return
      ++instance->instanceCount;
      return true;
   }
   instance->data = new float[ m_res3d ];
   ++instance->instanceCount;
   populateBuffer( filename, instance->data );
}

populateBuffer( const std::string& file, float* instanceBuffer ) {
   zenVoxelReader.readFullGrid( fAttr, densityBuffer, 2 ) ;
   // and a bit of other code to get the zen file read in
   for( int x=0; x < xVoxelRes; ++x ) {
   for( int y=0; y < yVoxelRes; ++y ) {
   for( int z=0; z < zVoxelRes; ++z ) {
      int index = xVoxelRes*yVoxelRes*z + xVoxelRes*y + x;
      float densValue = densityBuffer[index];
      densValue = m_fluid_density_ramp.linear_lookup( m_densityMultiplier * densValue ) ;
      instanceBuffer[index] = densValue;
      // So we're assigning to an instance->data buffer
      // which is static in memory and shared across shader instances
   } } }
}
// and it had a fluid destructor too!
// this lets us clean up any buffers that we're done with e.g. if different particle
shaders/fluid instances are being used across the width of the frame
```

Once these corrections to the code were made, the TDs were able to get on with cranking up the instance numbers and generating their pretty pictures.

Example shot from 2012: ashcloud was too epic to construct from fluids through standard methods, so particles were distributed across a sculpted ashcloud surface, and pyroclastic sims instanced into that. Lavabomb trails also used procedural noise blobs to avoid the need for fluid simulation

Bibliography

[1] Tom Lokovic, Eric Veach. Pixar Animation Studios. Deep Shadow Maps (Siggraph 2000)

(Jim Blinn's Corner Series)
A Trip Down the Graphics Pipeline (1996)
Dirty Pixels (1998)
Notation, Notation, Notation (2003)

Volume Rendering at Sony Pictures Imageworks

SIGGRAPH 2010 Course Notes: Volumetric Methods in Visual Effects

Magnus Wrenninge[1]

[1] magnus.wrenninge@gmail.com

1. Table of contents

History and overview 3

- Pipeline overview 4
- The volumetrics toolkit 5

Volume modeling pipeline 12

- Generators 15
- Filters 16
- Procedural volumes 16

Volume rendering 18

- Svea's rendering pipeline 18
- Shade trees 20
- Adaptive raymarching 22
- Empty space optimization 26

Production examples 30

- Hancock – Tornadoes 30
- Cloudy With a Chance of Meatballs – Stylized clouds 32
- Alice In Wonderland – Absolem, the smoking caterpillar 34
- Alice In Wonderland – The Cheshire Cat 36
- Alice In Wonderland – A mad tea party 38

2. History and overview

The volume rendering pipeline at Imageworks has been developed over the last 5 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 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.

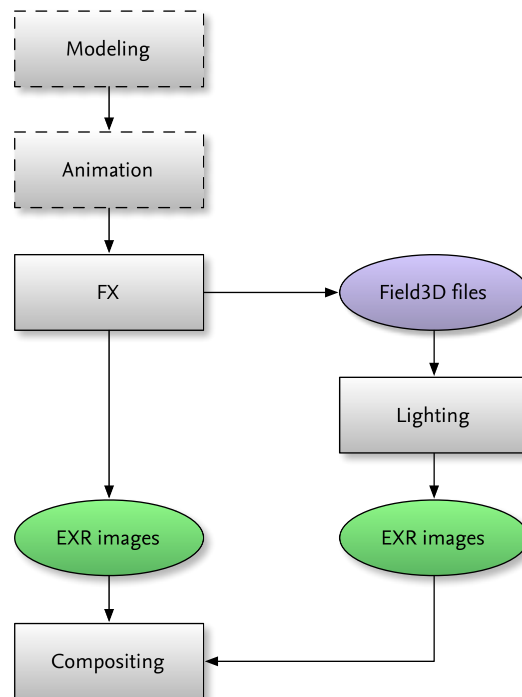
2008–2009 : FIELD3D
: The Field3D library was developed in an effort to unify the volumetric-related tools. Svea was updated to use the Field3D data structures as its native format for voxel buffers. The FieldTools were re-written (version 2) to support Field3D data structures natively in the Houdini 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 fully 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.

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,* and *Alice In Wonderland.*

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.

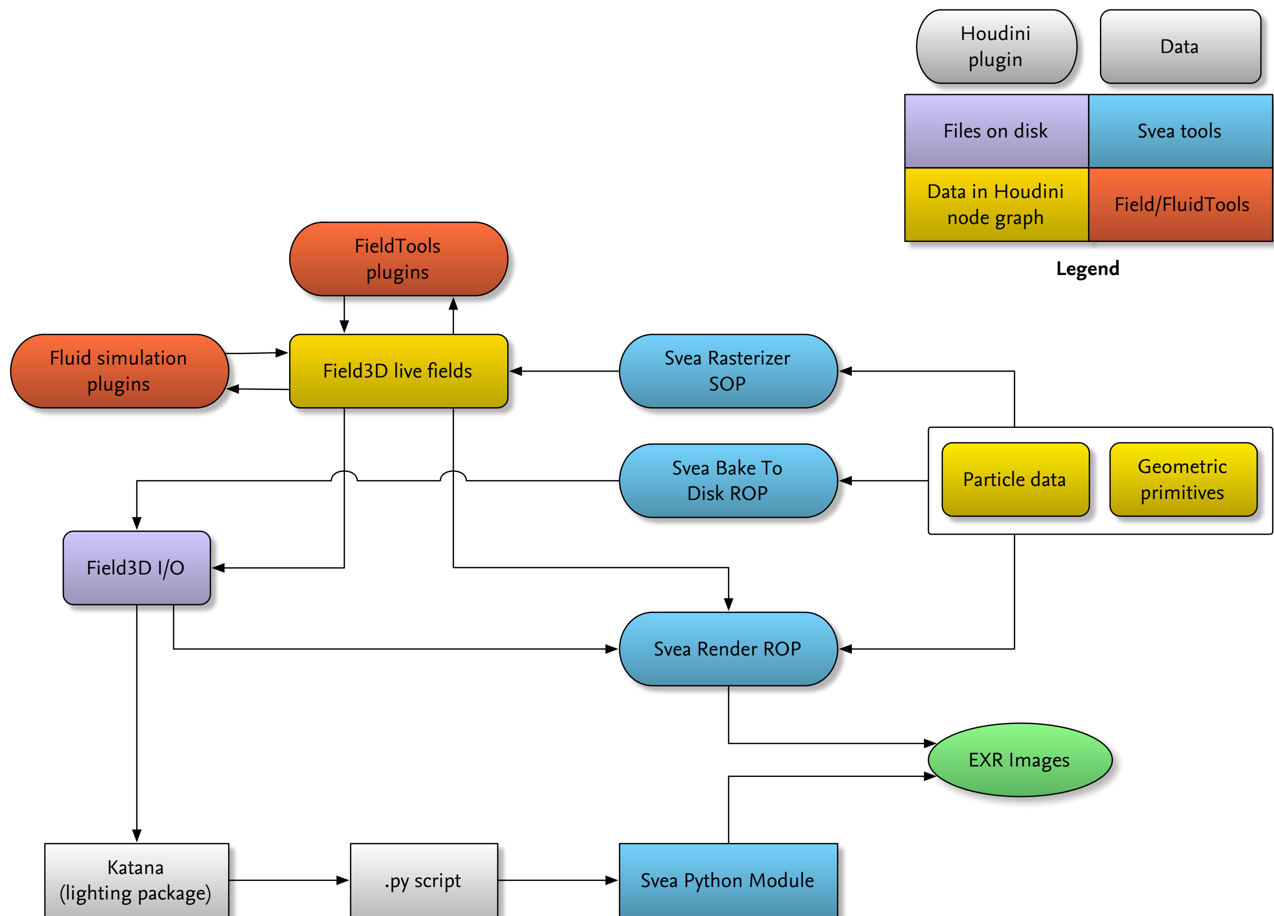
2.2.2. Svea

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

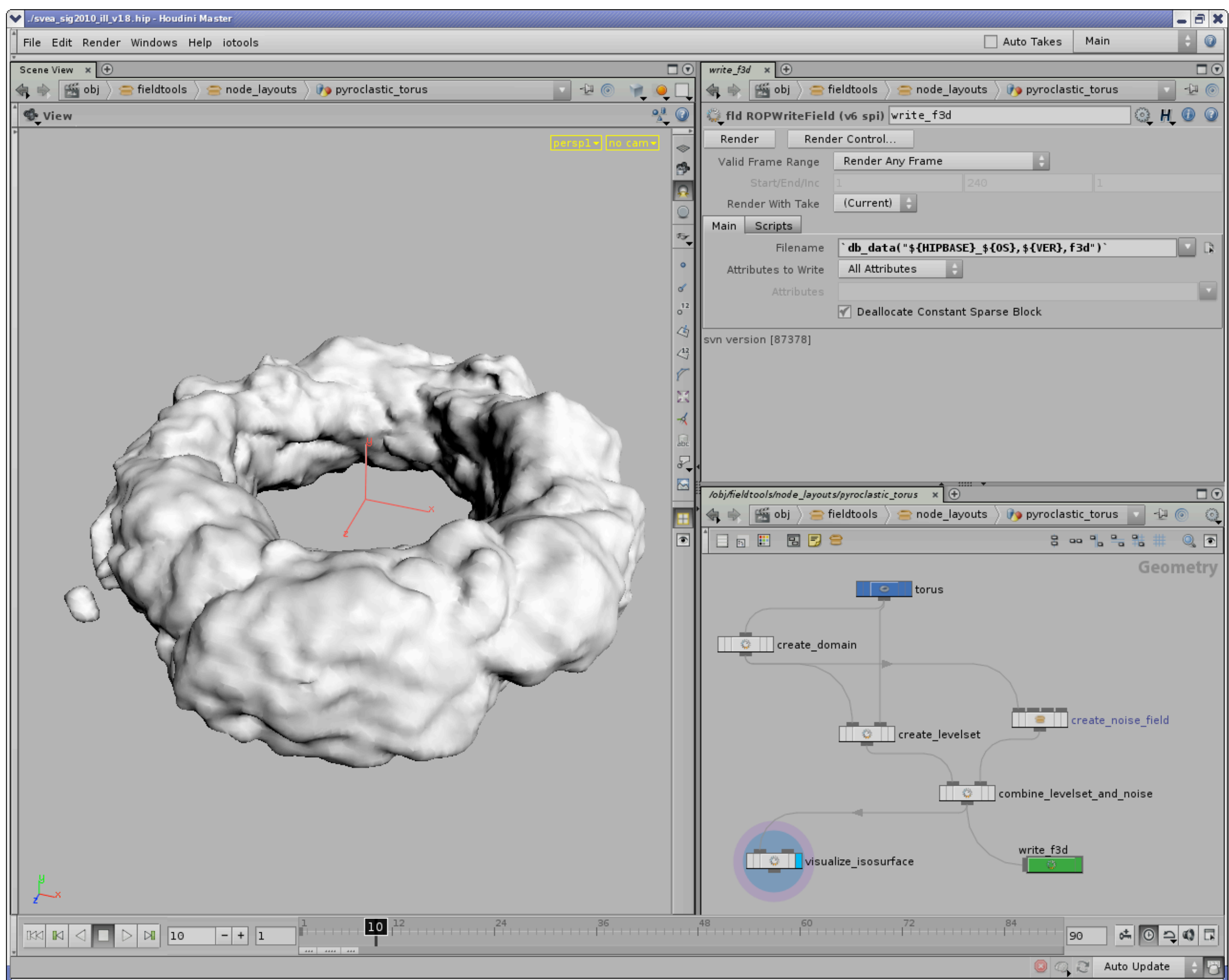
rasterized volumes can either be raymarched directly (in the Svea Render ROP), be output back into the Houdini node graph (the Svea Rasterizer SOP) or written to disk (the Svea Bake To Disk ROP).

- 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

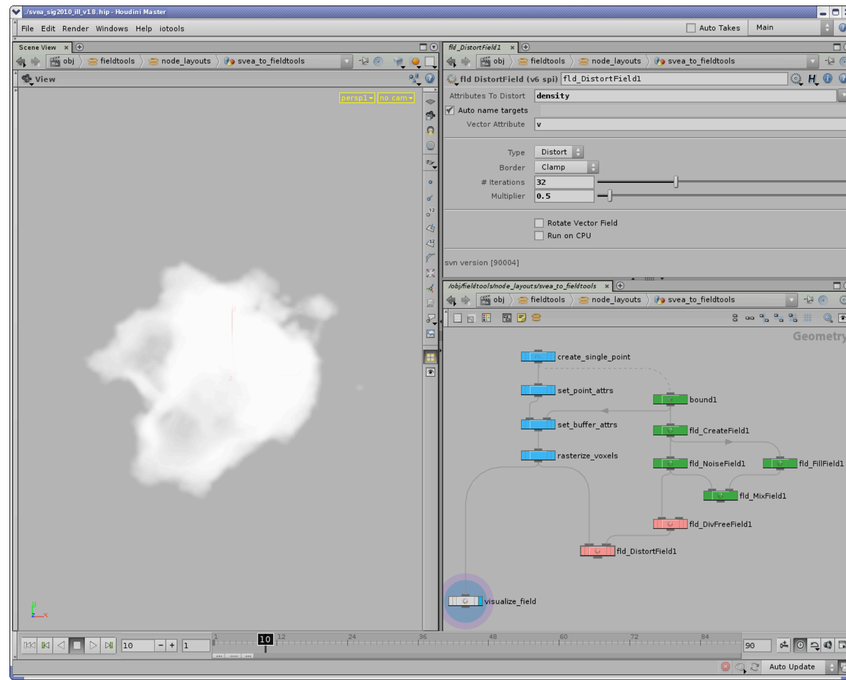
To illustrate how the various tool sets are integrated we will now see how some simple tasks are accomplished in the volume pipeline.

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.

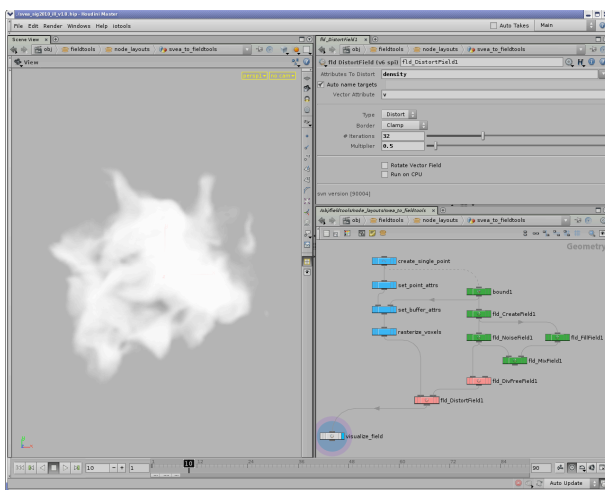


Processing level set data using FieldTools operations

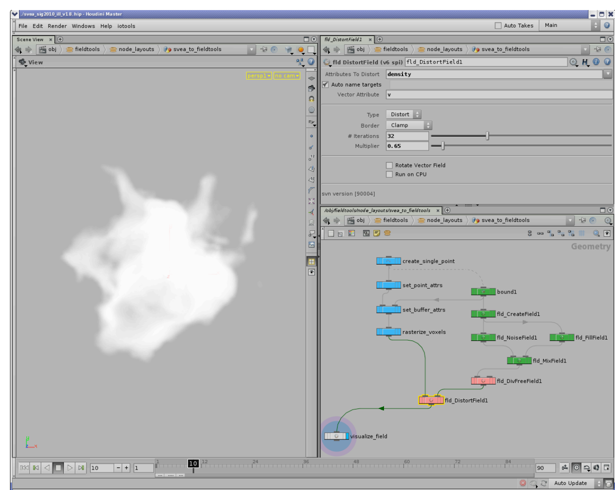
The next examples illustrates how the various tools can work together across functional boundaries to accomplish a task. All data is shared between the various plugins, and the voxel buffer created inside the voxel rasterization node is available directly to the distortion node. In this case, the resulting 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.



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

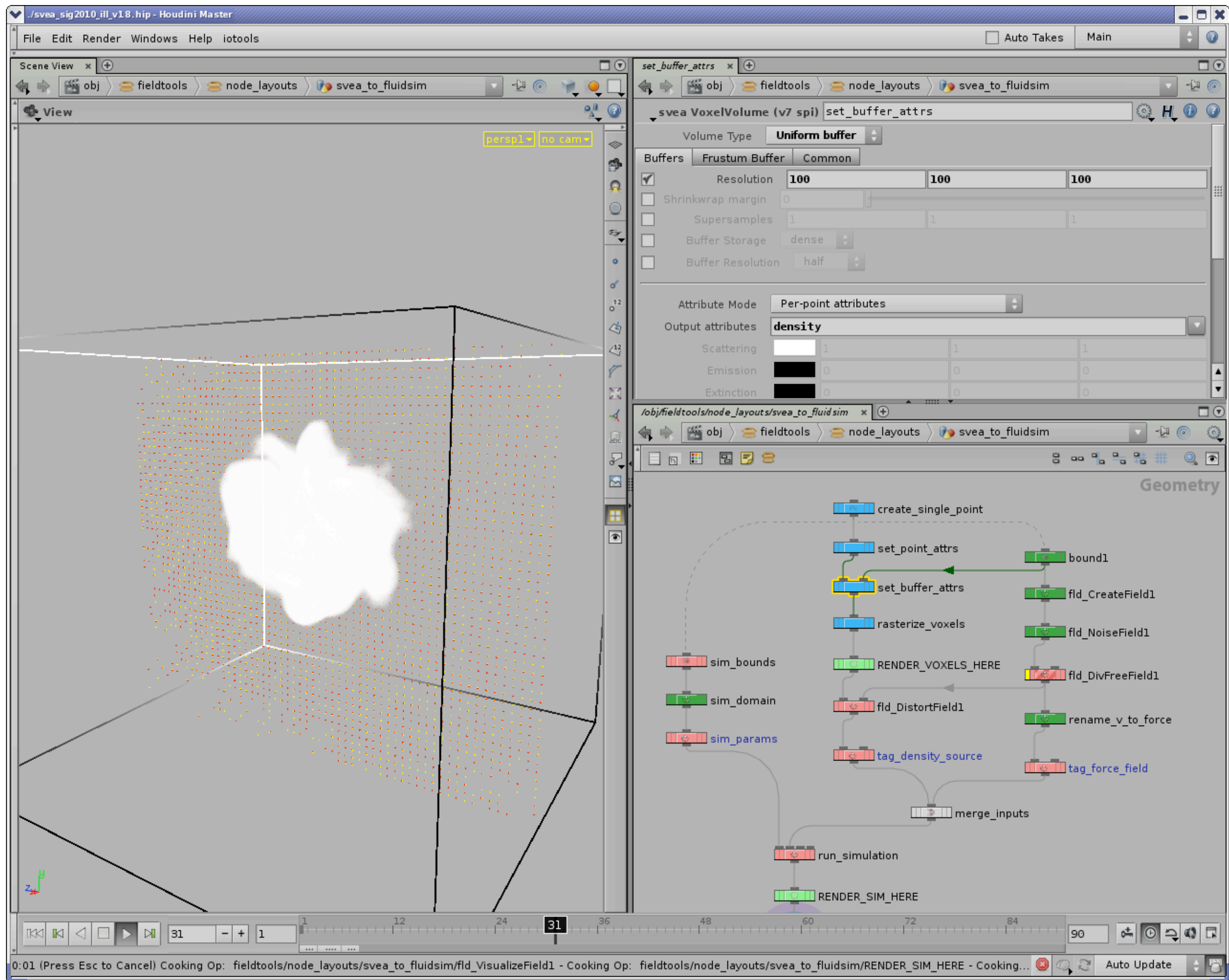


Svea point primitive distorted by noise vector field



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

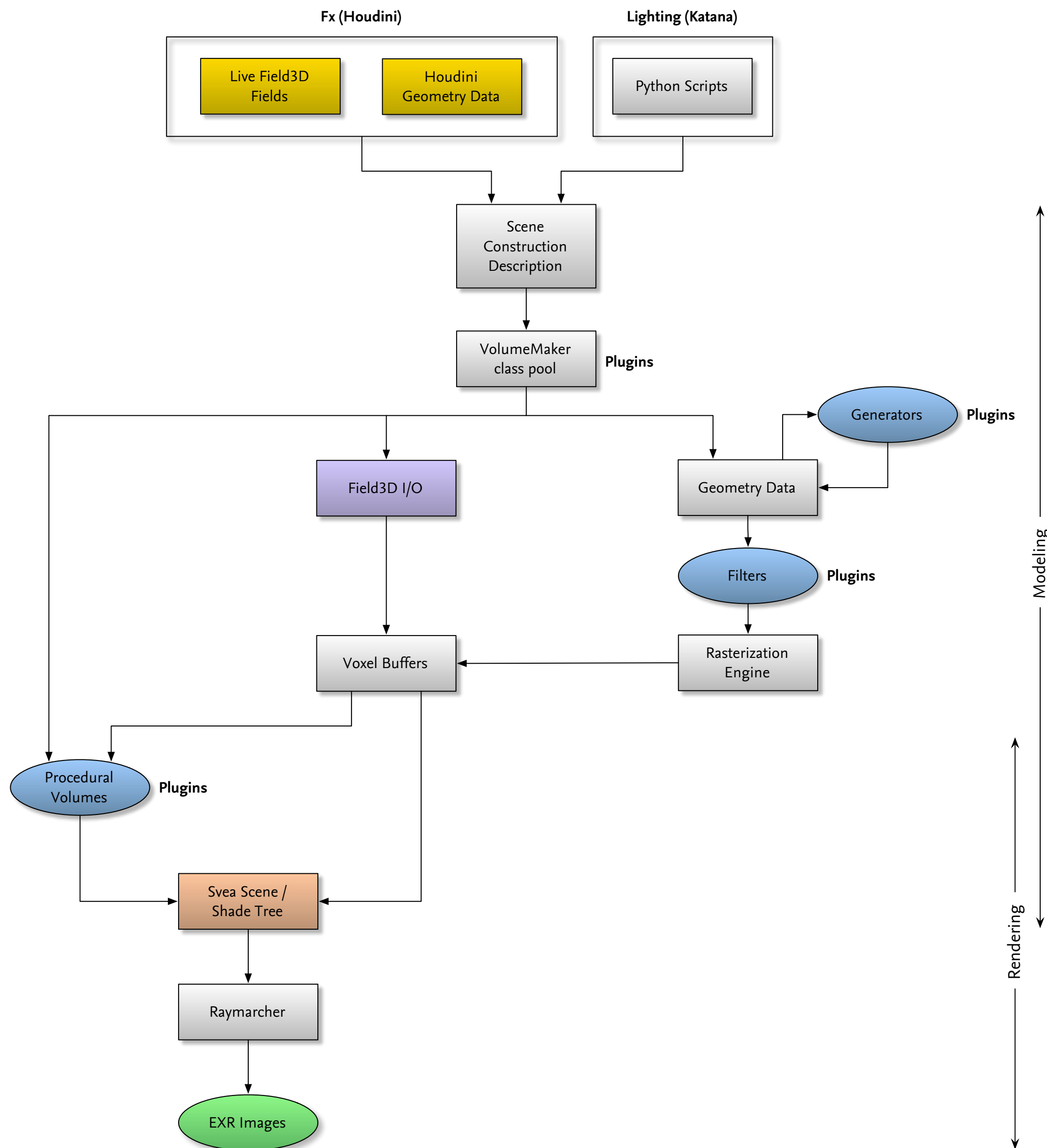
In this third example, the same noise point is animated and used as a density source in a fluid simulation. The flexibility given by this level of integration and direct interaction between components is important in production work, where each added step in a process is a potential source of errors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



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 pipeline



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.

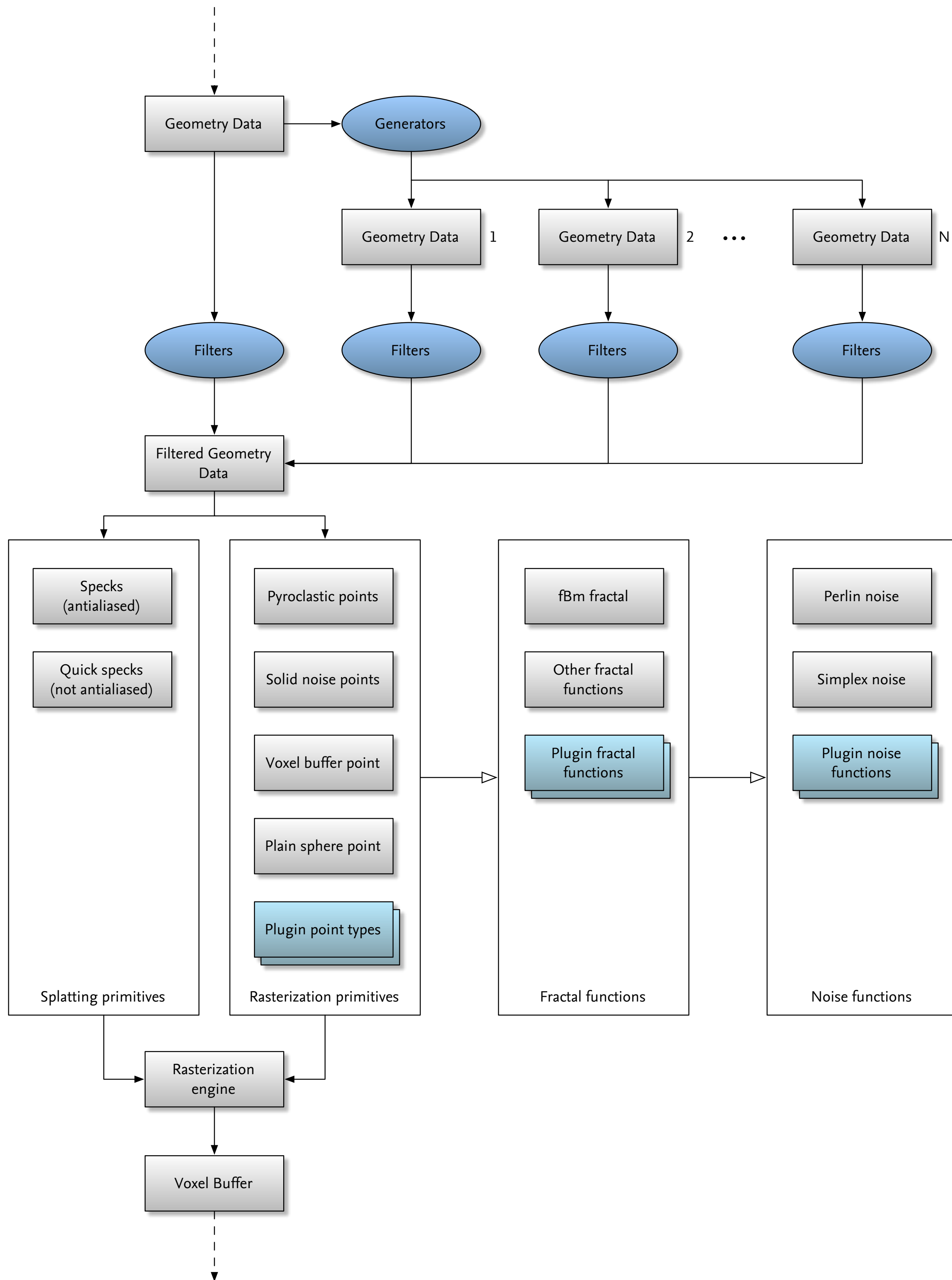
- *N* `GeometryData` objects
  - Detail (global) attributes
  - Primitives, primitive attributes and primitive groups
  - Points, point attributes and point groups

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.

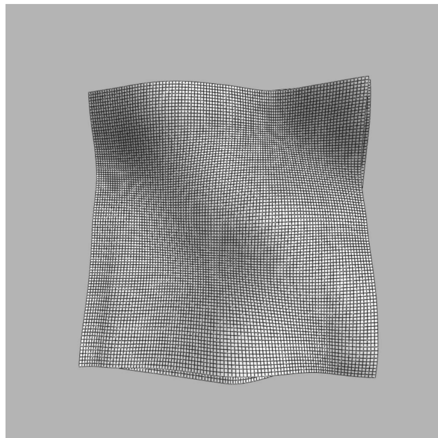


3.1. Generators

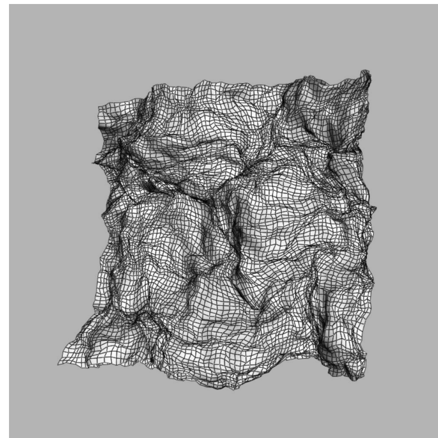
Svea's mechanism for instantiation-based primitives is called `Generators`. During the volume modeling process, each `GeometryData` instance can spawn an arbitrary number of `Generator` objects, each getting access to the `GeometryData` of its creator. This is done so that a single `Generator` object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

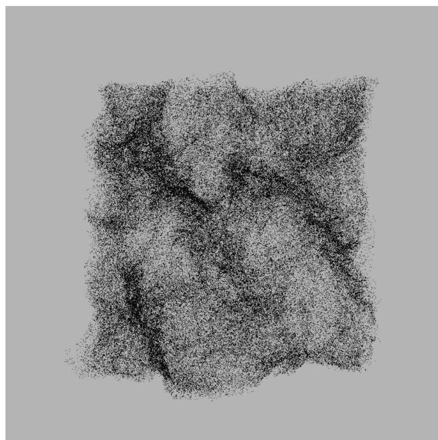
- **Speck Spline –** curve-based point instantiation.
- **Speck Surface –** surface-based point instantiation.
- **Cluster –** a space-filling algorithm that instantiates points based on the configuration of an input particle system. Used for everything from white water to mist and smoke-like effects.



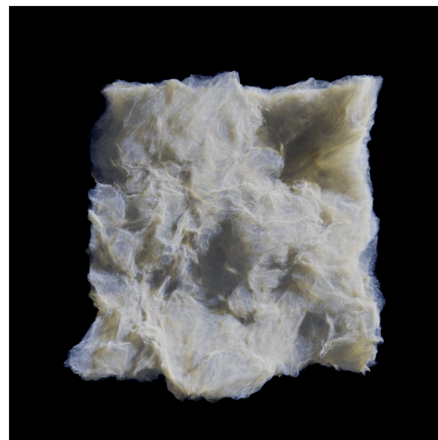
Underlying primitive



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.
- **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.

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:

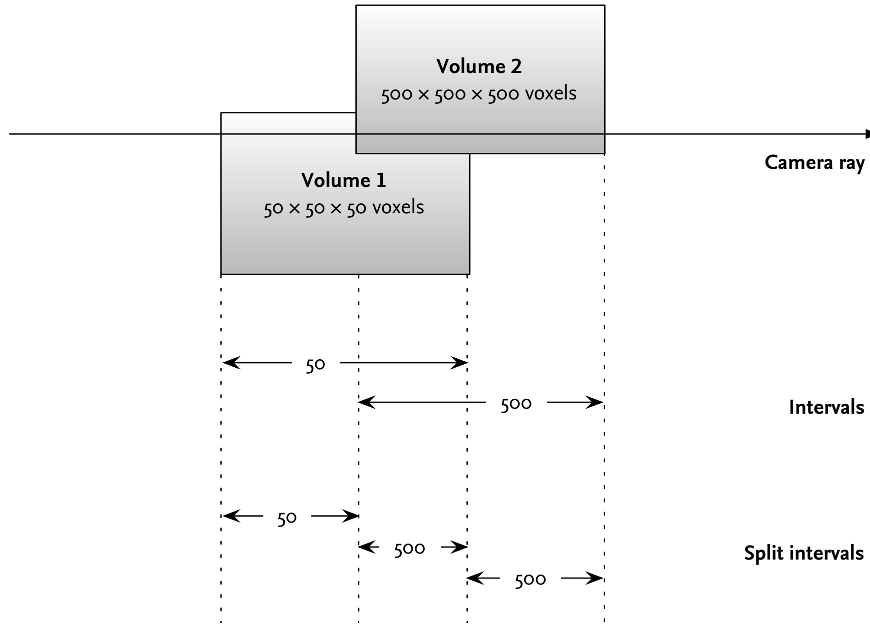
```
class Volume:
{
public:
   virtual Color value(const Attribute &attribute, const SampleState &state);
   virtual void getIntersections(const Ray &ray, std::vector<RaymarchInterval> &outIntersections);
};

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.

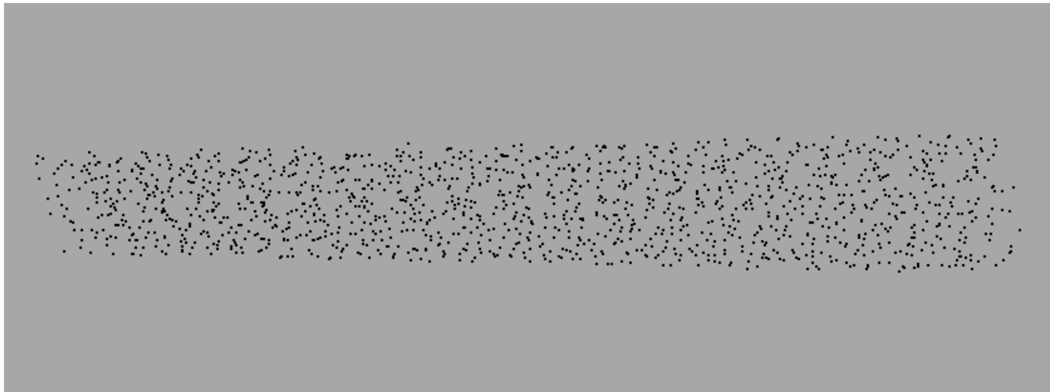
In slightly simplified code (ignoring lighting calculations, holdouts, etc.), the algorithm can be implemented as:

```
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;
```

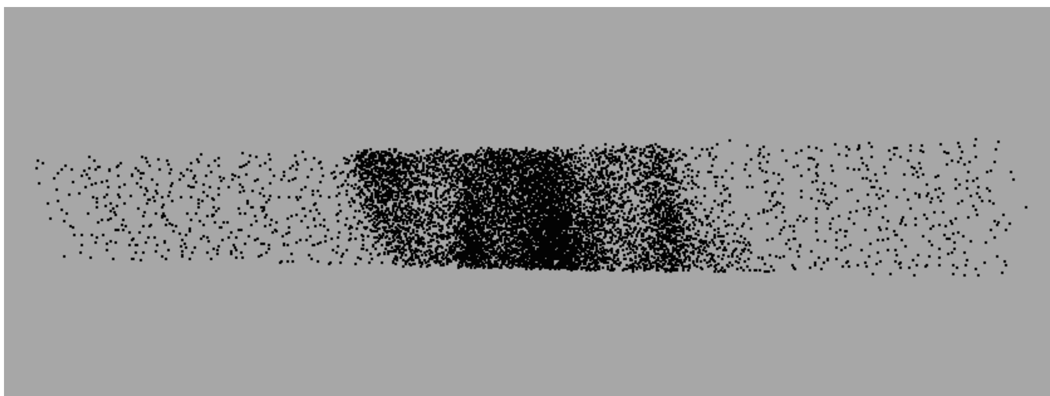
```
    // Sample point
    float t = (step_t0 + step_t1) * 0.5f;
    // Calculate incoming luminance and optical thickness
    lightingCalculation(scene->root, ray(t), &Lstep, &tau);
    // Account for current transmittance and step length
    Lstep *= T * stepLength;
    // Before applying the result, 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.

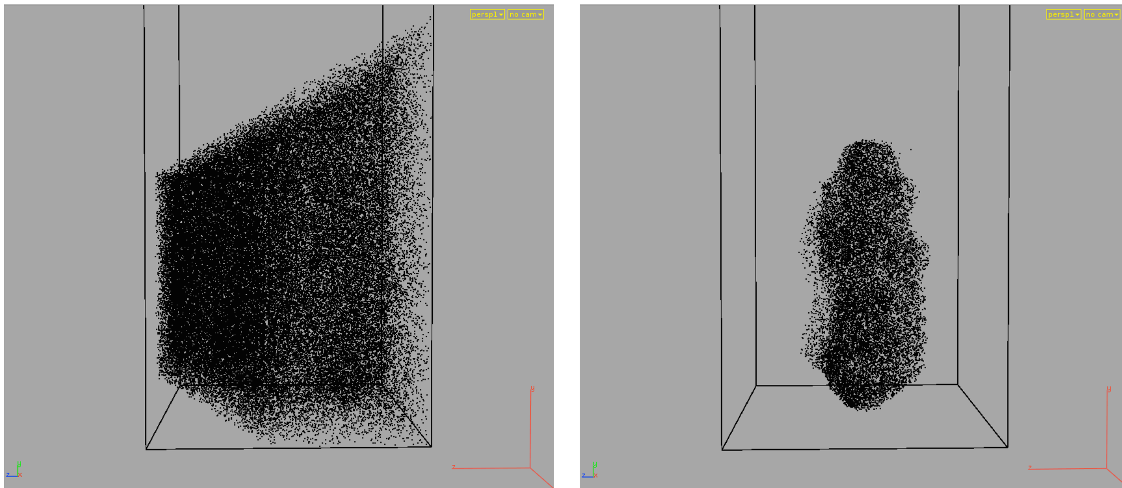
The following pseudo-code implements construction of a level set that also takes into account motion blur:

```
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:

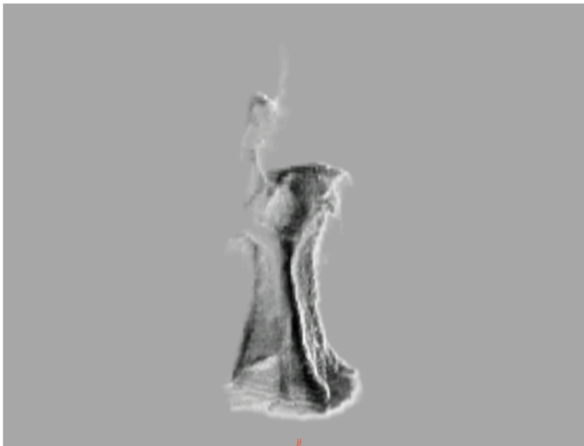


Plot of raymarch samples after intersecting primary rays against the simulation domain (left) and against a level set (right)

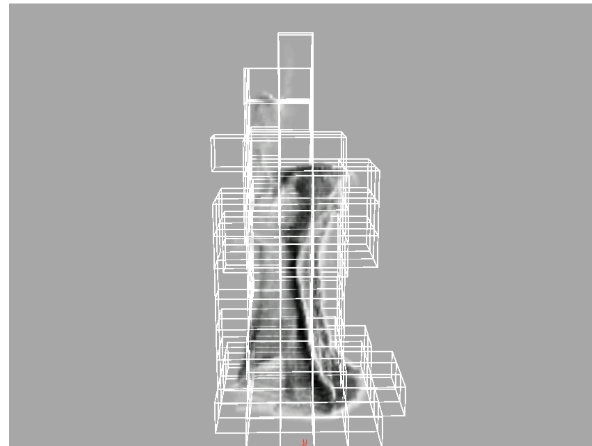
4.4.2. Using the buffer's data structure

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.

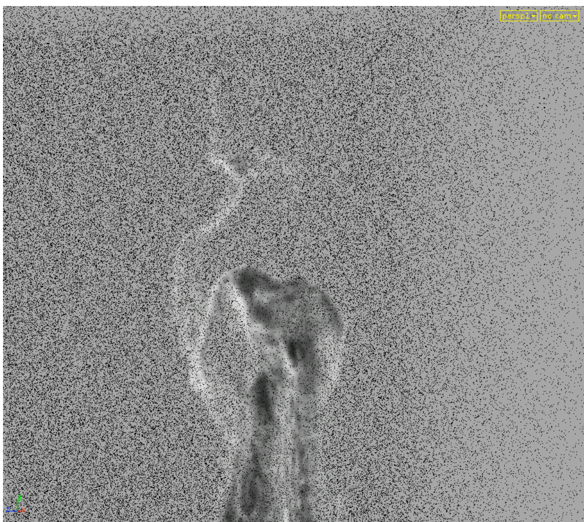
Intersecting an array of blocks is straightforward, and can be implemented either by testing the bounds of each block individually against the ray, or by using a 3D line drawing algorithm to find the blocks that overlap the ray's path.



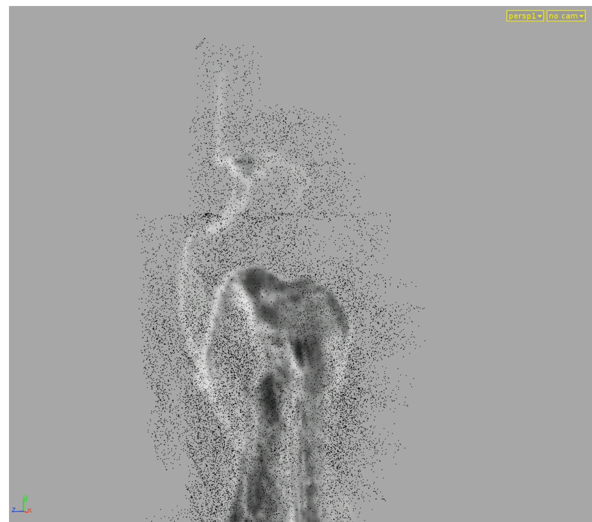
Fluid simulation element



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.

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

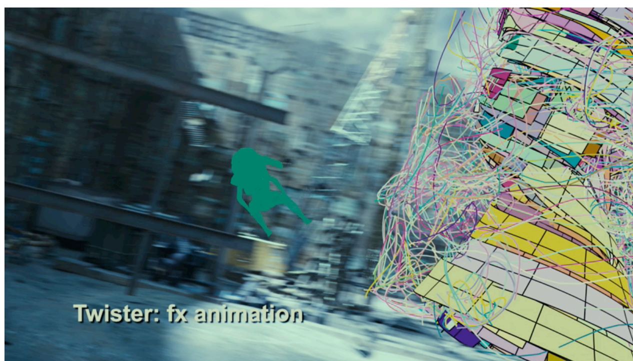
One of the major effects sequences in Hancock features tornadoes sweeping down on Hollywood Boulevard. The storm clouds in the background were modeled using existing rasterization primitives but the volume renderer had to be extended on several fronts in order to accomplish the tornado effects.

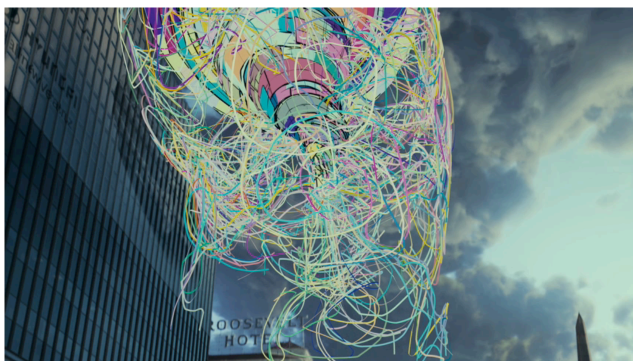
The first new development was the Generator concept, which is how Svea implements instantiation-based primitives. A surface-based primitive was used to model the core of the tornado, and a curve-based one was used to create wispy features that tore off the main funnels. The illustrations on the next 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.

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[3] 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[4], 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 $4000 \times 4000 \times 3000$, 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.

---

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

[4] 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.