Resolution Independent Volumes

Jerry Tessendorf
School of Computing, Clemson University

Michael Kowalski
Rhythm and Hues Studios

July 24, 2011

=1
= 1.5
= 1.5
hildren = 1000000000
ves = 5
n = 0.5

i

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

Forward

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

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

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

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

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

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

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

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

Contents

List of Figures

Chapter 1

Introduction

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

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

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

In addition to modeling volume data, Felt also modifies geometry, particles, and volume data generated with other tools, including animations and simulations. This gives fine-tuning control over data in a post-process, similar to the way a compositor can fine-tune images after they are generated. Conversely,

simulations can use FELT during their runtime to modify data and processing flow to suit special needs.

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

1.1 A Brief on Volume Rendering

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 Some Conventions

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

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

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

This simple script is equivalent to the mathematical notation:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 2

The Value Proposition for Resolution Independence

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

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

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

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

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

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

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

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

Performance Trade-Offs

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

Targeted grid usage

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

Procedural high resolution

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

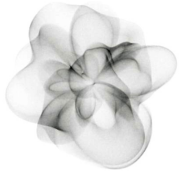
Cleaner coding of algorithms

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

Calculations only where/when needed

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

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



Chapter 3

Cloud Modeling

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

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

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

3.1 Cumulous cloud structure of interest

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

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

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

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

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

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

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

3.2 Levelset description of a cloud

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

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

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

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

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

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

3.3 Layers of pyroclastic displacement

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

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

3.3.1 Displacement of a sphere

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

1. Calculate the distance from the point of interest $\mathbf{x}$ to the center of the sphere $\mathbf{x}_{\text{sphere}}$:
$$d = |\mathbf{x} - \mathbf{x}_{\text{sphere}}| \tag{3.2}$$

2. Compare d to the displacement bound d_{bound} of the Perlin noise and the radius R of the sphere. If $d < R$, $\mathbf{x}$ is definitely inside the pyroclastic

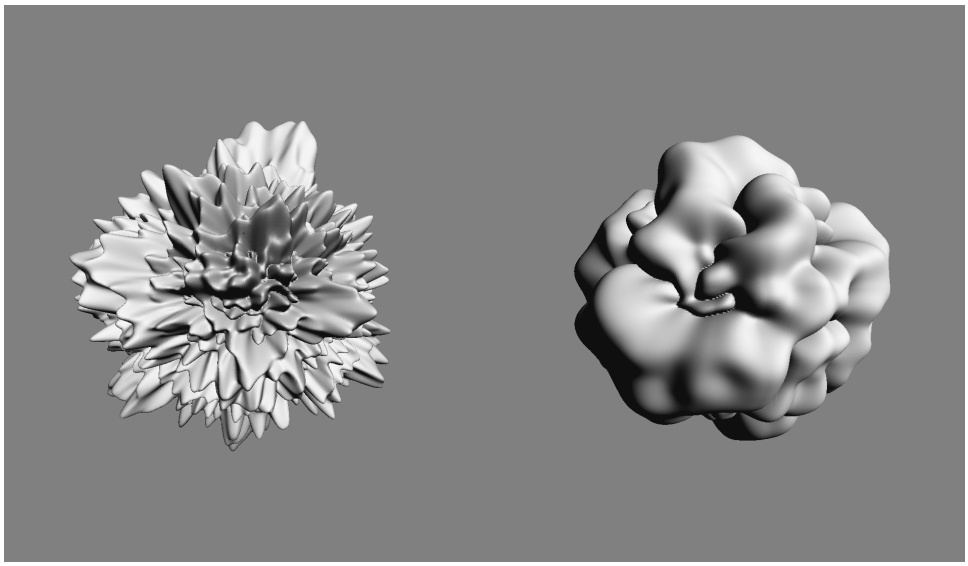


Figure 3.2: Examples of classic pyroclastically displaced spheres of density.

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

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

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

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

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

3.3.2 Displacement of a levelset

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

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

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

   for which typically 3-5 iterations are needed.

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

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

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

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

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

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

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

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

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

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

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

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

3.3.3 Layering strategy

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

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

Fractal layering

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

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

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

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

In terms of FELT code, we have:

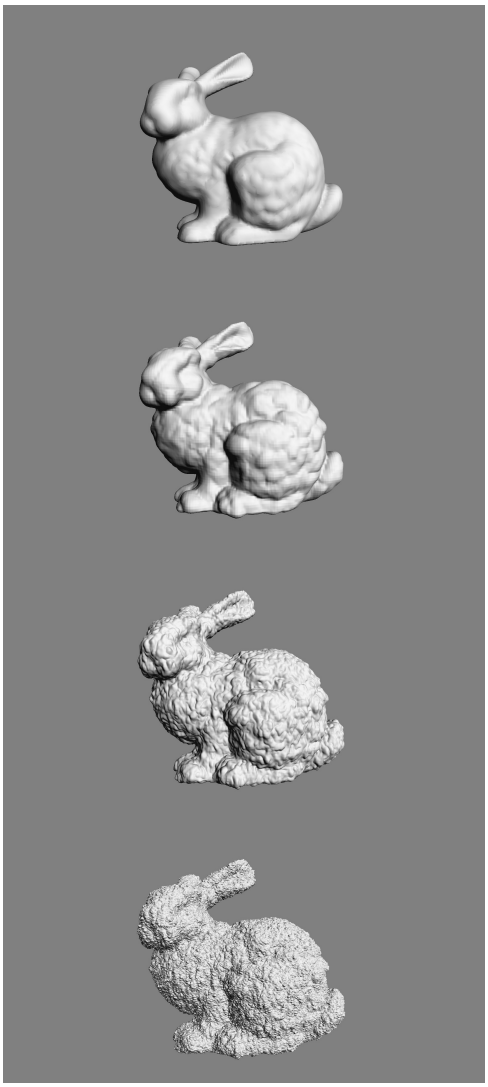


Figure 3.3: Illustration of layering of pyroclastic displacements. From top to bottom: No displacements; one layer of displacements; two layers; three layers. The displacements are applied to the levelset representation of the bunny, and the displaced bunny was converted into geometry for display.

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

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

Selectively sampling the levelset into grids

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

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

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

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

3.4 Clearing Noise from Canyons

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

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

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

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

over most of the volume, with the exception of narrow regions at the peak of displacement.

3.5 Advection

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

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

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

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

3.6 Spatial control of parameters

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

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

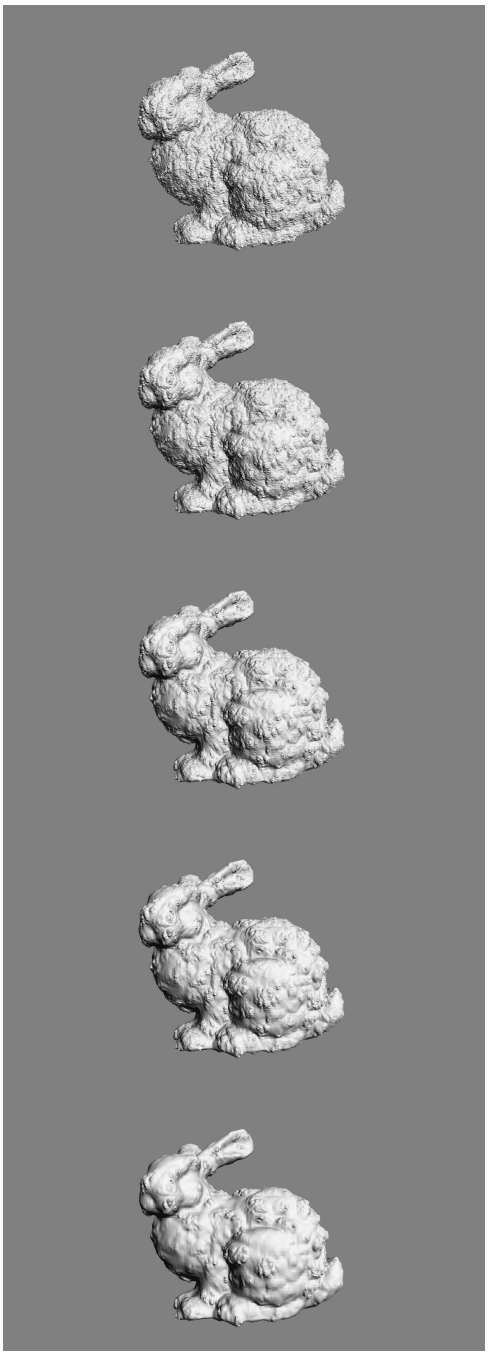


Figure 3.4: Illustration of clearing of displacements in the valleys using the billow parameter. The bottom of figure 3.3 illustrates the three layers of displacement with no billow applied. The noise is FFT-based, and $Q = 1$. From top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

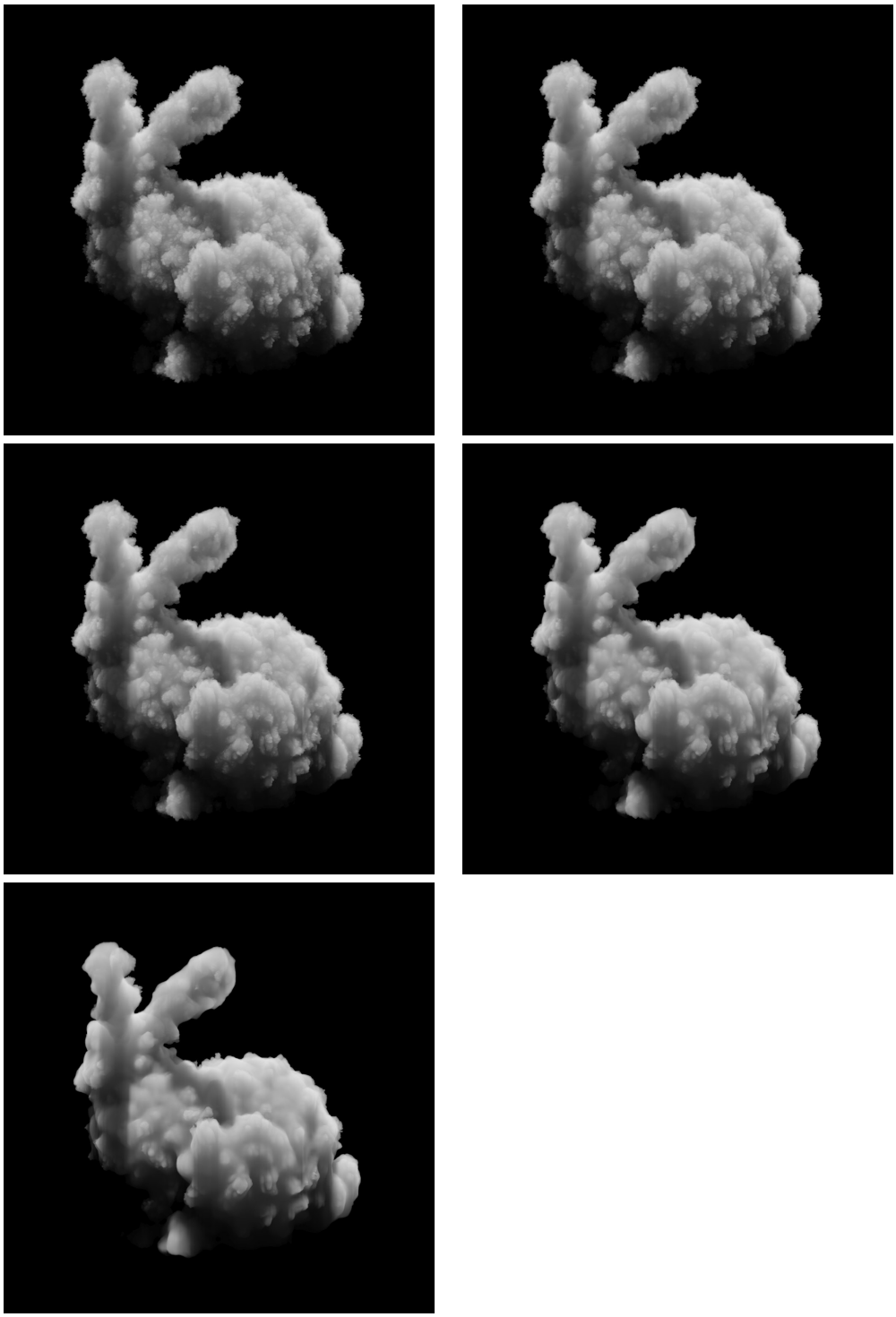


Figure 3.5: Volume renders with various values of billow. Left to right, top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

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

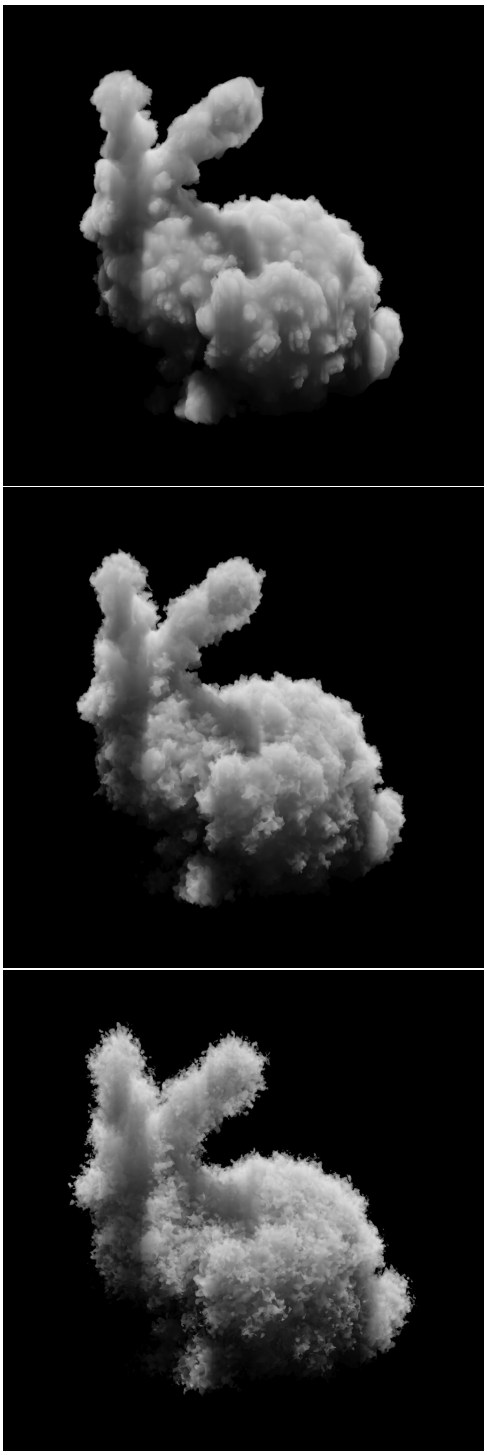


Figure 3.7: Volume renders with various setting of advection, for billow=1. Top to bottom: No advection, medium advection, strong advection.

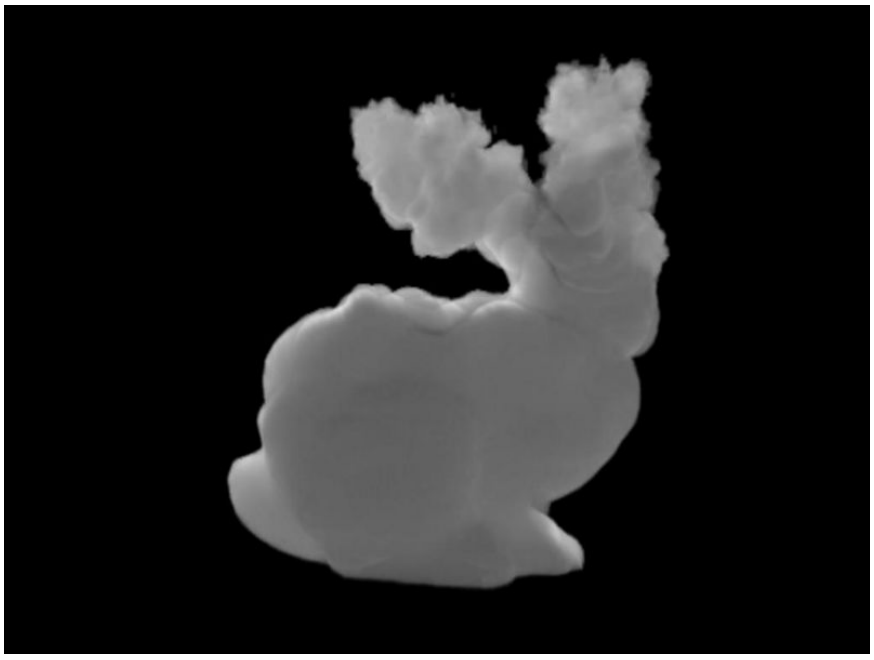


Figure 3.8: Volumetric bunny with spatial control over the pyroclastic displacement.

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

Chapter 4

Warping Fields

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

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

4.1 Nacelle Algorithm

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

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

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

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

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

Expanding this to quadratic order in Taylor series gives

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

Define matrix $\mathbf{M}$ as

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

so the Taylor expansion up to quadratic is

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

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

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

which has the solution

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

with the abbreviations

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

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

Then the next approximation is

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

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

4.2 Numerical implementation

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

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

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

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

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

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

4.3 Attribute transfer

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

Warp Levelsets

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

Attribute transfer

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

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

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

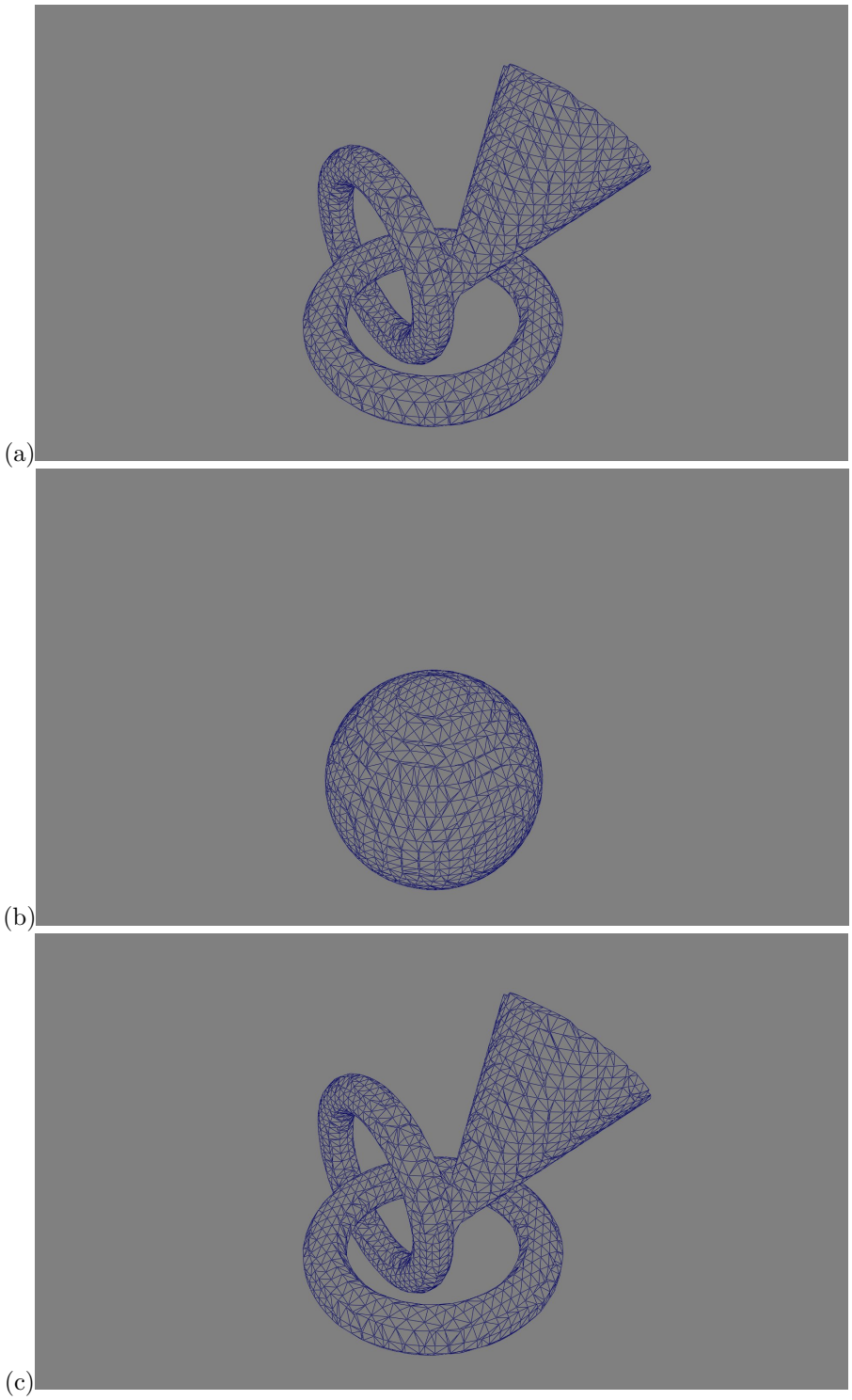


Figure 4.1: Warping of a reference sphere into a complex shape (cone and two torii). (a) Object shape; (b) Reference sphere; (c) Warp shape output from 1 iteration.

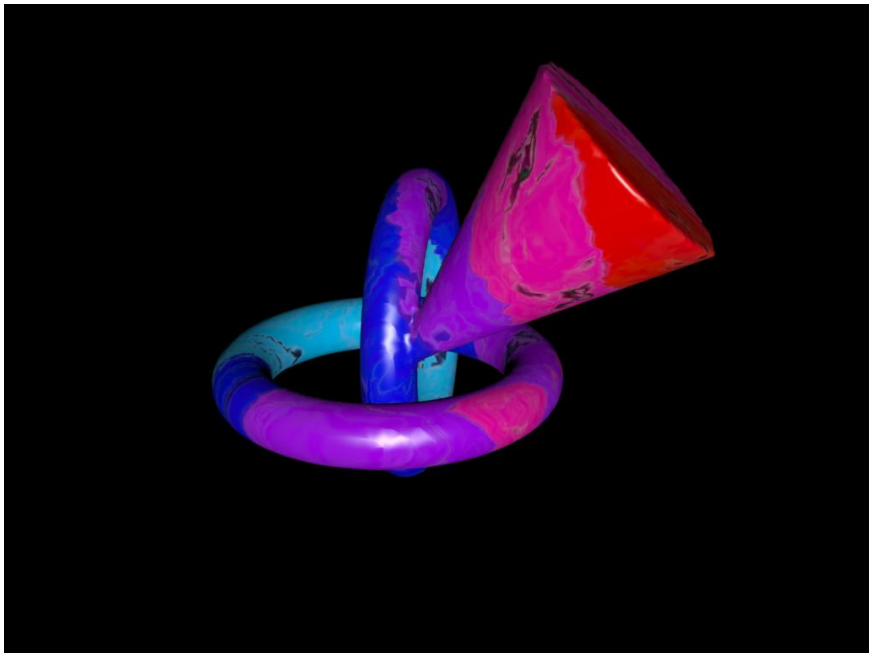


Figure 4.2: Texture mapping of the object shape by transfering texture coordinates from the reference shape.

Chapter 5

Cutting Up Models

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

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

5.1 Levelset knives

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

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

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

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

5.2 Single cut

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Multiple cuts

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

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

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

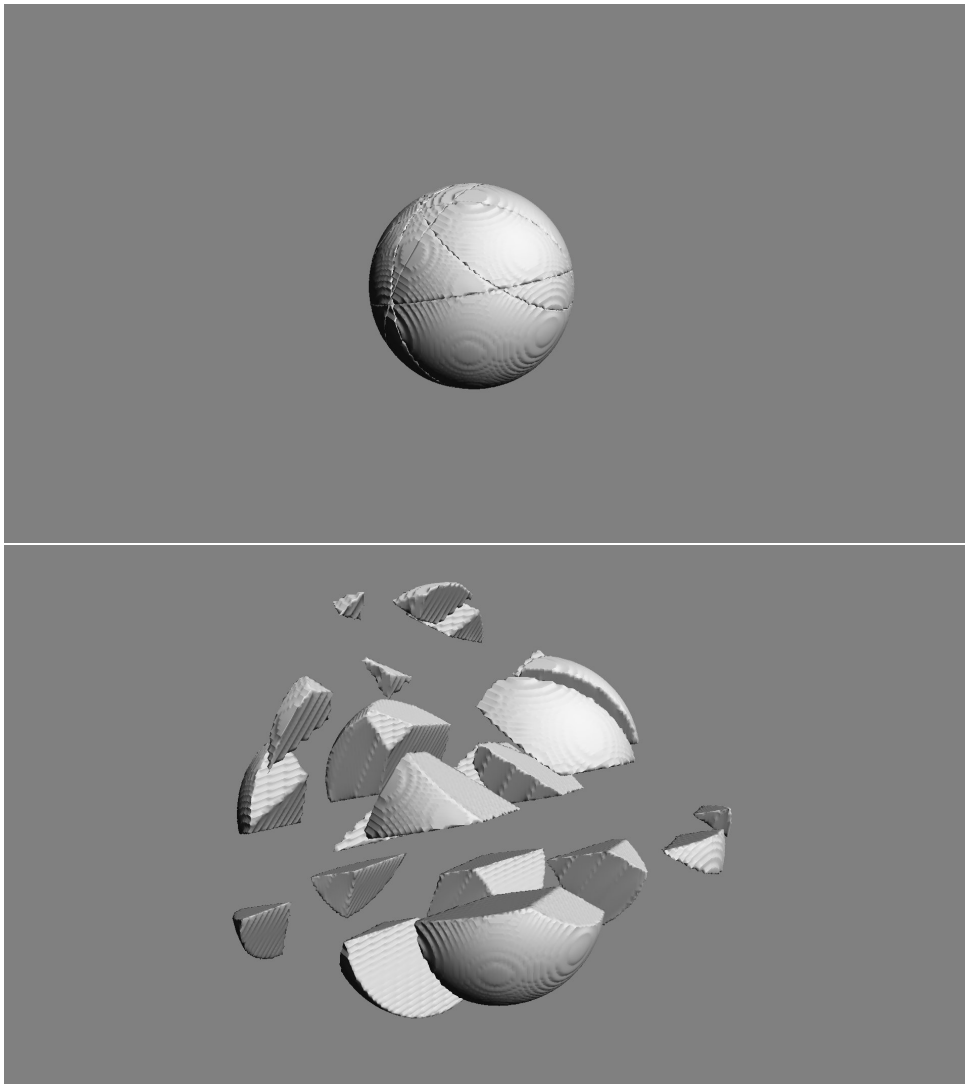


Figure 5.1: A sphere carved into 22 pieces using 5 randomly placed and oriented flat blades. The top shows the sphere with the cuts visible. The bottom is an expanded view of the pieces.

Chapter 6

Fluid Dynamics

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

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

6.1 Navier-Stokes solvers

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

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

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

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

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

1. Advect the density with the current velocity

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

2. Advect the velocity and add external forces

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

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

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

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

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

6.1.1 Hot and Cold simulation scenario

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

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

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

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

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

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

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

6.2 Removing the grids

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

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

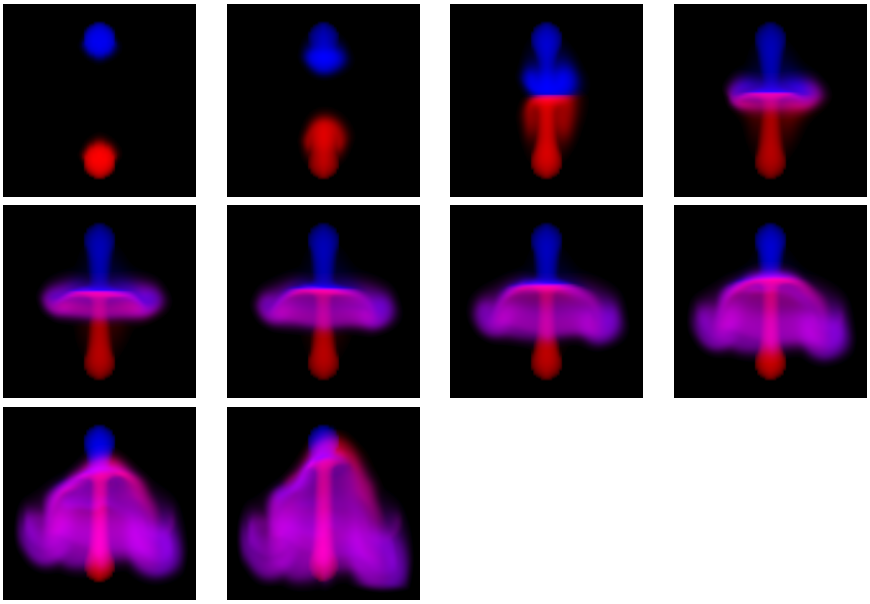


Figure 6.1: Simulation sequence for hot and cold gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

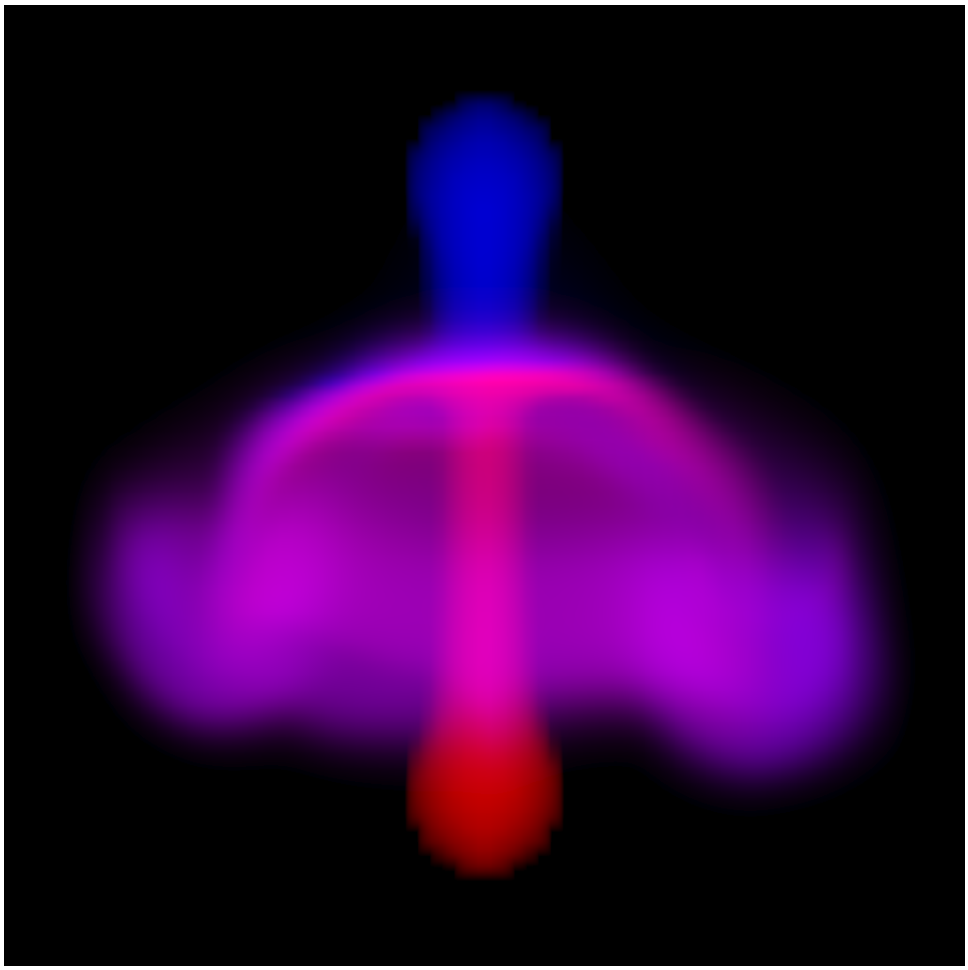


Figure 6.2: Frame of simulation of two gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

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

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

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

6.3 Boundary Conditions

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

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

The FELT code for this is

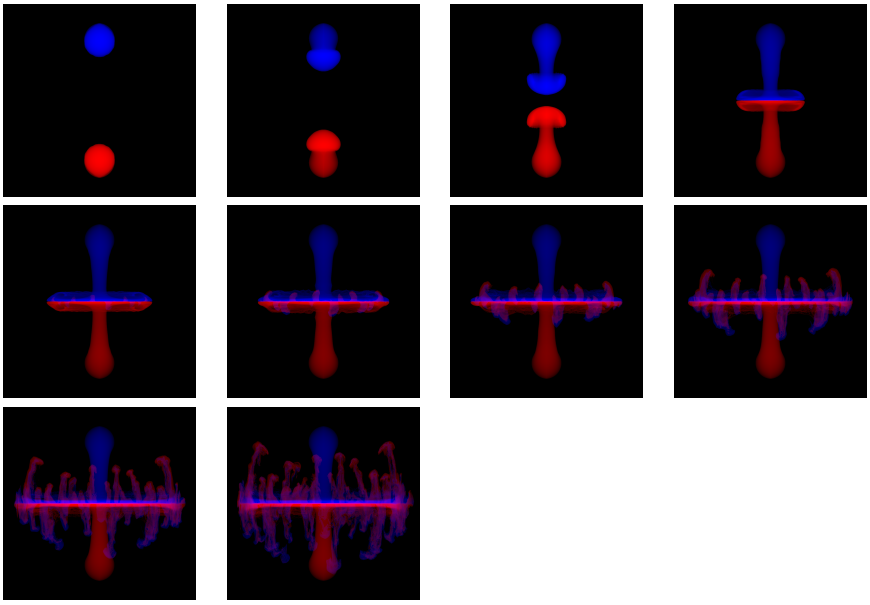


Figure 6.3: Sequence of frames of a simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected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

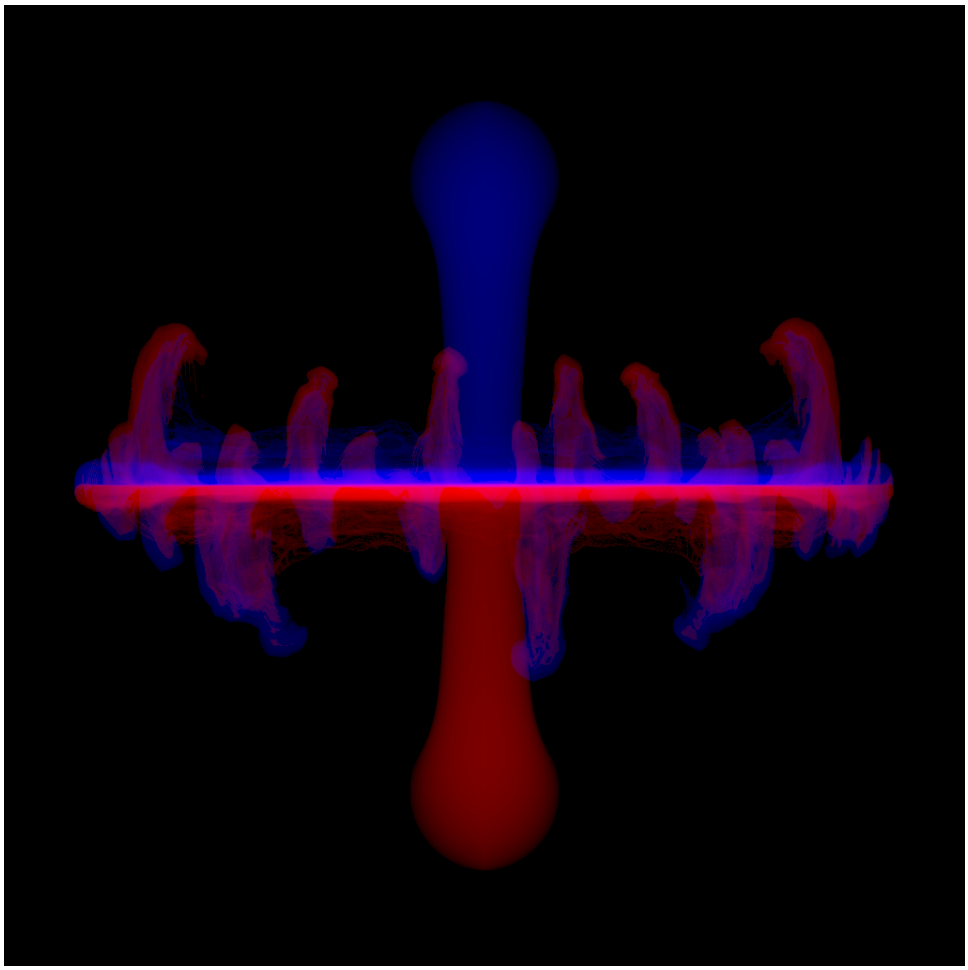


Figure 6.4: Frame of simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected but not sampled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

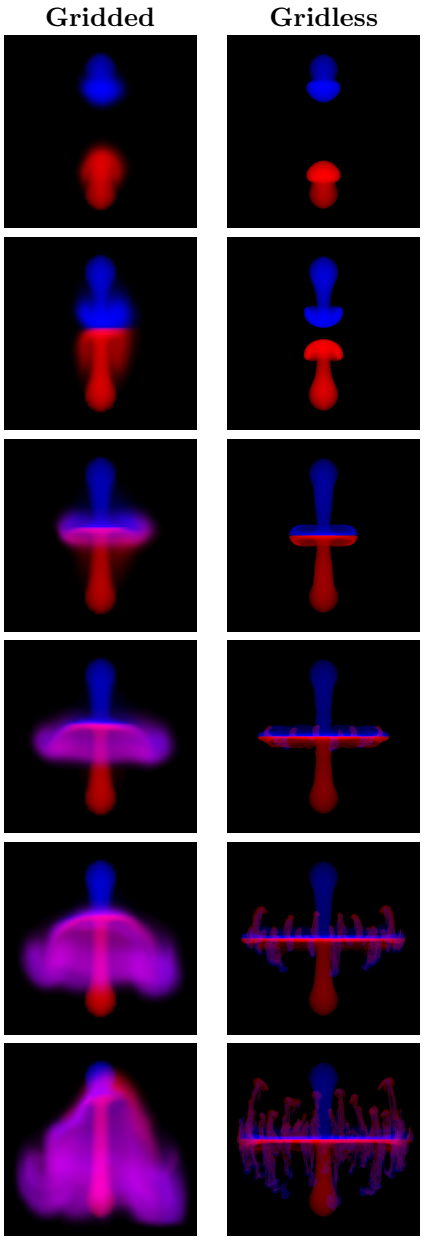


Figure 6.5: Simulation sequences with density gridded (left) and gridless (right). The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution is $50 \times 50 \times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

```
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.

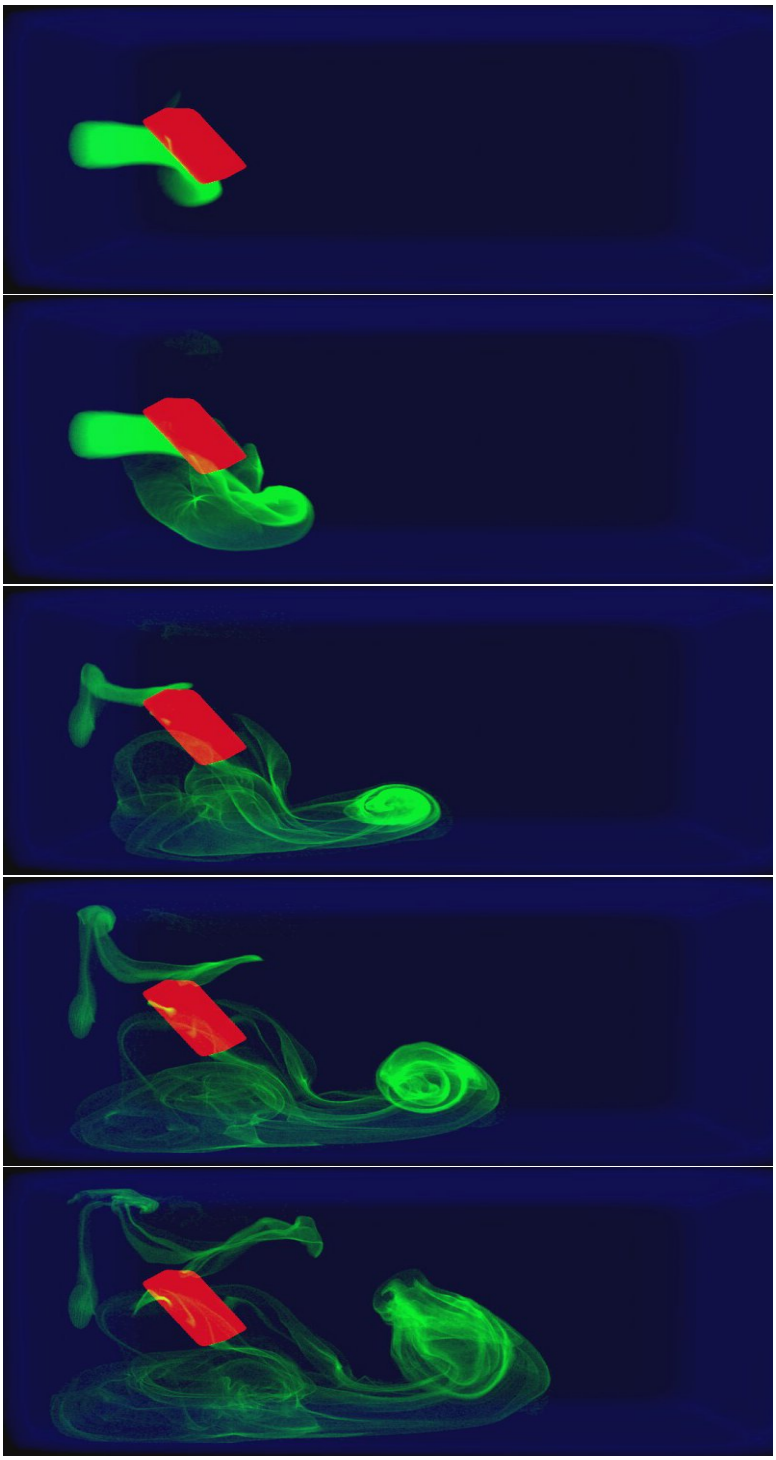


Figure 6.6: Time series of a simulation of bouyant flow (green) confined within a box (blue boundary) and flowing around a slab obstacle (red). Frames 11, 29, 74, 124, 200 from a 200 frame simulation.

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 Spatial Gradients

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

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

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

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

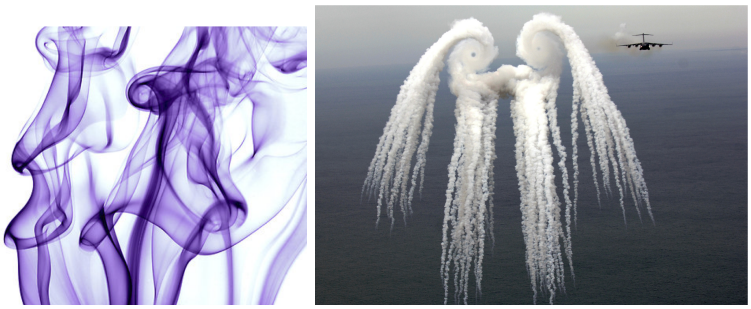


Figure 7.1: Examples of filaments and sheets forming in fluid flow.

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

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

7.2 Algorithm

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

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

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

This can be extended to two steps of advection:

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

and to three steps of advection:

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

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

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

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

7.3 Spatial Gradients in Gridless Advection

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

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

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

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

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

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

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

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

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

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

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

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

7.4 Examples

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

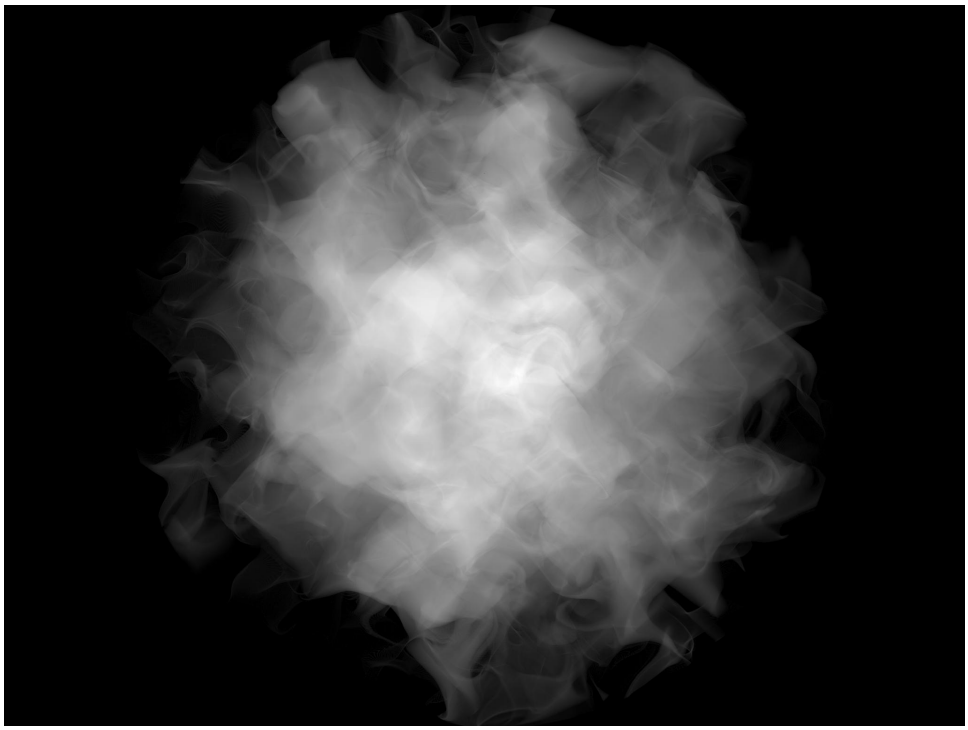


Figure 7.2: Illustration of the effect of a single step of gridless advection. The unadvected density field is a sphere of uniform density.

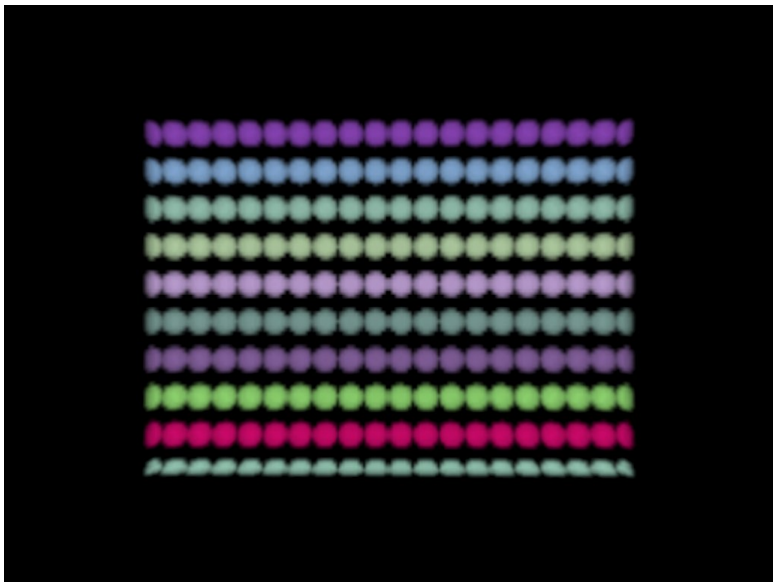


Figure 7.3: Unadvected density distribution arranged from a collection of spherical densities.

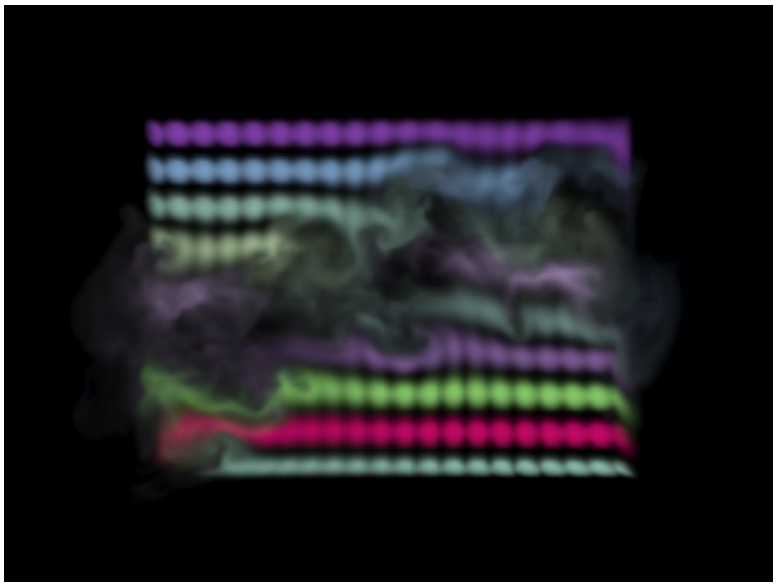


Figure 7.4: Density distribution after 60 frames of advection and sampling to a grid each frame.

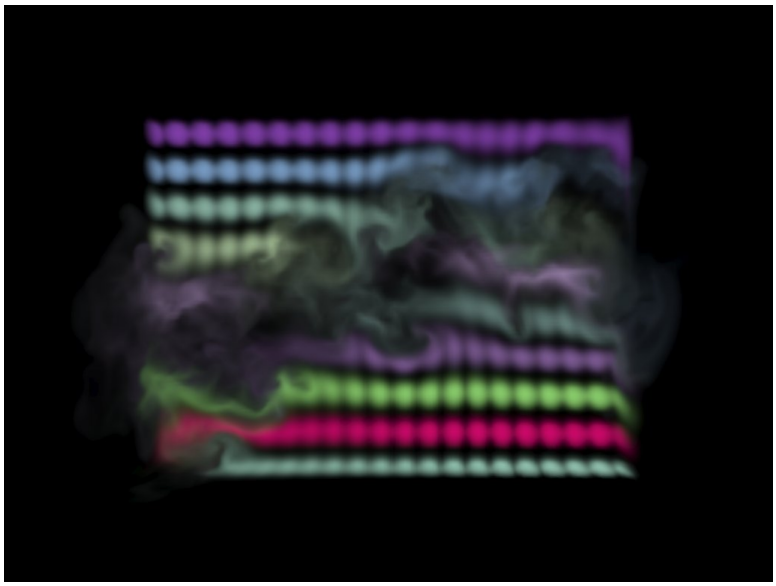


Figure 7.5: Density distribution after 59 frames of advection and sampling to a grid each frame, and one frame of gridless advection. The edges of filaments have been subtley sharpened.

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

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

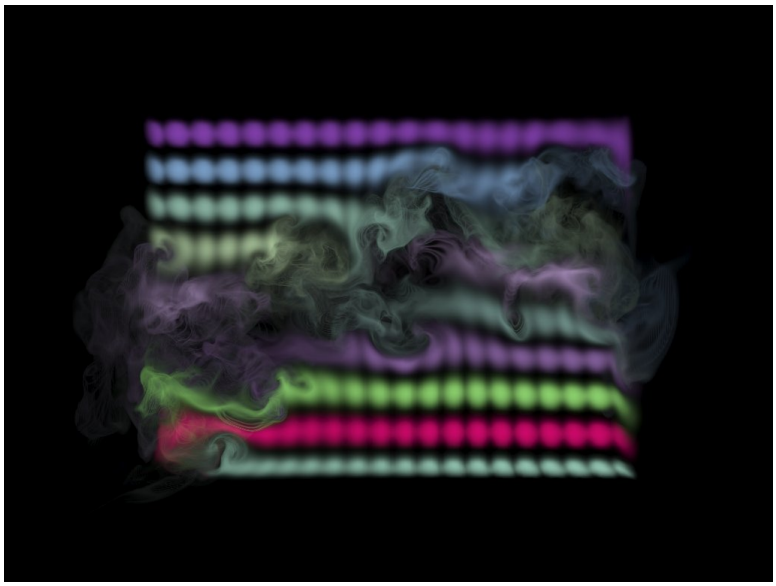


Figure 7.6: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The sharpening of details has increased to the point that the detail is finer than the raymarch stepping, causing significant aliasing in the render.

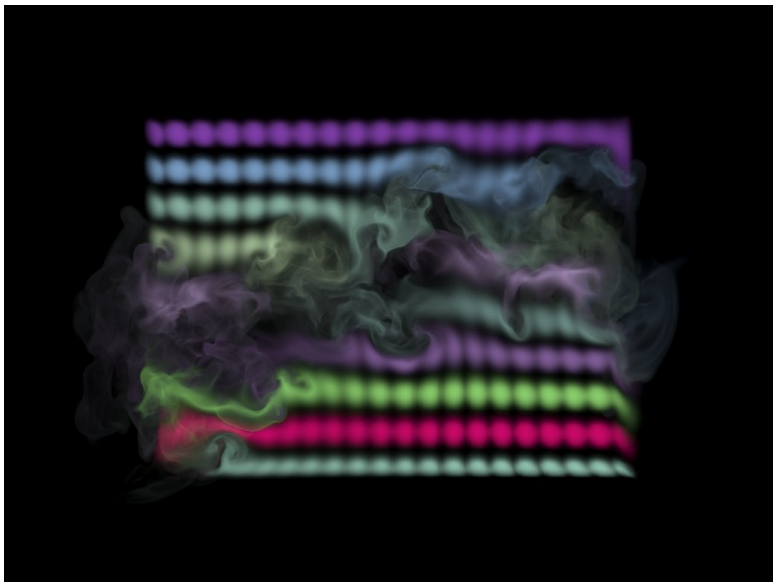


Figure 7.7: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The fine detail in the density field is now resolved by using a finer raymarching step (1/10-th the grid resolution).

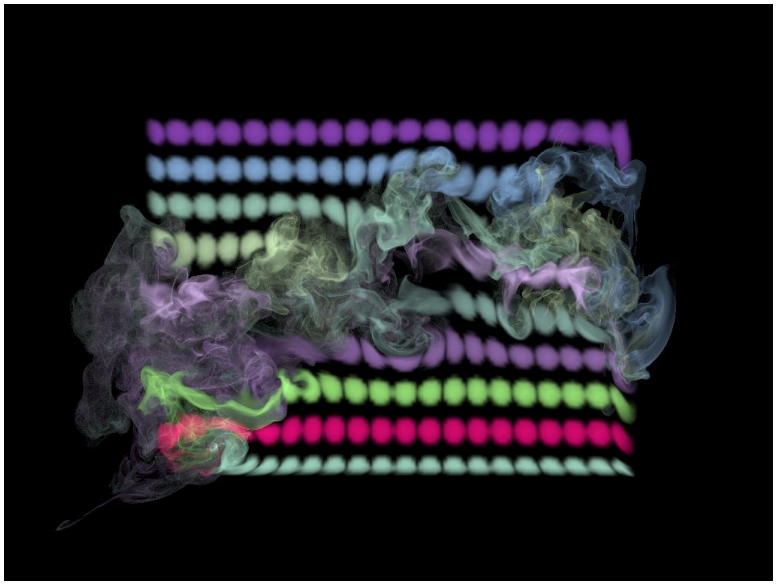


Figure 7.8: Density distribution after 60 frames of gridless advection. The fine detail in the density field is resolved by using a fine raymarching step.

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

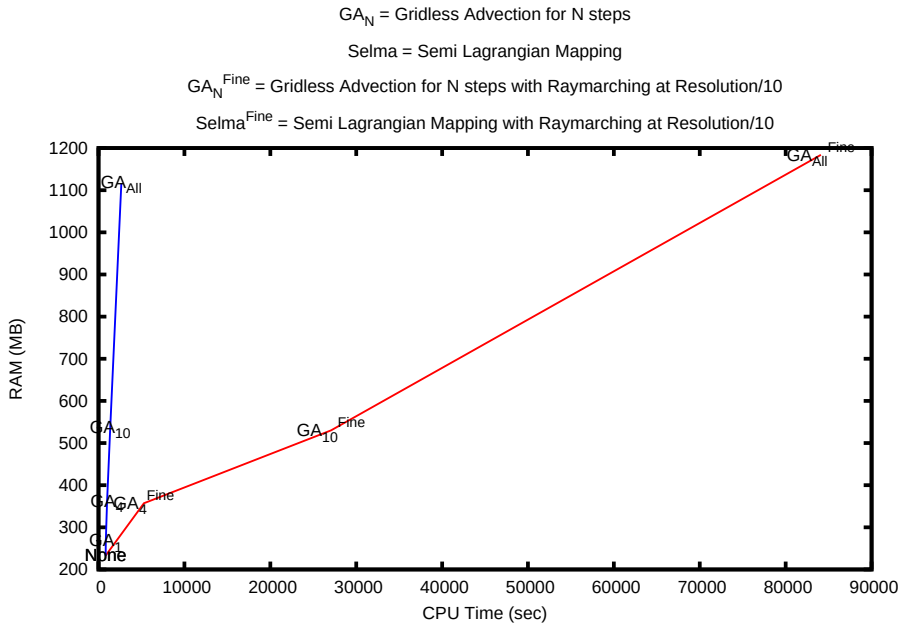


Figure 7.10: Performace of gridless advection as the number of advection frames grows. The steep blue line is gridless advection rendered with the raymarch step equal to the grid resolution. The red line is a raymarch step equal to one-tenth of the grid resolution. These results are not from a production-optimized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

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

advection for more than about 5-10 steps extends the render time, due to the additional advection evaluations and the finer raymarch stepping, to the limit that most productions choose to take.

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

Chapter 8

SEmi-LAgrangian MApping (SELMA)

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

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

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

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

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

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

where

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

and

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

where

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

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

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

with an iterative form for the mapping:

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

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

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

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

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

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

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

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

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

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

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

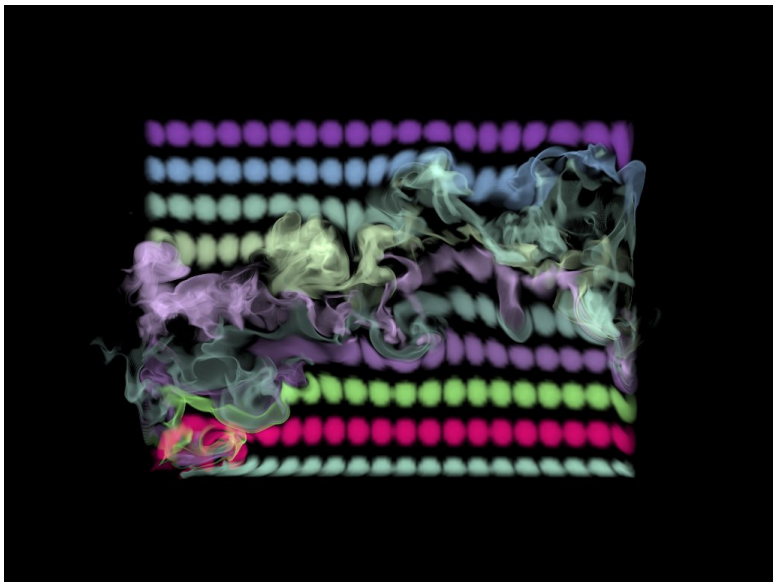


Figure 8.1: Density distribution after 60 frames of SELMA advection. The fine detail in the density field is resolved by using a fine raymarching step.

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

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

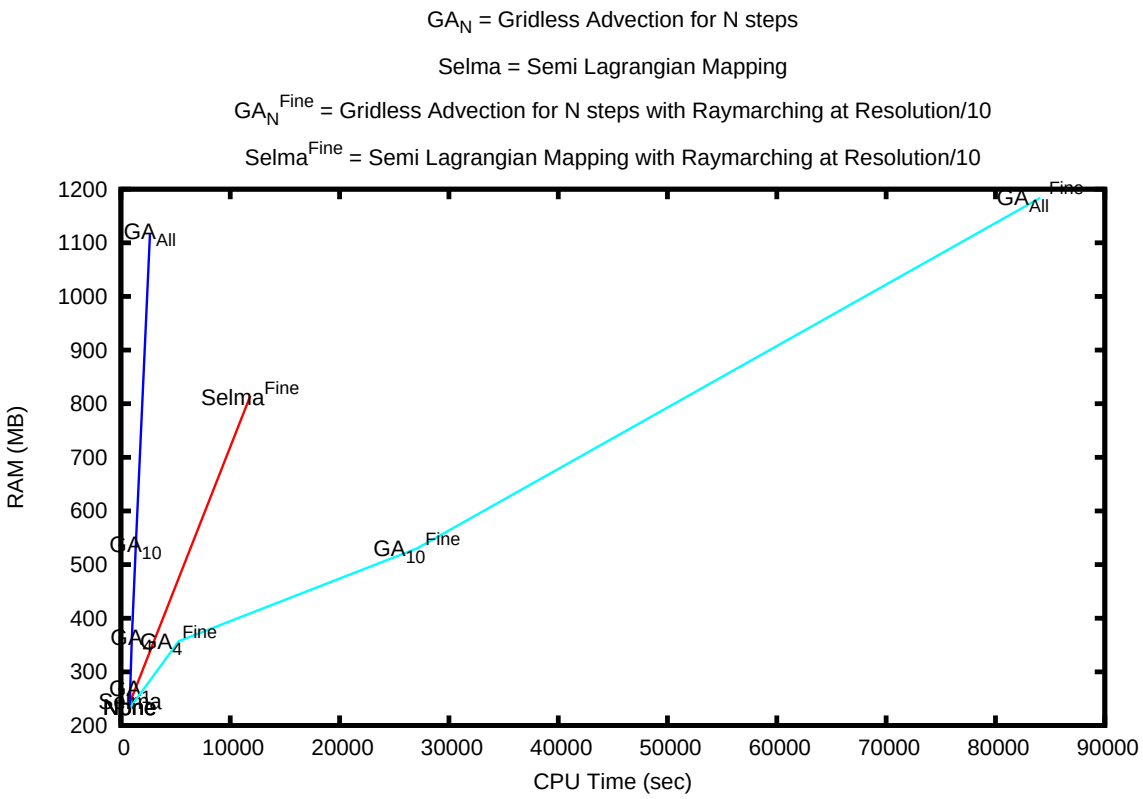


Figure 8.2: Comparison of the performace of Gridless Advection and SELMA.

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

Appendix A

Appendix: The Ray March Algorithm

A.0.1 Rendering Equation

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

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

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

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

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

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

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

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

with the light transmissivity being

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

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

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

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

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

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

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

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

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

$$\delta \;=\; \frac{4}{3(n-1)^2} \; \sin^2\left(\frac{\Theta}{2}\right) \tag{A.10}$$

$$\delta_{180} \;=\; \frac{4}{3(n-1)^2} \tag{A.11}$$

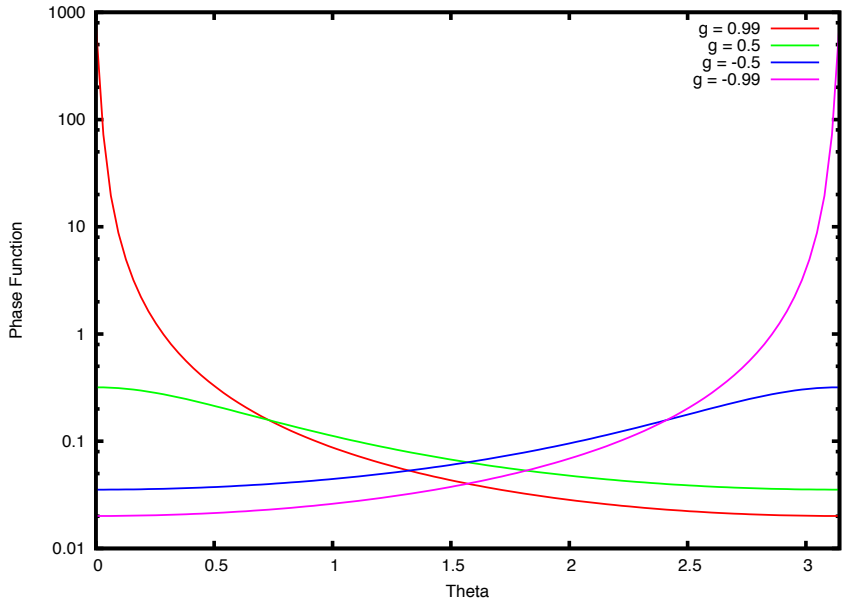


Figure A.1: The Henyey Greenstein phase function for $g = 0.99, 05, -0.5, -0.99$.

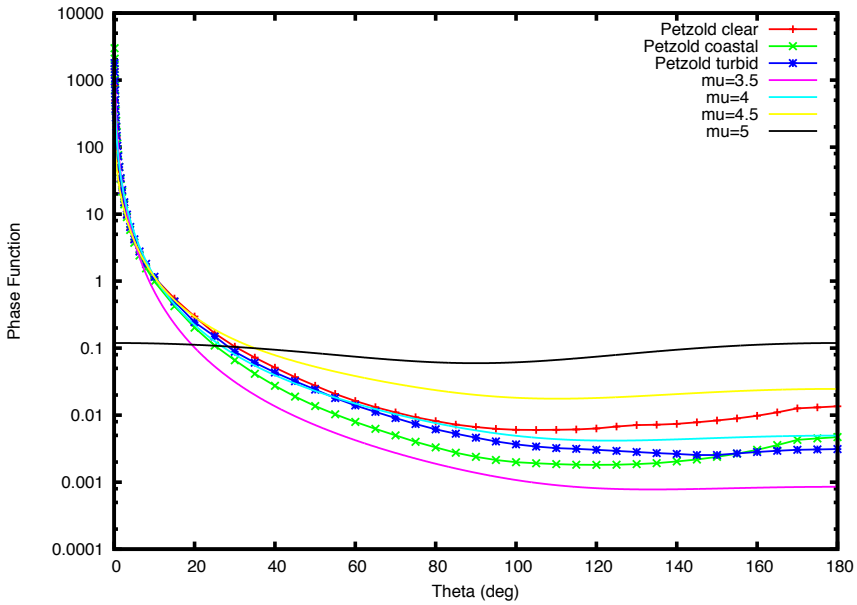


Figure A.2: The Fournier-Forand phase function for $\mu = 0.35, 0.4, 0.45, 0.5$. The parameter n has the value 1.05. Petzold's measured phase functions for clear, coastal, and turbid ocean waters are shown also.

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

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

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

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

A.0.2 Ray Marching

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

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

where

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

and the transmissivity factor T_j is

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

and

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

Note that we can construct these quantities iteratively through the relationships

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

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

with the initial conditions

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

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

which define the ray march process.

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

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

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

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

$$\mathbf{x}_j = \mathbf{x}_{j-1} + \Delta s \mathbf{n}_P \tag{A.23}$$

$$L \mathrel{+}= C^T(\mathbf{x}_j)\, T_j\, \frac{1-\Delta T_j}{\kappa} \tag{A.24}$$

$$T_{j+1} = T_j\, dT_j \tag{A.25}$$

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

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

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

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

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

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

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

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

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

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

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

If the color cannot be assumed to be constant in the interval Δs, then one approach to this is to subdivide the interval further. Here again the random

sampling idea can be brought to bear. Suppose we decide to subdivide into N_s subsegments, within each we can assume that the color and density are constant. The procedure can be as follows

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

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

Bibliography

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

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

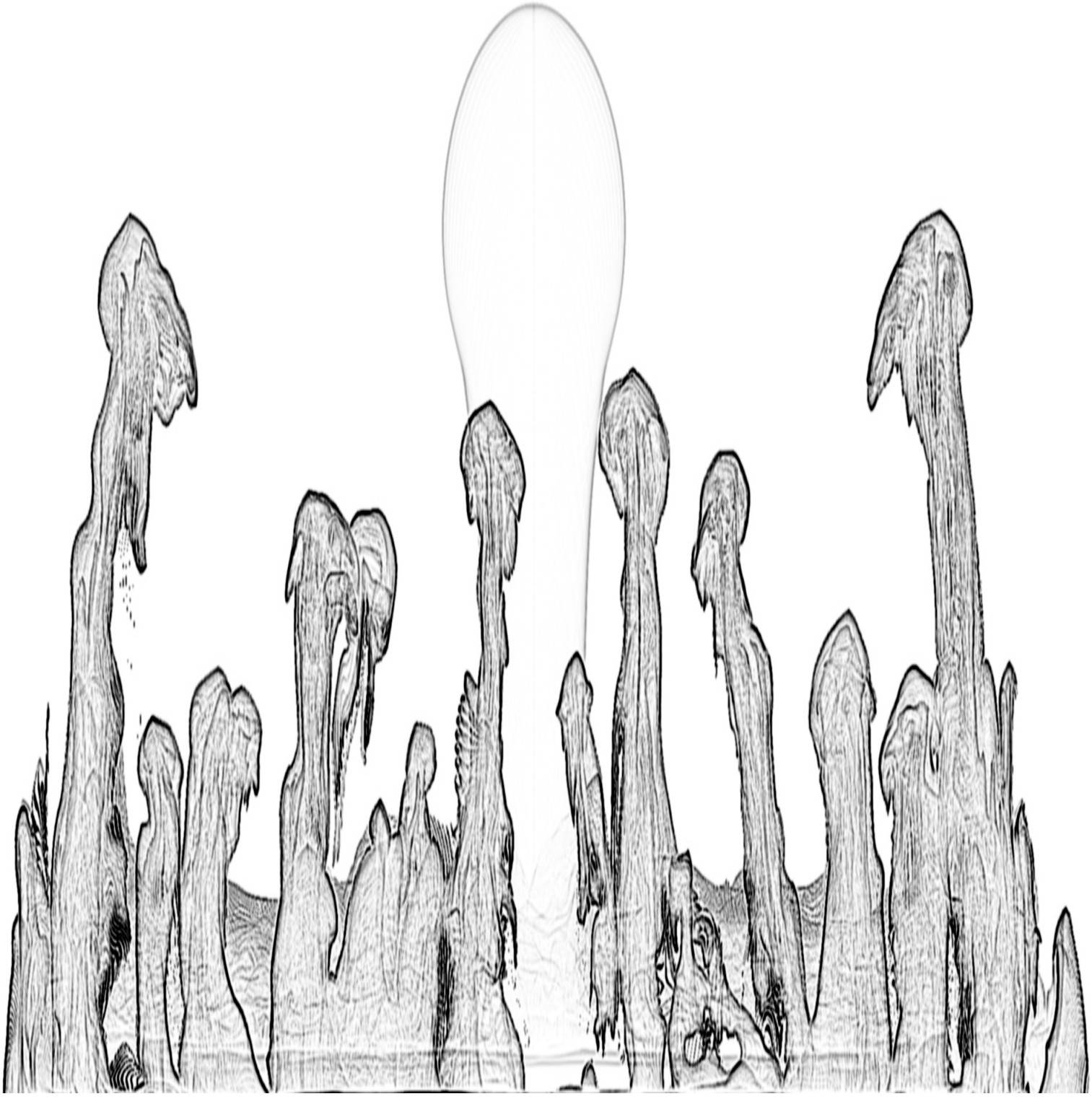
[3] Ken Museth and Michael Clive, *CrackTastic: fast 3D fragmentation in "The Mummy: Tomb of the Dragon Emperor"*, International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH, Los Angeles, California, 2008.

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

[5] Kajiya paper on rendering equation

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

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



Index

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