Gilligan[1] :
A Prototype Framework for Simulating and Rendering Maritime Environments

Jerry Tessendorf
School of Computing, Clemson University
jtessen@clemson.edu

February 24, 2017

[1] Wikipedia [2016]

Contents

List of Figures

1 Introduction

This document describes a prototype environmental scene simulator, called Gilligan, for simulating and rendering scenes containing ocean surfaces, interaction of water surfaces with objects on the water, and clouds. At present the scene simulation does not include an atmospher, terrain, foliage, targets, or emissions from targets. The simulation is intended to render images across an arbitrary number of channels in the hyperspectral range of 12μm – 400nm (longwave IR to UV).

As a prototype environmental scene simulator, Gilligan is assembled from a collection of existing software components and new code. Some of the existing software components are modified in some ways to assist integrating them together. Some existing components are Open Source tools that provide convenient and standardized capabilities.

Simulation of environment components (water surfaces and clouds) can be carried out on the CPU. Water surface simulation can optionally be carried out on the GPU directly.

Hyperspectral rendering of the scene can be accomplished with a raytrace rendering on the CPU, and can optionally carry out Global Illumination-like calculations. Both water surfaces and clouds can be rendered on the GPU, with water surfaces rendered in a tradional OpenGL pipeline for mesh rendering with a GLSL shader; and clouds rendered with a CUDA based ray march volume renderer.

Part I of the document covers ocean surface and interactive water simulation and rendering. Part II covers cloud modeling and rendering. Both Parts also cover the software framework for Gilligan, how simulation, modeling, and rendering processes fit together, and which software components already exist and which are new in Gilligan.

Gilligan is considered a prototype for several reasons:

- It contains a number of existing components that were not written originally for use in an environmental scene simulator, or even with each other.
- It is being assembled as quickly as possible, eliminating rigid and time consuming unit testing and standards for code quality. Instead the assembly is relying on developer experience and limiting goals for the outcome.
- There are limited goals for the performance of Gilligan, which serves primarily as a proof of concept tool.
- There has been no review or implementation of coding or API standards from TrueView or other systems.

At the end of each Part, there are references to additional material of interest, and some appendices containing whole documents that provide more detail on the immediate topics of the the document.

2 Simulation of Random Ocean Wave Surfaces

The statistical properties of wave heights of fully-developed and fetch-limited ocean surfaces have been the subject of a large amount of research and publication. An excellent summary of that work is Massel [2013]. From these statistical properties, random realizations of ocean surfaces can be generated. The choice of using full developed oceans ensures conceptually that the statistics are stationary over time. It also allows the ocean description to consist of a displacement of the wave height from a mean, flat ocean surface. The

random realization generates a simulation of wave height displacements at a rectangular grid of points on the flat ocean.

A key ingredient for these realizations is the power spectrum of spatial correlations of the waves, $P(\mathbf{k})$, for any particular wave vector $\mathbf{k}$. The wave length of the wave associated with the wave vector is $\lambda = 2\pi/k$ where $k = |\mathbf{k}|$, and the direction of travel of the wave is $\mathbf{k}/k$.

2.1 Random Surface Realizations Using Fast Fourier Transforms

The procedure we follow for generating the random realization is described in detail in Deusen et al. [2004], the relevant portion of which is reproduced in Appendix A. It begins with creating the random realization in Fourier space, i.e. equation 42 of Appendix A. This is a two dimensional array of complex valued amplitudes. Appendix A chooses a very simple model for the spatial spectrum. In section 2.3 we list many other models for the spatial spectrum that have have stronger ties to experimental and phenomonological studies than the Appendix A choice. For Gilligan, the user can select from the spectral models list in section 2.3.

The random realization of waves evolve in time according to equation 43 in Appendix A, where t is the time at which the realization is to be generated in real space, and $\omega(k)$ is the dispersion relation for the surface Bernoulli flow, where $k = |\mathbf{k}|$. Section 2.2 presents several options for dispersion relationships. Gilliganuses the combined form because all of the others can be achieved by appropriate choices of parameters in the combined version.

In addition to dispersion, a drift current with velocity $\mathbf{U}$ would modify wave motion by introducing an overall phase on the time evolution, so that the wave amplitude in Fourier space at time t is

$$\tilde{h}(\mathbf{k},t) \;=\; \left\{ \tilde{h}_0(\mathbf{k}) \exp\left[i\omega(k)\; t\right] \;+\; \tilde{h}_0^*(-\mathbf{k}) \exp\left[-i\omega(k)\; t\right] \right\} \; \exp\left(-i\mathbf{k}\cdot\mathbf{U}_D\; t\right) \tag{1}$$

The random surface realization in real space follows from applying a Fast Fourier Transform to the gridded data for the Fourier space amplitude $\tilde{h}(\mathbf{k},t)$. Properly normalized, this is

$$h(\mathbf{x},t) \;=\; \mathrm{FFT}^{-1}\left(\tilde{h}(\mathbf{k},t)\right) \; \frac{2\pi}{L_x}\frac{2\pi}{L_y} \tag{2}$$

where L_x and L_y are the physical dimensions of the rectangular extent of the simulation.

2.2 Dispersion Relationships and Group Velocity

The dispersive nature of surface wave propagation is characterized by a temporal frequency ω that is driven by the wavelength (via the magnitude of the wavevector, k) of the wave and physical properties like botton depth and surface tension. The dispersion relationship also defines two velocities. The phase velocity is defined as $v_p \;=\; \omega(k)/k$, and the group velocity is $v_g \;=\; d\omega(k)/dk$.

Table 1 lists the commonly chosen dispersion relationships and their group velocities. The combined dispersion relation is used in Gilligan.

2.3 Spectral Models

The spatial spectrum $P(\mathbf{k})$ embodies the outcome of numerous experiments and phenomenological modeling of the ocean surface. But there is no single model that is best to use. Researchers have built a variety of models for a variety of oceanographic circumstances. Here we list the models incorporated into Gilligan, but

Deep Water	$\omega^2 = g\ k$	$v_g = 0.5\ (\omega/k)$
Shallow Water	$\omega^2 = g\ k\ \tanh(kh)$	$v_g = 0.5\ (\omega/k)\ \left(1 + \frac{kh\left(1-\tanh^2(kh)\right)}{\tanh(kh)}\right)$
Capillary Waves	$\omega^2 = g\ k\ (1 + (k\ell)^2)$	$v_g = 0.5\ (\omega/k)\ \left(1 + 2\frac{(k\ell)^2}{1+(k\ell)^2}\right)$
Combined	$\omega^2 = g\ k\ \tanh(kh)\ (1 + (k\ell)^2)$	$v_g = 0.5\ (\omega/k)\ \left(1 + \frac{kh\left(1-\tanh^2(kh)\right)}{\tanh(kh)} + 2\frac{(k\ell)^2}{1+(k\ell)^2}\right)$

Table 1: Dispersion and group velocity expressions.

will little discusson of their origin or suitability for particular conditions. More detail about these models is available from Massel [2013].

The spatial spectrum is composed of three factors. The first is a frequency spectrum $S(\omega)$ arising from analysis and modeling of time series data of the wave height of ocean waves, generated from instrumentaton on anchored bouys. The frequency spectrum has no sensitivity to the directional distribution of the waves. The second factor is a directional spectrum $D(\mathbf{k}, \omega)$, providing the fractional amount of waves traveling in the wave vector direction, usually relative to the wind direction in some way. The last factor is a Jacobian of variable transformation that serves to normalize the frequency spectrum in term of wave vectors instead of frequency, using the dispersion relationship.

$$P(\mathbf{k}) \ = \ S(\omega)\ D(\mathbf{k}, \omega)\ \frac{v_g(\omega)}{2k} \tag{3}$$

where $k = |\mathbf{k}|$, ω is the frequency associated with the wavevector $\mathbf{k}$ and the appropriate dispersion relationship and group velocity v_g.

In the remainder of this section a collection of models for $S(\omega)$ and $D(\mathbf{k}, \omega)$ are listed. In general these models were probably not created with a "mix and match" view toward them, but in Gilligan we allow the user the option of selecting any combination of frequency spectrum and directional spectrum for the sake of versatility.

2.3.1 Frequency Spectra

These frequency spectral are contained in section 3.2.3 of Massel [2013].

Pierson-Moskowitz

$$S(\omega) \ = \alpha\ g^2\ \omega^{-5}\ \exp\left[-\frac{5}{4}\left(\frac{\omega}{\omega_p}\right)^4\right] \tag{4}$$

where the parameters are

α	0.0081
ω_p	$0.879\ g/U$
g	gravity ($9.8\ m/sec^2$)
U	Wind speed

JONSWAP

$$S(\omega) = \alpha\ g^2\ \omega^{-5}\ \exp\left[-\frac{5}{4}\left(\frac{\omega}{\omega_p}\right)^4\right]\ \gamma^{\delta} \tag{5}$$

$$\delta = \exp\left[-\frac{(\omega-\omega_p)^2}{2\sigma_0^2\ \omega_p^2}\right] \tag{6}$$

where the new and redefined parameters are

$$\alpha = 0.076\ \left(\frac{gX}{U^2}\right)^{-0.22} \tag{7}$$

$$\omega_p = 7\pi\ \left(\frac{g}{U}\right)\ \left(\frac{gX}{U^2}\right)^{-0.33} \tag{8}$$

$$\gamma = 3.3 \tag{9}$$

$$\sigma_0 = \begin{cases} \sigma_0' & \omega \le \omega_p \\ \sigma_0'' & \omega < \omega_p \end{cases} \tag{10}$$

$$\sigma_0' = 0.07 \tag{11}$$

$$\sigma_0'' = 0.09 \tag{12}$$

and X is the fetch.

Modified JONSWAP

$$S(\omega) = \beta\ g^2\ \omega_p^{-1}\ \omega^{-4}\ \exp\left[-\left(\frac{\omega}{\omega_p}\right)^4\right]\ \gamma^{\delta} \tag{13}$$

$$\delta = \exp\left[-\frac{(\omega-\omega_p)^2}{2\sigma_0^2\ \omega_p^2}\right] \tag{14}$$

where the new and redefined parameters are

$$\beta = 0.006\ \nu^{0.55} \tag{15}$$

$$\omega_p = 7\pi\ \left(\frac{g}{U}\right)\ \left(\frac{gX}{U^2}\right)^{-0.33} \tag{16}$$

$$\gamma = \begin{cases} 6.489 + 6\log\nu & 1.0 \le \nu < 5 \\ 1.7 & 0.83 < \nu < 1 \end{cases} \tag{17}$$

$$\sigma_0 = \begin{cases} \sigma_0' & \omega \le \omega_p \\ \sigma_0'' & \omega < \omega_p \end{cases} \tag{18}$$

$$\nu = \frac{\omega_p\ U}{2\pi\ g} \tag{19}$$

TMA

$$S(\omega) = S_J(\omega)\ r(\omega_*) \tag{20}$$
$$\omega_*^2 = \omega^2\ h\ /\ g \tag{21}$$
$$r(\omega_*) = f^{-2}\left[1\ +\ \frac{2\omega_*^2\ f}{\sinh(2\omega_*^2 f)}\right]^{-1} \tag{22}$$
$$1 = f\ \tanh(\omega_*^2 f) \tag{23}$$

where h is the bottom depth, $S_J(\omega)$ is the JONSWAP frequency spectrum, and f is computed iteratively.

Multipeak

$$S(\omega) = S_c(\omega/\omega_p)\ +\ S_h(\omega/\omega_p) \tag{24}$$
$$S_c(\omega/\omega_p) = A\ \exp\left[-B\ \left(\frac{\omega}{\omega_p} - 1\right)^2\right] \tag{25}$$
$$S_h(\omega/\omega_p) = C\ \left(\frac{\omega}{\omega_p}\right)^{-n}\ \exp\left[-7.987\ \left(\frac{\omega}{\omega_p}\right)^{-m}\right] \tag{26}$$

where the new and redefined parameters are

A	1.835
B	22.2222
C	4.211
n	5
m	8

2.3.2 Directional Spectra

These directional spectral are contained in section 3.4 of Massel [2013].

Pierson

$$D(\mathbf{k},\omega) = \frac{2}{\pi}\ \cos(\theta) \tag{27}$$
$$\cos\theta = \frac{\mathbf{k}\cdot\hat{U}}{k} \tag{28}$$

where $\hat{U}$ is the wind direction unit vector.

Krylov

$$D(\mathbf{k},\omega) = 2^s \frac{\Gamma(2(s+1))}{\Gamma^2(s+1)} (\cos(\theta-\theta_0))^s \quad (29)$$

$$s = 1.8/\omega_* \quad (30)$$

$$\omega_* = \omega \,/\, \overline{\omega} \quad (31)$$

$$(32)$$

where $\overline{\omega}$ is the mean wave frequency, and θ_0 is the wind direction.

Mitsuyasu

$$D(\mathbf{k},\omega) = \frac{2^{2s-1}}{\pi} \frac{\Gamma^2(s+1)}{\Gamma(2s+1)} \left(\cos\left(\frac{\theta-\theta_0}{2}\right)\right)^{2s} \quad (33)$$

$$s/s_p = \begin{cases} \left(\frac{\tilde{\omega}}{\tilde{\omega}_p}\right)^5 & \tilde{\omega} \le \tilde{\omega}_p \\ \left(\frac{\tilde{\omega}}{\tilde{\omega}_p}\right)^{-2.5} & \tilde{\omega} \ge \tilde{\omega}_p \end{cases} \quad (34)$$

$$\tilde{\omega} = \frac{\omega\, U}{g} \quad (35)$$

$$\tilde{\omega}_p = \frac{\omega_p\, U}{g} \quad (36)$$

$$s_p = 11.5\, \tilde{\omega}_p^{-2.5} \quad (37)$$

JONSWAP

$$D(\mathbf{k},\omega) = \frac{2^{2s-1}}{\pi} \frac{\Gamma^2(s+1)}{\Gamma(2s+1)} \left(\cos\left(\frac{\theta-\theta_0}{2}\right)\right)^{2s} \quad (38)$$

$$s/s_p = \left(\frac{\omega}{\omega_p}\right)^{\mu} \quad (39)$$

$$s_p = \begin{cases} 6.97 \pm 0.83 & \omega < \omega_p \\ 9.77 \pm 0.43 & \omega \ge \omega_p \end{cases} \quad (40)$$

$$\mu = \begin{cases} 4.06 \pm 0.22 & \omega < \omega_p \\ -(2.33 \pm 0.06) - (1.45 \pm 0.45)\left(\frac{U}{C} - 1.17\right) & \omega \ge \omega_p \end{cases} \quad (41)$$

$$(42)$$

where C is the phase velocity $C = \omega/k$.

von Mises

$$D(\mathbf{k},\omega) = \frac{1}{2\pi\ I_0(c)} \exp\left[c\ \cos(\theta-\theta_0)\right] \quad (43)$$

$$c = \frac{\log 2}{1-\cos\left[2\cos^{-1}\left(0.5^{0.5/s}\right)\right]} \quad (44)$$

where I_0 is the modifed Bessel function and s comes from the formula 34 for the Mitsuyasu directional spectrum.

Hyperbolic

$$D(\mathbf{k},\omega) = \frac{1}{2}\,\beta\,\cosh^{-2}\left[\beta\,\cos(\theta-\theta_0)\right] \tag{45}$$

$$\beta = \begin{cases} 2.61\,(\omega/\omega_p)^{1.3} & 0.56 < \omega/\omega_p < 0.95 \\ 2.28\,(\omega/\omega_p)^{-1.3} & 0.95 < \omega/\omega_p < 1.6 \\ 1.24 & \text{otherwise} \end{cases} \tag{46}$$

2.4 Horizontal Displacement

The spectral models of section 2.3 provide only for vertical displacements, whereas it is certainly true that horizontal displacements also occur. As described in 4.6 of Appendix A, horisontal displacements follow from the horizontal components of the equation defining the velocity potential. That equation leads to the description of the horizontal displacements directly from the vertical displacement amplitude as

$$\mathbf{D}(\mathbf{x},t) = \mathrm{FFT}^{-1}\left(-i\frac{\mathbf{k}}{k}\,\tilde{h}(\mathbf{k},t)\right)\,\frac{2\pi}{L_x}\frac{2\pi}{L_y} \tag{47}$$

This approach is used in Gilligan, along with an optional adjustable parameter to let the user decide whether to use horizontal displacements, and to artificially scale them if they chose.

2.5 Boundary Conditions

Because the ocean surface simulation uses Fast Fourier Transforms, the displacement data is periodic in each direction with the period being the length of that side of the simulation, i.e. L_x and L_y. This periodicity provides a convenient ability to tile one patch of simulation across an extended region of space, creating an ocean surface much larger than $L_x \times L_y$.

2.6 Multiple Layers of Displacement

An unfortunate artifact of this tiling is that a prominent wave can appear multiple times in a coherent line along the line of site from time to time. Figure 1 shows this artifact in a render from the simulation and rendering tool "RenderWorld".

This artifact can be substantially suppressed through the application of multiple layers of displacement, and through the impact of transmission loss and contrast reduction from the atmosphere. Atmospheric effects are not the subject of this current implementation of Gilligan, but multiple layering is.

Figure 2 shows an image from "RenderWorld" in similar circumstances, but with 3 layers of surface displacement. In this situation there is no artifact clearly visible. In practical usage, tiling artifacts have always been adequately suppressed with 3 or fewer layers. The prescription for how these layers are chosen is discussed in Gundersen [2015]. Gilligan allows for any number of layers to be used.

A consequence of using multiple layers of ocean displacement is that the spatial spectrum of the realization can be altered. Spatial scales that appear in more than one layer contribute duplicate amounts of power to the spectrum, producing artificial peaks in the spectrum. This can be calculated as a multiplicative factor

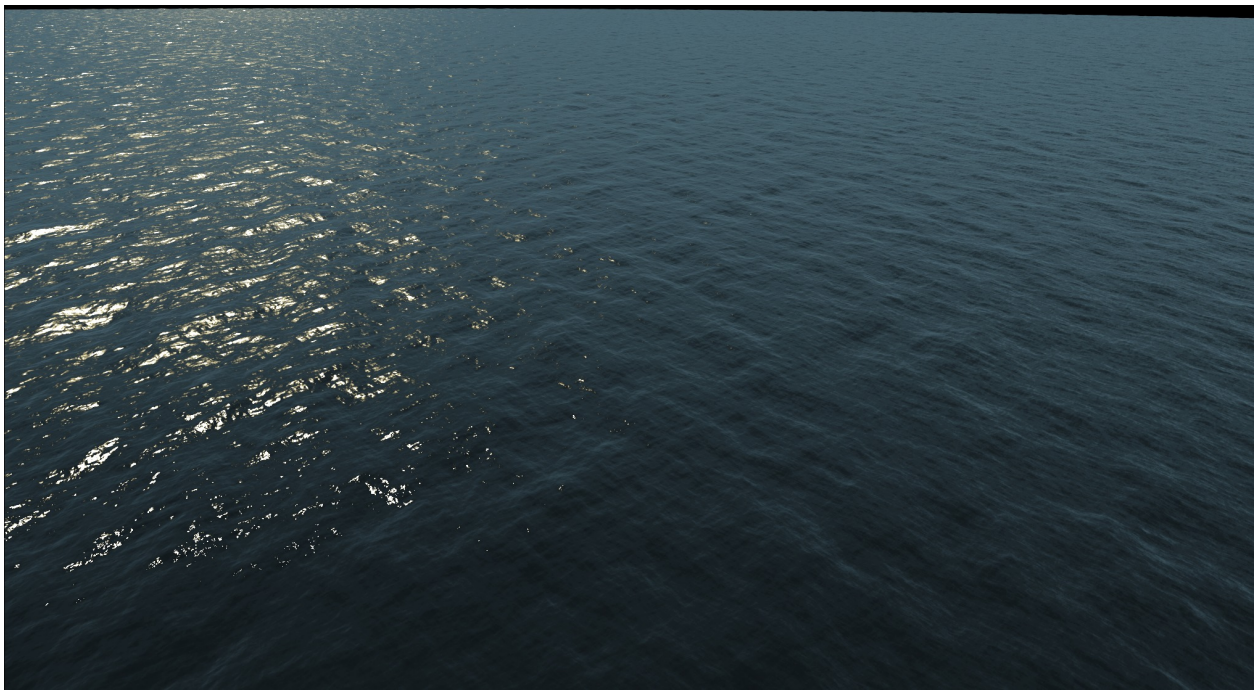


Figure 1: Simulated and rendered frame of ocean water from RenderWorld. The repetition artifact is clear throughout the frame, but particularly in the upper right portion of the image. Only one layer of displacements were used.

Figure 2: Simulated and rendered frame of ocean water from RenderWorld, using three layers of displacement. There is no visible repetition artifact.

on the spectrum. This multiplicative factor can be applied to each layer of simulation data, eliminating it. Gilligan does not apply this correction currently.

To see the impact layering has on the spectrum of the waves, let us represent layer a in terms of a continuous Fourier representation as

$$h_a(\mathbf{x},t) \;=\; \int \frac{d^2k}{(2\pi)^2}\; \tilde{h}_a^C(\mathbf{k},t)\; \exp(i\mathbf{k}\cdot\mathbf{x}) \tag{48}$$

The C superscript means that this is the continuous spectrum representation. When we construct the actual amplitudes as a random representation with Fourier components $\tilde{h}_a(\mathbf{k},t)$, they are constructed within the limited Fourier band driven by the maximum patch size (L_x, L_y) and the cell size $(\Delta x, \Delta y)$for the grid. Consequently the realization has no Fourier components beyond that band. Expressing this band limitation as a form of band filter in Fourier space, the relationship between the continuous amplitudes and the realization amplitudes is

$$\tilde{h}_a^C(\mathbf{k},t) \;=\; \tilde{h}_a(\mathbf{k},t)\; \Pi(k_x, L_x, \Delta x)\; \Pi(k_y, L_y, \Delta y) \tag{49}$$

where Π defines the band limit:

$$\Pi(k, L, \Delta) \;=\; \begin{cases} 0 & k < \pi/L \\ 1 & \pi/L < k < \pi/\Delta \\ 0 & k > \pi/\Delta \end{cases} \tag{50}$$

Suppose N surface realizations have been generated with different patch and cell sizes $(L_x^a, L_y^a, \Delta x^a, \Delta y^a)$, and they are added together. The combined wave height is

$$h(\mathbf{x},t) \;=\; \int \frac{d^2k}{(2\pi)^2}\; \exp(i\mathbf{k}\cdot\mathbf{x})\; \sum_{a=0}^{N-1} \tilde{h}_a(\mathbf{k},t)\; \Pi(k_x, L_x^a, \Delta x^a)\; \Pi(k_y, L_y^a, \Delta y^a) \tag{51}$$

and so the Fourier representation of the combined wave height is

$$\tilde{h}^C(\mathbf{k},t) \;=\; \sum_{a=0}^{N-1} \tilde{h}_a(\mathbf{k},t)\; \Pi(k_x, L_x^a, \Delta x^a)\; \Pi(k_y, L_y^a, \Delta y^a) \tag{52}$$

This height data is a random realization of an ocean surface from multiple sub-realizations. Each realization originates from a common physically-motivated spatial spectrum $P(\mathbf{k})$. Using ensemble averaging of these realizations, the spatial spectrum of the combination is

$$\begin{aligned} P^C(\mathbf{k}) \;&=\; \langle |\tilde{h}^C(\mathbf{k},t)|^2 \rangle && (53) \\ &=\; \sum_{a=0}^{N-1} \langle |\tilde{h}_a(\mathbf{k},t)|^2 \rangle\; \Pi^2(k_x, L_x^a, \Delta x^a)\; \Pi^2(k_y, L_y^a, \Delta y^a) && (54) \\ &=\; P(\mathbf{k})\; \sum_{a=0}^{N-1} \Pi^2(k_x, L_x^a, \Delta x^a)\; \Pi^2(k_y, L_y^a, \Delta y^a) && (55) \end{aligned}$$

The spectrum of the combined waves is altered by the factor

$$B(\mathbf{k}) \;=\; \sum_{a=0}^{N-1} \Pi^2(k_x, L_x^a, \Delta x^a)\; \Pi^2(k_y, L_y^a, \Delta y^a) \tag{56}$$

This function is zero on any scales that none of the sub-realizatons model, 1 on any scales modeled by only one realization, and on any scale modeled by more than one realization it is the square of the number of realizations modeling that scale. This factor induces alterations of the spectrum away from the physical spectrum. But it can be corrected in the simulation by scaling all of the realizations by it, i.e.

$$\tilde{h}_a(\mathbf{k}, t) \;\rightarrow\; \left(\frac{1}{B(\mathbf{k})}\right)^{1/2} \tilde{h}_a(\mathbf{k}, t) \tag{57}$$

This restores the spectrum of the combined waves to the desired physical spectrum. Gilligan includes the option of applying this scaling in WaveSurfer.

3 Simulation of Interacting Wave Surfaces

The random realization approach in section 2.3 propagates linear waves accurately. It is limited in that it does not provide for interaction with objects floating or moving on the surface. In section 5 of "Simulating Ocean Water" (part of Deusen et al. [2004] and reproduced in Appendix A), there is a description of an fast and effective method of handling sources of wave disturbance, and obstruction of waves by objects with arbitrary shape intersecting the surface. The propagation algorithm described there is unstable generally, and so damping has to be applied. The propagation algorithm was replaced with an accurate one in Tessendorf [2014] that is based on FFTs and is very stable. This algorithm, named "eWave", is mathematically identical to that of the spectral model solution, but arranged in a way that supports interaction with objects. The interactive character of eWave also allows it to simulate the interaction of "ambient waves", i.e. ocean surfaces produced from spectral models, with obstructions on the surface, by using the eWave surface to act as the scattered wave component. Gilligan uses eWave.

A strength and a drawback of eWave is that the mechanism for interaction with surface obstructions is very simple. The process amounts to (a) compute a 2D map of the intersection of obstructions, with a value of 1 where there is no intersection and 0 where there is intersection; (b) multiply the simulation data by that map or its compliment at several stages of the dynamic update. The computational cost of step (a) depends on the complexity of the model of the obstruction and the density of simulation grid points, but is typically not extremely expensive. Step (b) is trivial and very fast. Overall, this makes eWave an extremely efficient algorithm for simulation of waves interacting with surface objects. However, an accurate physical model of the surface disturbance produced by an object intersecting the water is much more complex that this two step eWave procedure. An accurate physical model would include a careful evaluation of boundary conditions, momentum conservation, and the flow field around the entire shape of the obstruction, including below the surface. eWave should be viewed as an accurate simulation of linear wave propagation after the obstruction produces a distrubance, with a simplistic model of the disturbance mechanism that could be improved in the future.

Figure 3 is an example of a height field produced by eWave for a model of a sailing ship.

The remaining subsections below briefly describe the algorithms in eWave, and the complete update procedure.

3.1 Linear Bernoulli Equation Dynamics using FFTs

For deep water waves, the linearized equations of motion on the free surface of the water update the behavior of the wave height $h(\mathbf{x}, t)$ and velocity potential $\phi(\mathbf{x}, t)$ at each 2D point $\mathbf{x}$ in the simulation region at each

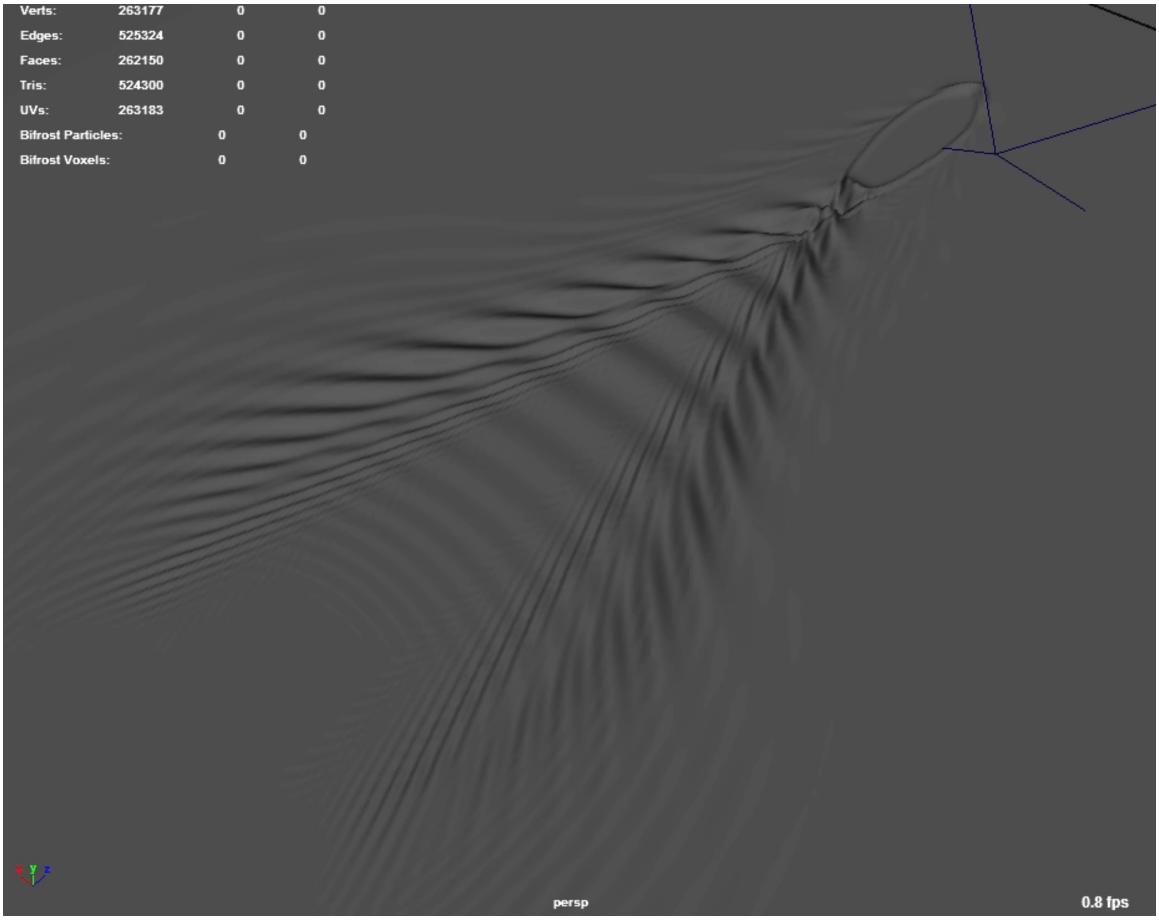


Figure 3: An example eWave calculation of the height field for a boat wake. This is a frame from a Maya playblast showing a mesh displaced by the eWave height simulation.

time t. The linearized equations of motion are

$$\frac{\partial h(\mathbf{x},t)}{\partial t} + \mathbf{U}_D(\mathbf{x},t)\cdot\nabla h(\mathbf{x},t) = \sqrt{-\nabla^2}\phi(\mathbf{x},t) + S_h(\mathbf{x},t) \quad (58)$$

$$\frac{\partial \phi(\mathbf{x},t)}{\partial t} + \mathbf{U}_D(\mathbf{x},t)\cdot\nabla \phi(\mathbf{x},t) = -g\ h(\mathbf{x},t) + S_\phi(\mathbf{x},t) \quad (59)$$

where g is the gravitational constant, $\mathbf{U}_D(\mathbf{x},t)$ is a drift current field that is input to the simulation, and S_h and S_ϕ are input sources of disturbance of the wave height and velocity potential. These equations do not model the physics of how these sources are computed, but allow for the presence of these sources.

The eWave solver for updating from time t to time $t+\Delta t$ is a multistep procedure:

1. Add sources to the dynamic fields:

$$h_1(\mathbf{x}) = h(\mathbf{x},t) + \Delta t\ S_h(\mathbf{x},t) \quad (60)$$

$$\phi_1(\mathbf{x}) = \phi(\mathbf{x},t) + \Delta t\ S_\phi(\mathbf{x},t) \quad (61)$$

2. Fourier transform the dynamic fields

$$\tilde{h}_1(\mathbf{k}) = FFT\left(h_1(\mathbf{x})\right) \quad (62)$$

$$\tilde{\phi}_1(\mathbf{k}) = FFT\left(\phi_1(\mathbf{x})\right) \quad (63)$$

3. Ignoring the drift current for the moment, propagate these fields forward in time in accordance with Tessendorf [2014] (also in Appendix B)

$$\tilde{h}_2(\mathbf{k}) = \cos\left(\omega(k)\Delta t\right)\ \tilde{h}_1(\mathbf{k}) + \frac{k}{\omega(k)}\sin\left(\omega(k)\Delta t\right)\ \tilde{\phi}_1(\mathbf{k}) \quad (64)$$

$$\tilde{\phi}_2(\mathbf{k}) = \cos\left(\omega(k)\Delta t\right)\ \tilde{\phi}_1(\mathbf{k}) - \frac{g}{\omega(k)}\sin\left(\omega(k)\Delta t\right)\ \tilde{h}_1(\mathbf{k}) \quad (65)$$

where $\omega(k)$ is the dispersion relation for deep water.

4. Inverse Fourier transform the dynamic fields

$$h_2(\mathbf{x}) = FFT^{-1}\left(\tilde{h}_2(\mathbf{k})\right) \quad (66)$$

$$\phi_2(\mathbf{x}) = FFT^{-1}\left(\tilde{\phi}_2(\mathbf{k})\right) \quad (67)$$

5. If a drift current exists, apply an advection scheme (eWave uses Semi-Lagrangian, but in the future others could be substituted)

$$h_2(\mathbf{x}) \leftarrow \text{advect}\left(h_2(\mathbf{x}), \mathbf{U}_D(\mathbf{x},t)\right) \quad (68)$$

$$\phi_2(\mathbf{x}) \leftarrow \text{advect}\left(\phi_2(\mathbf{x}), \mathbf{U}_D(\mathbf{x},t)\right) \quad (69)$$

6. Update to time $t+\Delta t$:

$$h(\mathbf{x}, t+\Delta t) = h_2(\mathbf{x}) \quad (70)$$

$$\phi(\mathbf{x}, t+\Delta t) = \phi_2(\mathbf{x}) \quad (71)$$

This dynamic update scheme is sufficiently stable that no damping mechanism is ever needed to maintain the stability. If a friction term is desired for other reasons, the wave height and velocity potential can be modified in step 1 in this fashion:

$$h_1(\mathbf{x}) = h(\mathbf{x},t)\ e^{-\Delta t/\tau} + \tau\ S_h(\mathbf{x},t)\ \left(1 - e^{-\Delta t/\tau}\right) \tag{72}$$

$$\phi_1(\mathbf{x}) = \phi(\mathbf{x},t)\ e^{-\Delta t/\tau} + \tau\ S_\phi(\mathbf{x},t)\ \left(1 - e^{-\Delta t/\tau}\right) \tag{73}$$

where τ is the relaxation time of the friction.

3.2 Generalized Dispersion

In circumstances more general than deep water, the dispersion relationship changes as described in section 2.2. In these situations, equations 64 and 65 continue to hold, but with a more general dispersion relationship.

3.3 Interaction with Objects Intersecting the Surface

eWave computes an interaction of the water surface with objects that intersect the water surface. These obstructions interact with the water surface in three ways: (1) the obstruction prevents waves from propagating to the interior of the objection, producing a corresponding scattered wave; (2) the obstruction can be a source of displacement; and (3) the obstruction can act as a barrier to "ambient" ocean waves, producing the corresponding scattered waves. The third item is the subject of the next subsection.

All interactions of objstructions with the water surface on controlled with a 2D map, called an obstruction map, and has values ranging from 0 to 1. A value of 0 at a location of the map means that the obstruction is present at that location and has full effect. A value of 1 means that the obstruction is not located at that position and has no effect. A value between 0 and 1 means that the obstruction is close to that location ("close" usually meaning within a grid point), and so a partial effect is allowed for. This obstruction map is designated $O(\mathbf{x})$.

The first interaction mechanism updates the wave surface by multiplying the wave height by the obstruction map:

$$h_O(\mathbf{x}) = O(\mathbf{x})\ h(\mathbf{x}) \tag{74}$$

This operation causes the wave height to satisfy the most basic boundary condition for obstructions, that there are no waves at the location of the obstruction. The operation alone is sufficient to produce scattered waves that form ripples and boat wakes, including Kelvin Wakes, simply by translating the location of the obstruction during a sequence of frames.

The second effect uses the compliment of the obstruction map as a source of wave height disturbance, i.e. as the S_h term in equation 60, in the form

$$S_h(\mathbf{x}) = S_0\ (1 - O(\mathbf{x})) \tag{75}$$

with the scaling factor S_0 chosen by the user. Figure 4 shows a breakdown of these elements of simulation in eWave.

3.4 Interaction with Ambient Waves

Ambient waves come into eWave simulation as a height map $h_A(\mathbf{x},t)$ that has been created by other means. For example, the output of wavesurfer spectral models qualifies. The most obvious aspect of ambient waves

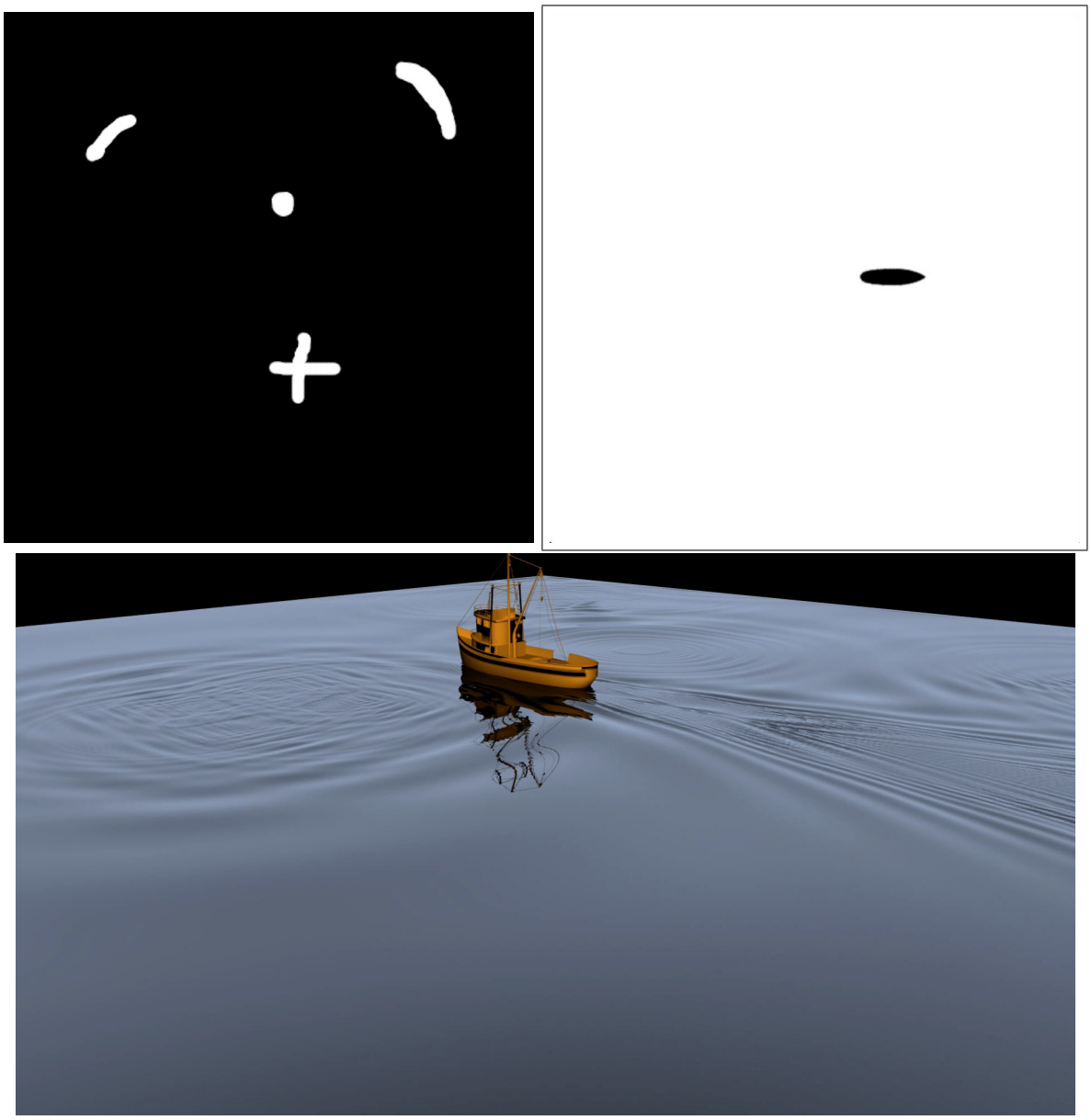


Figure 4: Breakdown of an eWave simulation. A boat travels at constant speed across the surface, while artificial wave height sources exist around it. Top left: Source map constructed S_h in Gimp. The source is on continuously throughout the simulation. Top right: The obstruction map from the boat at one frame. As the boat moves, the black oval region moves from left to right in the map from frame to frame. The obstruction map is also a source as described in equation 75, and added to the source map from the top left. Bottom: rendering of the eWave surface and boat.

is that they exist in the region of the obstruction, i.e. where $O(\mathbf{x}) = 0$. Those waves are the ones that should be reflecting from the obstruction.

The ambient waves located only inside the obstruction could be suppressed by adding

$$-h_A(\mathbf{x}, t)\ (1 - O(\mathbf{x})) \tag{76}$$

to the ambient waves. Alternatively, we can add this displacement to the eWave displacement, so that it acts like a special source of displacement which propagates out of the obstruction. Hence the ambient wave interaction is to update the eWave height to

$$h(\mathbf{x}, t)\ -=\ h_A(\mathbf{x}, t)\ (1 - O(\mathbf{x})) \tag{77}$$

As with the interaction with obstructions, this method of interacting with ambient waves is only a very limited physical model that omits issues of momentum conservation and mass conservation. However, it captures the lowest level physical requirement that waves do not exist inside the obstruction, it is a very fast computation, and it captures accurate propagation of the interaction away from the obstruction.

3.5 Horizontal Displacements

The mathematical pedigree of eWave is the same as spectral model-based ocean surfaces. As such, the procedure described in section 2.4 is completely applicable here, without alteration.

3.6 Boundary Conditions

The propagation update procedure in section 3.1 uses FFTs on the wave height and velocity potential. Consequently, these quantities are spatially periodic at their boundaries. It would be unfortunate for eWave to compute a transient wave disturbance that propagates to other regions by virtue of periodicity. Boundary conditions should be enforced that suppress propagation of waves past the boundaries.

A trim operation that uses linear tapering accomplishes this. The trim $T(\mathbf{x})$ is applied to the dynamical variables as

$$h(\mathbf{x}) \ \leftarrow\ T(\mathbf{x})\ h(\mathbf{x}) \tag{78}$$

$$\phi(\mathbf{x}) \ \leftarrow\ T(\mathbf{x})\ \phi(\mathbf{x}) \tag{79}$$

The trim is a product of two 1D trim operations

$$T(\mathbf{x})\ =\ T_x(x)\ T_y(y) \tag{80}$$

Each of the 1D trims is a linear taper at locations close to the edges of the simulation:

$$T_x(x)\ =\ \begin{cases} x/L & x/L < 1 \\ |x - L_x|/L & |x - L_x|/L < 1 \\ 1 & \text{otherwise} \end{cases} \tag{81}$$

where the edge in the x direction is located at $x = 0$ and $x = L_x$, and L is the size of the linear taper region. In practice, this trim procedure suppresses propagation across the periodic boundary, but it also frequently

induces reflected waves, because the trim acts roughly like a very "soft" obstruction map. However, a modification of the 1D trim to include a power low fall off as

$$T_x(x) = \begin{cases} (x/L)^\alpha & x/L < 1 \\ (|x - L_x|/L)^\alpha & |x - L_x|/L < 1 \\ 1 & \text{otherwise} \end{cases} \tag{82}$$

has shown that a very small value of α, e.g. $\alpha = 0.05$, both suppresses periodic propagation and reflections. This is the method of choice for eWave boundary conditions.

3.7 eWave Update Scheme

Sections 3.1 through 3.6 are the collection of operations in an eWave simulation. The order in which those steps are taken in a single update in time is:

1. Apply obstruction (section 3.3)
2. Apply ambient waves (section 3.4)
3. Apply trim (section 3.6)
4. Apply propagation (section 3.1)
5. Apply horizontal displacements (section 3.5)
6. Apply obstruction (section 3.3)
7. Apply trim (section 3.6)

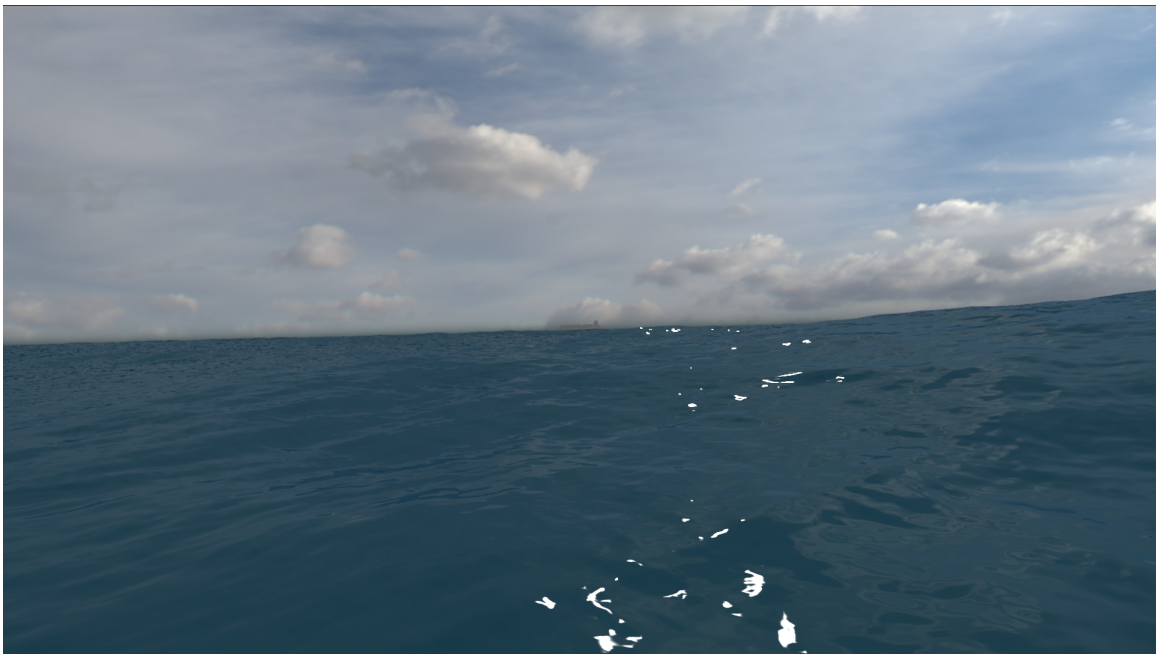
4 Wave Simulation Platforms

Ocean surface and interactive water waves can be efficiently simulated on both CPU and GPU platforms. Even on CPUs, a steady frame-by-frame update process can run in excess of 40 fps for a reasonably sized grid (e.g. 512×512) and a quad-core processor. Multithreaded processing is most effective in the FFT computation, and most OSS and commercial FFT packages support it. The CPU-based C++ code we are using for ocean surfaces, called WaveSurfer, runs at these speeds. Similarly, the interactive water C++ code we are using, called Ewave, also is dominated by FFT calculations, and so runs around 40 fps on CPUs for reasonable grids.

For GPU-based simulation, CUDA codes will be written that implements the algorithms in WaveSurfer and Ewave. This will use the FFT routines in the standard library that accompanies CUDA.

5 Rendering Wave Surfaces

Rendering an environment containing water surfaces and other elements brings together a variety of software components and "glue" software that has them communicate with each other. A demonstration of glued rendering is in figure 5 It uses several of the tools brought into Gilligan, but without a careful integration and optimization of their code and functionality. The collection of tools and the process for each frame in this demonstration is as follows:



(a)

(b)

Figure 5: Demonstration of a rendered complex water water environment, showing (a) ocean surface over long distances, atmospheric and water volume effects effects; (b) dynamic water surface motion related to motion of other scene elements.

Ocean Waves The utility WaveSurfer was used to create two layers of water surface: one representing wind waves, and one representing long swells. Table 2 shows execution time for a frame of ocean surface from WaveSurfer, as a function of the grid size. The wind wave grid was 512×512, and the swell grid was 2048×2048.

Interactive Waves Interactive waves are produced when the shark fin and tail appear above the water surface, and their intersection with the surface produce a wake using eWave. The input to eWave that describes the height disturbance "source" is a map of the same size and resolution as the simulation that describes where the source is present - essentiall values of 0 everywhere that the fin and tail do not intersect the surface, and 1 where they do. This map is generated in Ash as the image in an orthographic camera when the ocean surface and shark model are rendered with appropriate surface shaders. Table 2 shows execution time for a frame of interactive waves, as a function of grid size. The grid for this example was 1400×600 with a resolution of 1 cm.

Water surface mesh The mesh uses level-of-detail meshing methods (Polack and LaMothe [2003]) to generate a grid that is high resolution near the camera and iteratively lower resolution away from the camera. The surface mesh uses all three water surface simulations (two oceans and one interactive). In figure 5 the mesh triangles have approximately 3 cm edges near the camera, and approximately 31 m edges 8 km from the camera, for a total of just over 1 million triangles.

Sky and ocean floor The Sky is a HDR photograph, shown in figure 6 mapped onto an environment sphere. The ocean floor is a flat plane with a fixed color. Figure 7 is the scene with the ocean surface removed, revealing the skymap, bottom plane, shark model, and ship model.

Atmosphere and water volume Because the sky is an HDR photo, the impact of atmospheric volumetric scattering and attenuation is "baked in" to the photo. However the scene also has a thin haze layer in the atmosphere that is not part of the HDR photo. This atmospheric haze geometrically is a hydrostatic profile of haze density, with a lapse rate of 50m. The shading of the haze consists of simple spectral attenuation, and a constract color. The shader computes the analytically exact spectral transmission between any two points. The constrast color is applied as the color times the spectral opacity between the two points of interest. Under the water surface, the same shader was used to create a water volume scattering and attenuation. The lapse rate underwater was set to 10000 m to effectively produce a constant density field, and the spectral attenuation and constract parameters were set based on Figures 1 and 9 of Stentz [1975]. Figure 7 shows the impact of both atmosphere and underwater shaders for attenuating light and adding contrast.

Ship and shark The ship and the shark are models from OBJ files. The ship includes texture maps, but the shark is shaded as a solid color.

Rendering Shading of the water surface mesh included several contributions: reflections generated a ray that was added to the stack of rays being rendered, as did refractions. From the intersection point, a glitter contribution to the light was computed based on a "roughness" parameter based partly on the unresolved wave glitter model in Tse et al. [1990]. The atmospheric attentuation and contrast shader was also used. The scene was rendered with the Ash ray-trace renderer. All of the scene geometry, shaders, light and camera were assembled via a python script. Python bindings exist for Ash, WaveSurfer, and eWave, and the python script handles simulation control, scene assembly, render control, and image output. The python script assembles each frame independent of any other frame, including running eWave simulations, so that a render farm could be used. The ray tracer cast 40 rays

Figure 6: High dynamic range photo of a sky used to render figure 5.

per pixel to suppress alias artifacts, and each ray cast spawned reflected/refracted rays up to 20 times, although in practive the actual number of spawning events was typically below 5 per cast. The ray tracer is multithreaded using OpenMP, and ran with 8 threads. Total execution time for a single frame - wave surface generation, interactive wave simulation, mesh assembling, loading all geometry into Ash, rendering, and image output - was approximately 5.5 minutes for each frame. However, this process is far from optimized, and there is considerable room for speed up via the implementation in Gilligan.

This particular process does not represent the extent of capabilities of Gilligan, and is not a comprehensive collection of the pre-Gilligan capabilities. This is just a snapshot of the process needed to create and render this particular scene.

Regardless of the method of rendering a wave surface, the surface has to be assembled from a collection of surface simulations. The collection consists of multiple layers of ocean surface waves with a range of spatial scales, and zero or more eWave simulations of transient wave motion. The process for merging the assembly into a unified surface is described in sectio 5.1.

The geometry of the unified surface can be represented in two ways within rendering engines. The highest-quality, lowest-memory-footprint method is as a displacement map attached to simple geometry of the mean ocean surface. A case study for creating unified water surfaces in the context of a renderman displacement shader is contained in Gundersen [2015].

OpenGL rendering does not support displacement maps. Vertex shaders can handle some of the functionality of displacement maps, but on-the-fly generation of micro-polygons is not yet easily accomplished. Because of this, Gilligan supports only the second geometric representation, i.e. a displaced mesh of polygons, as described in section 5.2.

The interaction of the light with the wave surface is modeled in Gilligan with three major components

Grid Size	Average FPS
WaveSurfer	
128×128	833
256×256	215
512×512	59
1024×1024	14
eWave	
128×128	454
256×256	59
512×512	14
1024×1024	2

Table 2: CPU simulation rates for various grid sizes, for one thread. Machine: Intel Core i5-2500 1.6 GHz. Average FPS computed from the bash shell time command real usage for a routine that runs the surface update 1000 times.

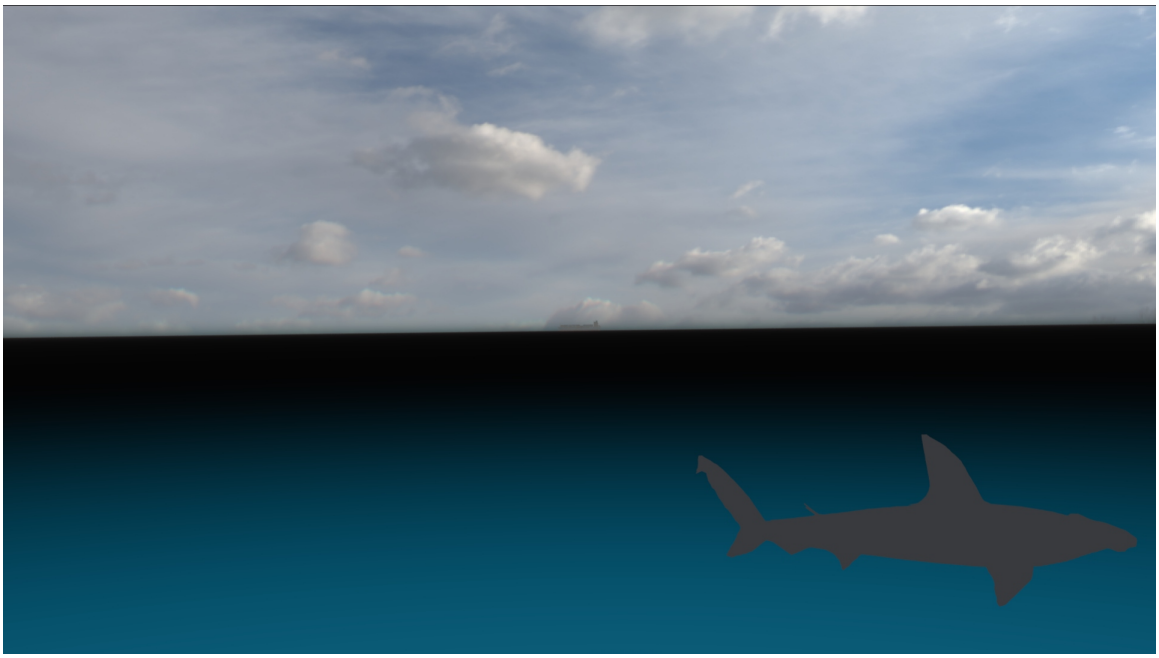


Figure 7: The scene rendered without the water surface, revealing the skymap, bottom plane, ship and shark models.

1. Specular reflection and refraction with Fresnel coefficients.
2. A microscopic model of reflection of unresolved waves. This can be significant when alignment of camera, surface, and light source induce a lot of glitter in the image.
3. Blackbody emission based on the surface temperature and Fresnel transmissivity. This component is extremely small in visible and UV bands, but important in short- and mid-wave IR, and dominant in long-wave IR.

Each of these is described in the sections below, and an single comprehensive shading model containing all of these presented. For iterative ray-trace renders in MaryAnn, the shading model includes the needed hooks for handling multiple reflections/refractions.

5.1 Merging Multiple Surfaces

Both of the water simulation methods employed in Gilligan, FFT spectal models and eWave, are based on the linearized Bernoulli equation for free surface motion. Within this framework, two solutions can be added to produce another valid solution. Indeed, this is what makes it possible to use FFTs in the computation. The concept of addition here means direct numerical addition of the wave height of two different simulations at the same location, i.e. for two wave height simulations $h_1(\mathbf{x}, t)$ and $h_2(\mathbf{x}, t)$, the addition

$$h_{add}(\mathbf{x}, t) \;=\; h_1(\mathbf{x}, t) \;+\; h_2(\mathbf{x}, t) \tag{83}$$

is also a valid simulation. Similarly, if horizontal displacements are computed, they also can be directly added together:

$$\mathbf{D}_{add}(\mathbf{x}, t) \;=\; \mathbf{D}_1(\mathbf{x}, t) \;+\; \mathbf{D}_2(\mathbf{x}, t) \tag{84}$$

This additive behavior extends to any number of simulated wave surfaces, and allows mixing of eWave and ocean FFT simulations.

This general behavior is implemented in Gilligan via a base C++ class called `WaveSurface`, which provides access to underlying simulation data via a standard interface. For a point $\mathbf{x} = (x, y)$ on the undisplaced surface, the `WaveSurface` class provides virtual methods such as `WaveHeight(x,y)` for the wave height, `WaveCuspDisplacementX(x,y)` and `WaveCuspDisplacementY(x,y)` for the components of $\mathbf{D}(\mathbf{x})$, and similar access to other wave surface properties. A derived class called `HdNWaves` stores any number of `WaveSurface` instantiations, and implements the virtual methods to carry out the needed additions to fulfill operations like those in equations 83 and 84. This is a powerful mechanism for building complex wave surfaces from components that are individually simulated for specific purposes.

5.2 Wave Surface as a Displaced Mesh

The geometry for the ocean surface begins with an undisplacement mesh, which consists of a collection of points in space, $P_i, i = 0, \ldots, N-1$, each of which has three pieces of information: (1) a position in space $\mathbf{x}_i$, a normal to the surface $\hat{\mathbf{n}}_i$, and a list of edges connection point i to other points in the collection, $e_i^j, j = 0, \ldots M_i - 1$. This is summarized by defining a mesh M as

$$M \;=\; \left\{ P_i \mid P_i \equiv \left(\mathbf{x}_i, \hat{\mathbf{n}}_i, \{e_i^j\} \right),\; i = 0, \ldots N-1 \right\} \tag{85}$$

Typically, the undisplaced mesh is chosen as a rectangular grid of flat points connected as triangular faces. In this situation it is convenient to use an index with two components, replacing the index i in equation 85

with the pair ij, each index running in orthogonal directions. The undisplaced mesh has the form (assuming the upward direction is the z component)

$$\begin{aligned} \mathbf{x}_{ij} &= (x_0 + i\Delta x,\ y_0 + j\Delta y,\ 0) && (86)\\ \hat{\mathbf{n}}_{ij} &= (0,\ 0,\ 1) && (87)\\ \left\{e^k_{ij}\right\} &= \{\ i-1j,\ ij-1,\ i+1j,\ ij+1,\ i-1j-1,\ i+1j+1\ \} && (88) \end{aligned}$$

where (x_0, y_0) is an origin for the location of the mesh, and $\Delta x \times \Delta y$ is the dimensions of a cell of the grid.

Displacement of this mesh modifies the point location and normal, but does not alter the connectivity of the grid

$$\begin{aligned} \mathbf{x}_{ij} &\leftarrow \mathbf{x}_{ij} + \hat{\mathbf{z}}\ h(\mathbf{x}_{ij}, t) && (89)\\ \hat{\mathbf{n}}_{ij} &\leftarrow (\hat{\mathbf{z}} - \vec{\epsilon}(\mathbf{x}_{ij}, t)) \ /\ \left(1 + |\vec{\epsilon}(\mathbf{x}_{ij}, t)|^2\right)^{1/2} && (90)\\ & && (91) \end{aligned}$$

with $\vec{\epsilon}$ being the slope of the displaced surface:

$$\vec{\epsilon}(\mathbf{x}, t) = \nabla h(\mathbf{x}, t) \quad (92)$$

If horizontal displacements are included, this displacement becomes

$$\begin{aligned} \mathbf{x}_{ij} &\leftarrow \mathbf{x}_{ij} + \hat{\mathbf{z}}\ h(\mathbf{x}_{ij}, t) + \mathbf{D}(\mathbf{x}_{ij}, t) && (93)\\ \hat{\mathbf{n}}_{ij} &\leftarrow \frac{\left((1,0,0) + \hat{\mathbf{z}}\ \epsilon_x + \frac{\partial \mathbf{D}}{\partial x}\right) \times \left((0,1,0) + \hat{\mathbf{z}}\ \epsilon_y + \frac{\partial \mathbf{D}}{\partial y}\right)}{\left|\left((1,0,0) + \hat{\mathbf{z}}\ \epsilon_x + \frac{\partial \mathbf{D}}{\partial x}\right) \times \left((0,1,0) + \hat{\mathbf{z}}\ \epsilon_y + \frac{\partial \mathbf{D}}{\partial y}\right)\right|} && (94) \end{aligned}$$

5.3 Material Shading

Both OpenGL rendering and CPU-based ray trace renderers separate the rendering operations into a multi-step process. One of the key steps is Shading, which is the process of computing the amount of light leaving a surface, based on the local surface structure, material properties that drive a BSDF (Bidirectional Scattering Distribution Function) model, and the distribution of light incident on the surface from all light sources in the scene. In ray trace and Global Illumination renderers, the shading process can also drive the generation of additional rays representing reflected and transmitted light that interact with other elements of the scene.

For water surfaces, the BSDF model is specular reflection and transmission. The Fresnel reflectivity and transmissivity are driven by the index of refraction as described in section 5.3.1.

Depending on the camera properties, viewing conditions, and wave simulation conditions, the camera may be unable to resolve all of the detail of the waves in a pixel. This could lead to temporal and spatial aliasing, which can be suppressed by increasing the quality of the sampling, at the cost of larger computational resources. An alternative to increasing the computational effort is to modify the BSDF from being purley specular, to one that averages the specular behavior over the distribution of waves that are unresolved. This is very similar in concept to the Torrance-Sparrow model of BRDFs of rough surfaces. Numerous oceanographic researchers have looked at the radiometric impact of unresolved waves, and a BRDF model has been generated. More detail on this topic is in section 5.3.2.

For infrared wavelengths, blackbody emission becomes important and even dominant in some situations. The model for this is discussed in section 5.3.4.

Finally all of this is put together into a unified shading model in section 5.3.7.

5.3.1 Fresnel Reflection and Transmission

Specular reflection and transmission directions are described in section 6.1 of Appendix A. The reflectivity and transmissivity for unpolarized light is equation 62 in Appendix A. These quantities depend on the ratios of the indices of refraction of air and water. For Gilligan, we assume that the index of refraction of air is 1. For water, the index of refraction (IOR) is wavelength dependent. An excellent source of a model of the IOR for water that covers are broad spectral range in UV to IR bands is IAPWS [1997]. This is the model of water IOR that will be in Gilligan.

5.3.2 Unresolved Waves and Glitter

Unresolved surface waves contribute to the reflected radiance through a mechanism similar to that in the Torrance-Sparrow model of the BRDF of a rough surface. This similarity was formalized into a specfic model of reflectivity by Tse et al. [1990] (their equation 10) that is suitable for the Gilligan shading system. The reflected glitter light is

$$L_{glitter} \;=\; I_{inc}\,\frac{r_{flat}}{4\,\cos\theta_n^4\,\cos\theta_v}\,P_U(\vec{\epsilon}) \tag{95}$$

where I_{inc} is the incident irradiance from sunlight, r_{flat} is the Fresnel reflectivity of the flat water surface, θ_n is the refracted angle with respect to the flat surface normal, θ_v is the viewing angle with respect to the flat surface normal, and $P_U(\vec{\epsilon})$ is the probability density of unresolved surface slopes. As in Tse et al. [1990], Gilligan uses a gaussian probability distribution for slope, with a variance derived from the spectral model for the ocean surface and the spatial scale of the limits of resolution. Assuming the wave height spectrum is $P(\mathbf{k})$, the slope variance for any scale of resolution is

$$\sigma_U^2(R) \;=\; \frac{1}{N_P}\int_{k_{max}}^{\infty} dk\;k\;(-k^2)\;S(\omega)\;\int_0^{\pi} d\theta\;D(\theta,k) \tag{96}$$

where the normalization factor is

$$N_P \;=\; \int_0^{\infty} dk\;k\;S(\omega)\;\int_0^{\pi} d\theta\;D(\theta,k) \tag{97}$$

and $k_{max} = \pi/R$ is the wave number for smallest resolveable scale of the camera R, i.e. the resolution of an individual pixel.

5.3.3 Whitecaps

Gilligan generates a whitecap map, which describes where whitecaps should exist on the ocean surface due to peaking of wave heights, and how old the whitecap is post-generation. The map is not a suitable "texture" of whitecap foamy detail, but acts as a guide for where to put whitecap texture in the scene. The whitecap map can also act as a mixing parameter for the decay of the foamy texture.

The value at each pixel of the map is a number between 0 and 1. The value 0 means that no whitecap is present at that location, and the pixel value is set to 1 when that location on the ocean surface has whitecap generation (described below). Over time, if no more whitecaps are generated at a pixel, the value decays exponentially in time, with a half-life that is input from the user to provide for the physics of foam decay (see Tse et al. [1990]).

Generate of whitecap is detected from the following process: The ocean surface data includes horizontal displacements, and also gradients of the horizontal displacement. These gradients in turn produce two

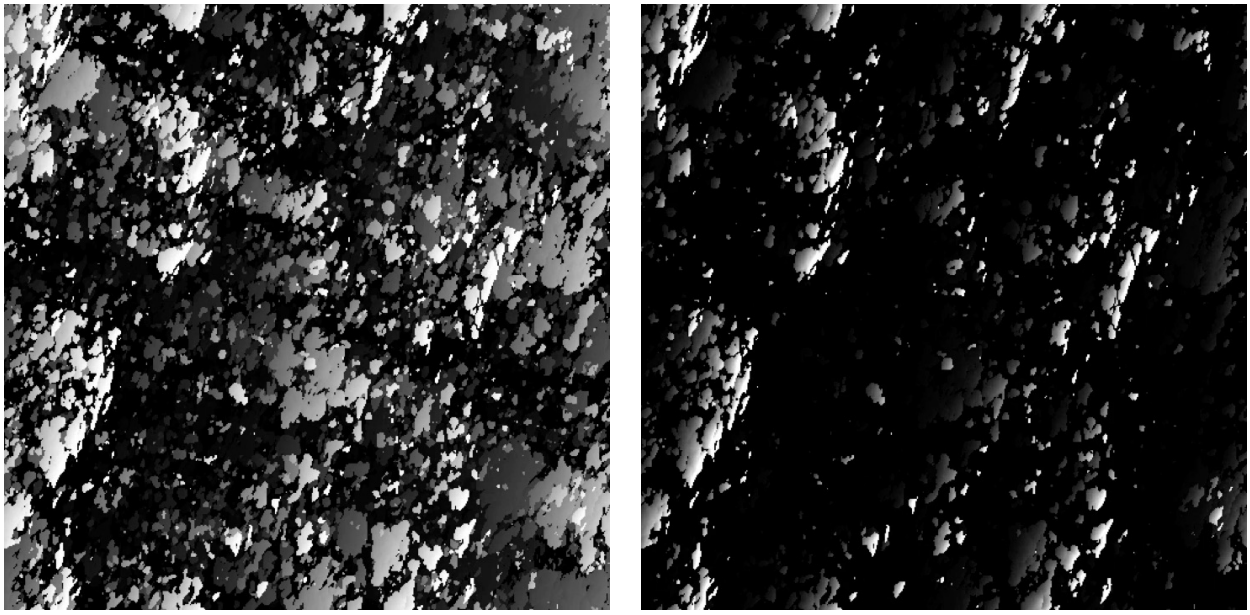


Figure 8: Two examples of whitecap maps. The maps cover an approximately $500m \times 500m$ area. Left: Decay time of 8 seconds. Right: Decay time of 4 seconds.

eigenvalues that are dimensionless measures of the area compression of the horizontal displacement process (see appendix A, section 4.6 for details on eigenvalue computation). When the minimum of the two eigenvalues approaches 0, the peak of the wave is very sharp, and physically is the location where whitecaps are generated. To update the whitecap map, a threshold value is provided by the user. At every location at which the minimum eigenvalue falls below that threshold, the whitecap map is updated to the value 1. If that pixel is not updated to 1, then its value is reduced by a factor of $\exp(-\Delta t/T)$ for the exponential time decay, where Δt is the time step in the simulation, and T is the whitecap half-life.

Figure 8 illustrates the data generated in whitecap maps. Area coverage of whitecaps depends on both the decay time and the eigenvalue threshold.

5.3.4 Blackbody Emission

In Gilligan, blackbody emission assumes that the emissivity is the same as the Fresnel transmissivity, and that there is a fixed temperature T for the ocean surface water that does not vary from place to place. The emitted radiance is

$$L_{blackbody} \;=\; (1 - r_F)\; B_\lambda(T) \tag{98}$$

where r_F is the Fresnel reflectivity and $B_\lambda(T)$ is the Blackbody radiance for the particular wavelength λ and temperature T.

5.3.5 Skylight

In its most basic form, the contribution of skylight comes from specular reflection of the color of the sky

$$L_{skylight} \;=\; r_F \; L_{sky} \tag{99}$$

where r_F is Fresnel reflectivity. Because Gilligan does not handle atmospheric models, L_{sky} will be an input value.

5.3.6 Upwelling

In its most basic form, the contribution of upwelling light from below the water surface comes from specular transmission of the color upward

$$L_{upwelling} \;=\; t_F^{\uparrow} \; L_{ocean} \tag{100}$$

where $t_F^{\uparrow}$ is the Fresnel transmissivity for upward transmission of light into the direction of the incoming ray (view direction), and L_{ocean} is the color of light propagating upward from the ocean volume just below the surface. Gilligan does not have a mechanism for simulating L_{ocean}, and instead uses an input value.

5.3.7 Unified Shading Model

Here all of the components of the surface shading are combined into a single model. At its basic level, the unified shading model can be written as

$$L_{water\ surface} \;=\; L_{blackbody} \;+\; L_{glitter} \;+\; L_{sky} \;+\; L_{upwelling} \tag{101}$$

As described so far, this is a suitable shading model for an OpenGL render because it requires only local properties of the wave surface, a small set of input parameters, and some calculations based on that information. The GPU renderer Ginger will use this shading model.

In the case of the ray trace renderer MaryAnn, the shading system is capable of launching reflected and refracted rays and track the impact of more subtle effects like multiple reflections/refractions through waves, underwater caustics, and structured water volumes and skies. However the shader will not implement all of those capabilities. The shader will implement multiple reflections/refractions through the surface, but retain input values for the sky and upwelling ocean color.

6 Simulation of Volumetric Clouds

The simulation of a cloud that is both "fully formed" and evolving involves several steps of modeling the cloud structure, including the spatial "noise", then evolving over time several key parameters in the model. Interleaved with the steps are options for storing the results in efficient sparse grids, or keeping the results procedural for additional steps. Prior to rendering most of the operations should be committed to grid(s) for the sake of render speed, but some aspects can optionally be left procedural during render in order to capture detail with moderate cost in render time.

Gilligan's system for modeling and rendering clouds is based on a procedural framework for creating and manipulating volume data. It is capable of describing many kinds of volumetric data beyond clouds. The procedural approach creates opportunities for altering the data processing workflow to generate variations,efficiencies, or research on new concepts. This section describes modeling, animation, simulation, rendering, and data representation techniques that are all implemented in this procedural system.

6.1 Sparse Grids for Clouds

For a variety of uses, volumetric data will need to be stored as values in 3D rectangular grids of voxels. If the memory for the grid were naively allocated in an initialization phase, the size of the grid would be substantially limited by the available memory on the CPU and/or GPU. For example, if only a grid for density were allocated, with dimensions $N_x \times N_y \times N_z$, the required amount of memory would be

$$\text{sizeof(float)} \times N_x \times N_y \times N_z. \tag{102}$$

Assuming 32GB were available for this one grid, and that the grid is the same size in each direction, then the maximum grid size is $N_x = N_y = N_z = 2048$. If this cloud is to cover a region of 1 km^3, then it could only be resovled to as small as 0.5 meters. This is very limiting on the size of the volume, and very demanding on the size of the memory available.

A very efficient alternative is a flexible voxel allocation mechanism called "sparse grids". Sparse grids is a scheme for allocating memory for voxel data only in parts of the grid that have values differing from a default value. Because clouds have large amounts of empty space around their shape and even within their structure, the grid of density can avoid allocating memory for large fractions of the grid. This in turn allows the options of having bigger grids covering larger volumes at finer resolution. Sparse grids can also be hierarchical in their allocation strategy. In practice, 2000^3 grids and higher, with significant cloud structure, have been created in situations in which the largest non-sparse grid that could be used was 100^3, which is less than 5MB. This example is extreme. The actual efficiency gain depends on the details of the cloud spatial structure. In the opposite extreme, if the cloud had significant density in every voxel, the memory usage would be a little higher than for a simple rectangular grid, because there is some memory usage for the sparseness mapping.

Sparse grids can be implemented on the CPU and GPU such that they have the same data structure, so that gridded data generated on the host can be transfered to the device without conversion. The details of the sparse grid algorithm and implementation are given in appendix C.

6.2 Modeling Techniques for the Density Field

The construction of the density field for an individual cloud involves up to four steps of volumetric data manipulation, each step refining the structure. This section describes each of these steps, followed by a description of how these steps are implemented in Bishop. Chapter 3 of Appendix D describes most of these steps in a very general way. For Gilligan, these steps use some alternative calculations that are faster that the ones presented in Appendix D.

6.2.1 Base Cloud Geometry and Level Set

The starting point for creating the density volume of a cloud is with a basic, smooth, closed surface, created as a polygonal model.

From the polygonal model for the basic surface, a level set representation, $\ell(\mathbf{x})$ is generated on a grid with a resolution appropriately chosen for the simulation. The level set representation of a surface is equivalent to the polygonal geometry, but the volumetric information available for a level set makes volumetric manipulations quicker to perform.

Note that the level set can be used to directly create a simple model for density inside the basic shape. Equation 3.1 of Appendix D is a simple suggestion for a density field with very sharp transitions from full

density to none. The full density value is a dimensionless value of 1. For physical simulations, it is necessary to scale the value by a constant with appropriate units (e.g. g/cm^3).

An alternative with more gradual transition is

$$\rho_{base}(\mathbf{x}) \;=\; \text{clamp}\left(\frac{\ell(\mathbf{x})}{\Delta x_{skin}},\; 0,\; 1\right) \tag{103}$$

In this form, the parameter Δx_{skin} is the "skin thickness" of a transition region that ramps from 0 density to full on density. The function clamp is

$$\text{clamp}(f, a, b) \;=\; \begin{cases} a & f < a \\ f & a \le f \le b \\ b & f > b \end{cases} \tag{104}$$

6.2.2 Cumulative Pyroclastic Displacement of the Base Cloud (Cumulo)

Pyroclastic displacement is a technique for using a noise field to create a shape with a bumpy surface. In chapter 3 of Appendix D, the case is made for how this is useful in cloud modeling, and shows a particular implementation of accumulating multiple iterations of pyroclastic displacement, a process called Cumulo. For Gilligan the approach to iteration has been simplified so that it is easier to implement and more efficient, without giving up the essential features of Cumulo.

The base shape to which displacement is applied is represented as a level set grid or signed distance function. For example, the base shape may begin as a closed polygonal shape that provides a smooth hull for the cloud, from which a level set is generated. This level set is designated $\ell(\mathbf{x})$. We can think of this level set as being a continuous field in space by using an interpolation scheme between the grid points. In practice, tri-linear interpolation is very effective. The convention in Gilligan is that level sets and signed distance functions are positive-valued inside the closed surface, and negative-valued outside.

From any point $\mathbf{x}$ in space, the location of the closest point on the surface is immediately available as output of the Closet Point Transform (CPT),

$$\mathbf{X}_{CPT}(\mathbf{x}) \;=\; \mathbf{x} \;-\; \ell(\mathbf{x})\;\nabla\ell(\mathbf{x}) \tag{105}$$

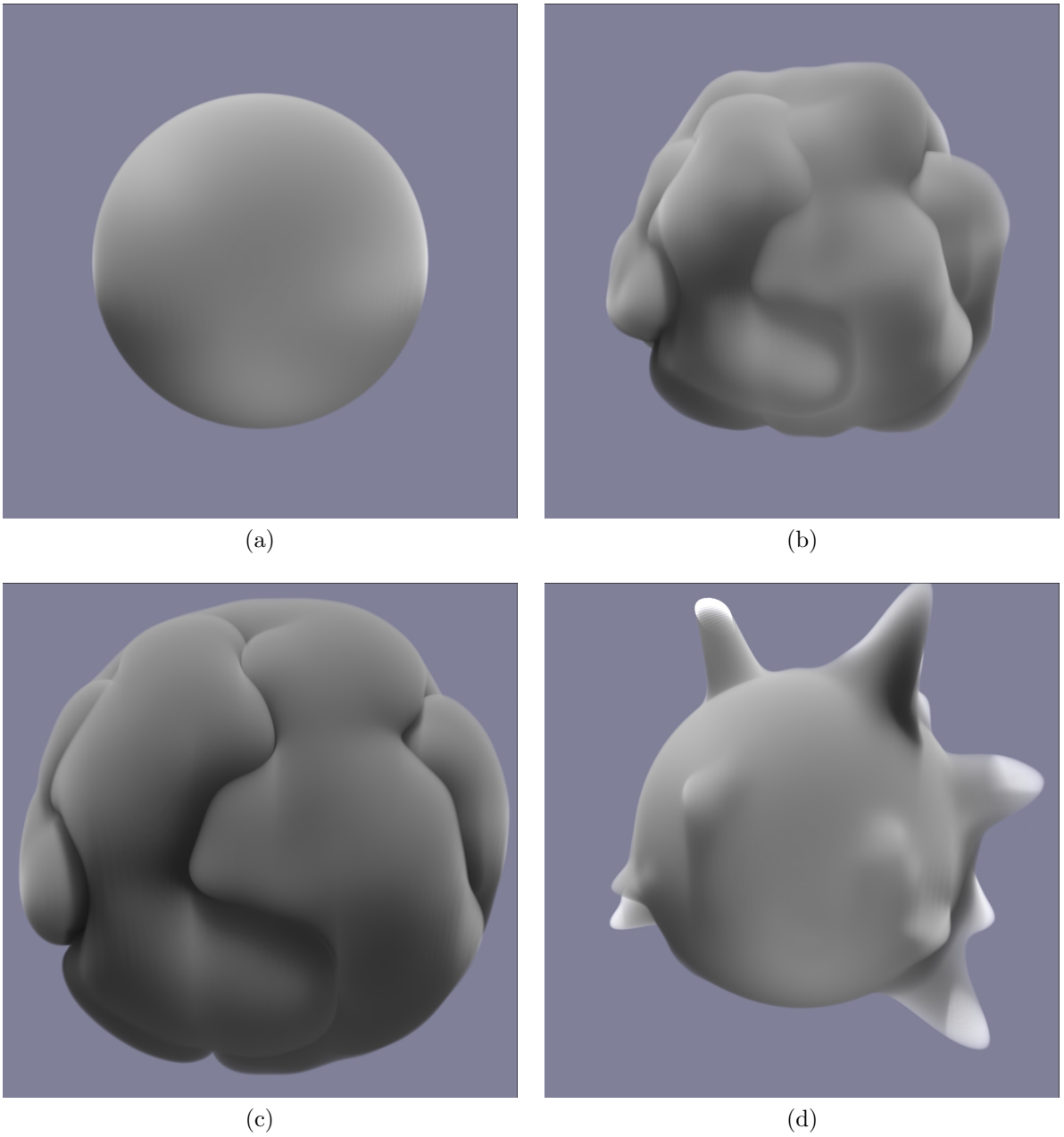
The CPT is an efficient tool for building the first layer of pyroclastic displacement. The displacement is driven by a noise function, $N(\mathbf{x})$ that returns a "noise" value at any point in space. Noise functions such as Perlin noise, cellular noise, and others are frequently chosen. The noise function is typically extended for more detail using fractal summation Ebert et al. [2002], which is assumed to be already implemented in $N(\mathbf{x})$. The displaced signed distance function is

$$\ell_1(\mathbf{x}) \;=\; \ell(\mathbf{x}) \;+\; \left|N\left(\mathbf{X}_{CPT}(\mathbf{x})\right)\right|^{\gamma} \tag{106}$$

The absolute value of the noise is used in order to create sharp, well-defined valleys between pyroclastic "puffs". The parameter γ controls the sharpness of the valleys. To illustrate this pyroclastic displacement, figure 9(a) is a base shape (a sphere) rendered as a volume using equation 103 to generate density from $\ell(\mathbf{x})$. Figure 9(b) shows the volume of $\ell_1(\mathbf{x})$, after choosing scale and fractal values for the Perlin noise function and choosing $\gamma = 1$. Figures 9(c) and 9(d) have other values of γ. These examples illustrate the extreme ranges of structure achievable with pyroclastic displacement.

This displacement can be iterated. From ℓ_1 a "Procedural Point Transform" is constructed as

$$\mathbf{X}^2_{PPT}(\mathbf{x}) \;=\; \mathbf{x} \;-\; \ell_1(\mathbf{x})\;\nabla\ell_1(\mathbf{x}) \tag{107}$$



(a) (b)

(c) (d)

Figure 9: Variations of pyroclastic displacements for different values of γ, using fractal-summed Perlin noise. (a) Base shape (sphere); (b) $\gamma = 1$; (c) $\gamma = 0.15$; (d) $\gamma = 6$.

This is not a CPT because the input function ℓ_1 is not strictly a signed-distance function. Ideally we should be using SDFs and CPTs in each level of iteration. Converting ℓ_1 to a SDF is an option, but takes extra computation time that experience shows is not necessary in many circumstances.

In addition to creating the PPT, the iteration also requires a noise field. For this a fractal repetition of the original noise field is used, i.e.

$$N_2(\mathbf{x}) \;=\; r\; N\left(\mathbf{x}\; f_j\right) \tag{108}$$

The parameter r is called the *roughness* or *lacunarity*, f_j is the *frequency jump*. With this noise field, the iteration of pyroclastic displacement is

$$\ell_2(\mathbf{x}) \;=\; \ell_1(\mathbf{x}) \;+\; \left|N_2(\mathbf{X}^2_{PPT})\right|^\gamma \tag{109}$$

This iteration process is extendable to any number of iterations. Defining the next level of noise as

$$N_{n+1}(\mathbf{x}) \;=\; r\; N_n\left(\mathbf{x}\; f_j\right) \tag{110}$$

the $n+1$ iteration is

$$\mathbf{X}^{n+1}_{PPT}(\mathbf{x}) \;=\; \mathbf{x} \;-\; \ell_n(\mathbf{x})\; \nabla\ell_n(\mathbf{x}) \tag{111}$$

$$\ell_{n+1}(\mathbf{x}) \;=\; \ell_n(\mathbf{x}) \;+\; \left|N_{n+1}(\mathbf{X}^{n+1}_{PPT})\right|^\gamma \tag{112}$$

Figure 10 shows the progress through three levels of cumulo pyroclastic displacement.

6.2.3 Noise Stamping

The term *stamping* refers to the process of filling a 3D grid with values from a volumetric algorithm. For this purpose, the grid can be thought of as a collection of pairs $\{\mathbf{x}_{ijk}, v_{ijk}\}$, where $\mathbf{x}_{ijk}$ is the physical location of gridpoint ijk, and v_{ijk} is the value stored at that location. The grid may be rectangular or some other shape.

Noise stamping fills a grid with values from a noise function. Functions such as Fractal Summed Perlin Noise return values at 3D locations that appear to be noise, although the algorithm does not involve a pseudo-random number generator. Using fractal sums gives control over the range of detail. Other potential noise functions are Fractal Summed Cellnoise, Band-limited FFT Turbulence, and many others Ebert et al. [2002].

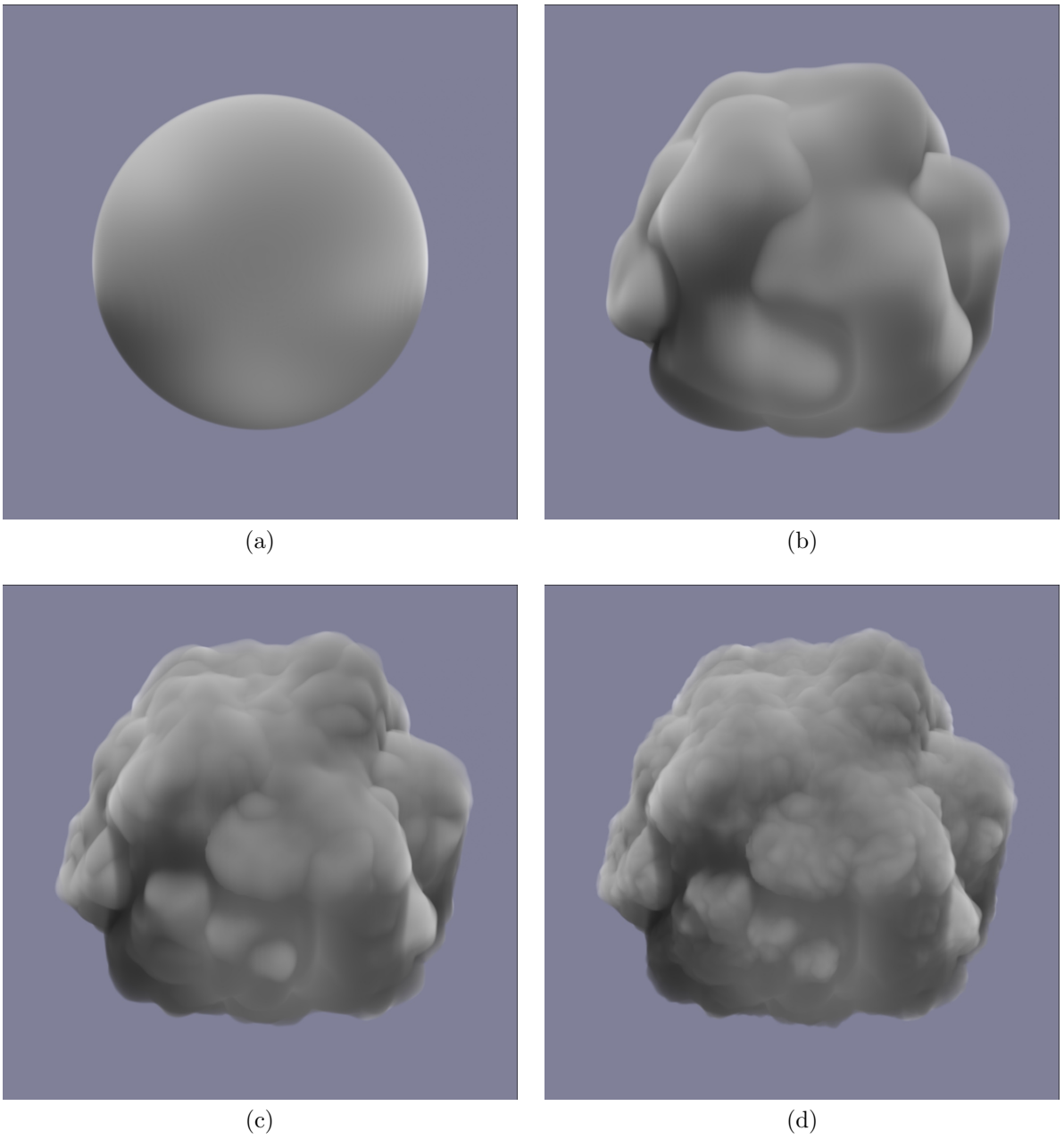
Simply filling the grid with values from noise functions fails to provide control over the shape of the overall structure of the noise. To provide control, a geometric shape must be used to provide a boundary for the region of stampling. Here we discuss two approaches:

Signed Distance Function

Any SDF can serve as a boundary guide. For example, the following could be used

$$v_{ijk} \;=\; \max\left(0,\; N(\mathbf{x}_{ijk})\;\left(\rho_{base}(\mathbf{x}_{ijk})\right)^{fade}\right) \tag{113}$$

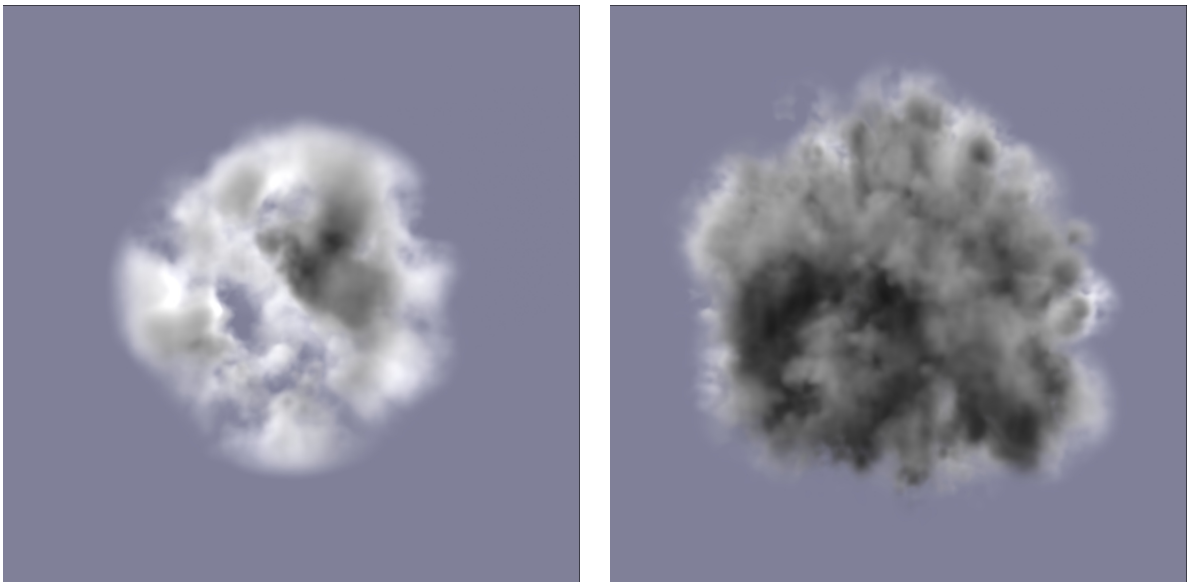
where N is the noise function, ρ_{base} is SDF-based soft mask defined in equation 103, and $fade \geq 0$ is a parameter to control the onset of the the ramp region of the boundary. Using the max function causes the noise pattern to include holes. Figure 11(a) shows Fractal Summed Perlin noise inside a sphere. The fade process helps to reduce the apparent boundary at the edge of the sphere, but it is difficult to make it completely unobvious.



(a) (b)

(c) (d)

Figure 10: Progressive build up of pyroclastic displacements, using Fractal Summed Perlin Noise as the base noise function. (a) Base shape (sphere); (b) one layer of displacement by fractal-summed Perlin noise; (c) two layers of displacement; (d) three layers of displacement.



(a) (b)

Figure 11: Noise fields stamped into a voxel grid. Fractal Summed Perlin Noise as the base noise function. (a) A single sphere of noise, with fading at the boundary $roughness = 0.5$. (b) 900 randomly distributed small spheres stamped with noise; $roughness = 0.75$; $\mathbf{x}_t = random$. For both cases the number of octaves is 4 and the $fjump$ is 2.2.

Replication over many SDFs.

The stamp process for a single SDF can be extended to stamping over many SDFs. The collection of SDFs would be arranged to correspond to a base shape of interest. In order to provide more control and more variability, associated with each SDF there is a collection of parameters for the noise. Since the SDFs typically overlap, the noise at a given grid point has to be blended for all of the SDFs common to that grid point. This sets up an iteration for the value at a grid point as

$$v_{ijk}^{m+1} \;=\; \max\left(v_{ijk}^{m},\; N^{m+1}(\mathbf{x}_{ijk})\;\left(\rho_{base}^{m+1}(\mathbf{x}_{ijk})\right)^{fade}\right) \tag{114}$$

where ρ_{base}^{m+1} is the $m+1th$ SDF, and N^{m+1} is the noise function set with the parameters for the $m+1th$ SDF. Figure 11(b) shows this method applied with 900 small spheres of noise stamping.

6.2.4 Advection

The fourth cloud-construction tool is advection. Advection arises in a dynamical context in many systems involving fluid motion. Here advection is used as a way to mimic that structural impact of motion on the volume of the cloud. The simplest way to apply advection to a field of density or temperature is a process called Semi-Lagrangian advection. For example, a density field $\rho(\mathbf{x})$ is transformed to the field $\rho_{SL}(\mathbf{x})$ via the operation

$$\rho_{SL}(\mathbf{x}) \;=\; \rho\left(\mathbf{x} \;-\; \mathbf{u}(\mathbf{x})\;\Delta t\right) \tag{115}$$

where $\mathbf{u}(\mathbf{x})$ is a velocity field to be discussed below, and Δt is a timestep chosen based on the scenario of the cloud being constructed. Semi-Lagrangian advection is the simplest and least-accurate approximation of the solution for the advection equation. It is important to note that equation 115 can be rewritten as

$$\rho_{SL}(\mathbf{x}) \;=\; \rho\left(\mathbf{X}_{SL}(\mathbf{x}, \Delta t)\right) \tag{116}$$

where $\mathbf{X}_{SL}$ is a vector field with the form

$$\mathbf{X}_{SL}(\mathbf{x}, \Delta t) \;=\; \mathbf{x} \;-\; \mathbf{u}(\mathbf{x})\, \Delta t \tag{117}$$

This vector field is an example of a mapping function called a *Characteristic Map* (CM), and is the basis of a generalized advection discussed below.

The velocity field $\mathbf{u}(\mathbf{x})$ can be chosen from a mix of dynamical and statistical considerations, including a wind vector field. A commonly used method is statistically motivated, using one of a several approaches to create the field:

1. A spatial spectrum can drive the generations of a random realization of velocity values in a grid. A natural spectrum to use could be the Kolmogorov -5/3 power-law, with soft cutoffs at the smallest and largest scales. This approach is analogous to the spectrum-driven random ocean surface realization process discussed in section 2.

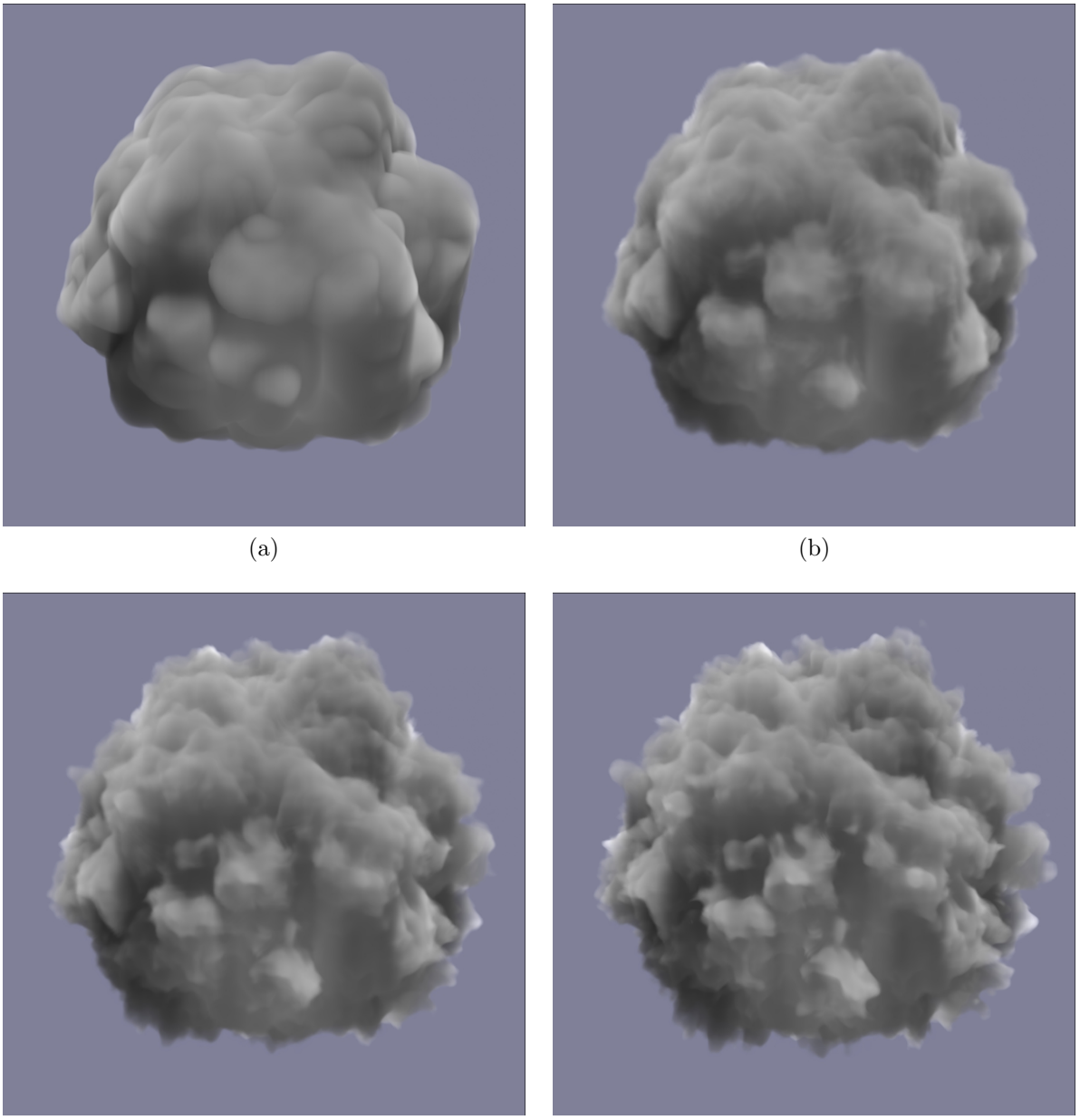
2. Spatial patterns like Perlin Noise and others Ebert et al. [2002] are combined with fractal summation to complex spatial noise. In this approach, the spatial noise is used to create vector-valued noise using the parameter for translation along with various schemes for differentiation.

Figure 12 demonstrates the impact of Semi-Lagrangian advection on the density field. The time step parameter Δt is used to effectively scale the magnitude of the advection step in the Semi-Lagrangian algorithm. Very large time steps can produce spatial structure that is less desireable. This is because the Semi-Lagrangian algorithm is a low-quality approximation of advection, with the error growing rapidly as the time step increases.

One way to adapt to this error is to break up the advection into multiple advections over smaller time steps. For example, if the advection time were $3\Delta t$, one advection would use the CM would be $\mathbf{X}_{SL}(\mathbf{x}, 3\Delta t)$. Alternatively, advection could be applied 3 times, each with a time step of Δt, which would be the CM $\mathbf{X}_{SL}\left(\mathbf{X}_{SL}\left(\mathbf{X}_{SL}\left(\mathbf{x}, \Delta t\right), \Delta t\right), \Delta t\right)$ These CMs correspond to the same overall advection time, but produce very different results because of the error in the Semi-Lagrangian algorithm, as illustrated in figure 13 by the difference between 13(a) and 13(b).

Other advection approaches choose other approximations to advection. Reference Tessendorf [2015] and Appendix E discuss a variety of advection schemes, and ways to evaluate them with different amounts of accuracy. Figure 13 (c) and (d) show the impact of choosing an advection algorithm called Modified MacCormack (which is more accurate than Semi-Lagrangian), and also of evaluating it with very high precision by using many small steps in time.

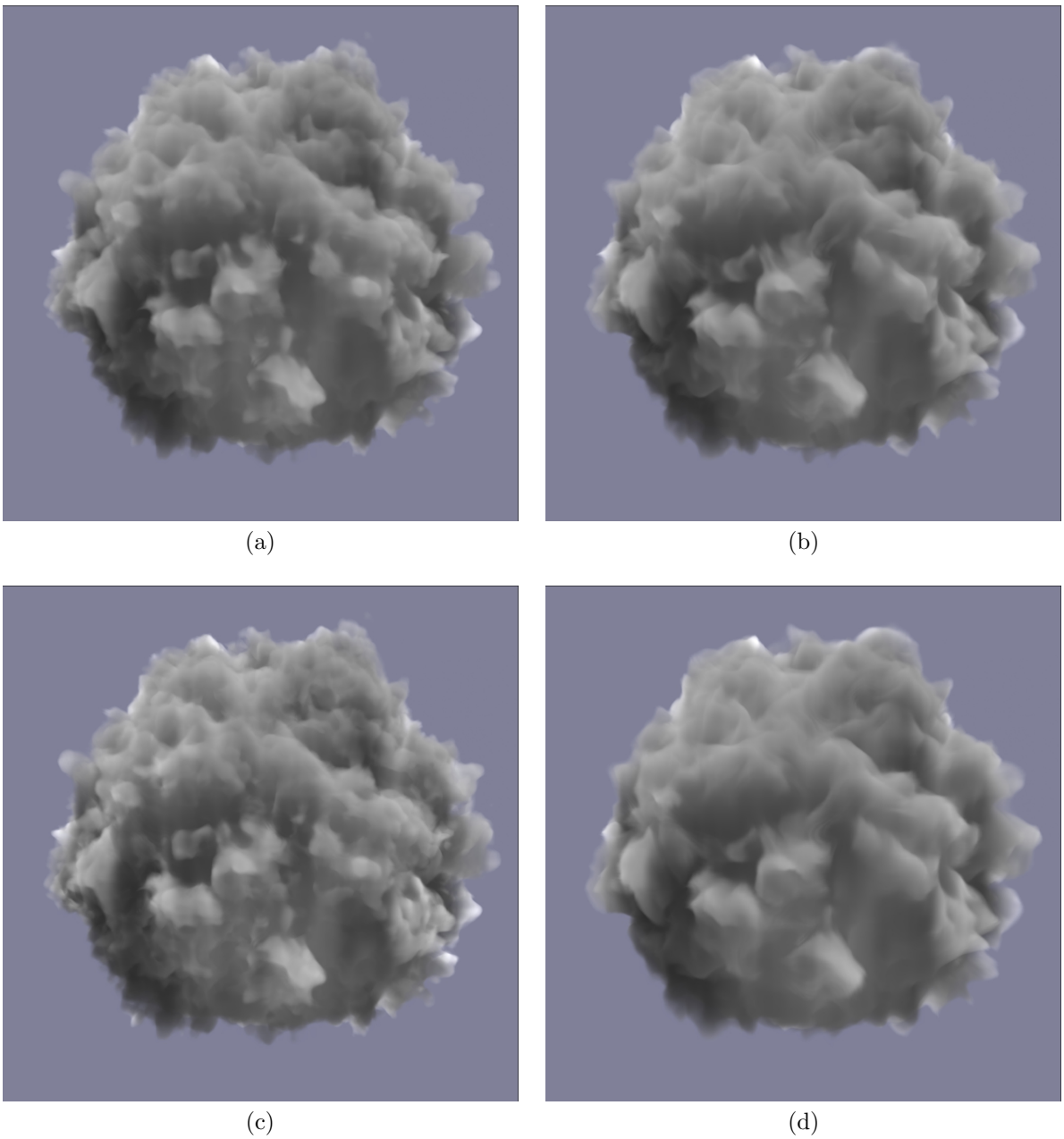
Advection can impact the spatial structure of the cloud in many ways. The choice of advection algorithm is one of the drivers. Another choice is the construction of the velocity field and the time step, and how many advection substeps are used. Another driver is whether the CM is stored in a grid or evaluated analytically. If the CM is stored in a grid, the resolution of the grid is also a driver of the eventual spatial structure of the cloud. This choice of grid, grid resolution, or procedural CM is important becasue advection of material induces fine spatial structure. In turbulence, there is a notion of an energy cascade from one scale of motion to smaller scales (in 3D) or to large scales (in 2D), and the mechanism of this energy transfer between scales is advection. This is turn means that the spatial scales represented in the CM have an impact on the details of the fine structure that is created.



(a) (b)

(c) (d)

Figure 12: Semi-Lagrangian advection applied on top of pyroclastic displacement. (a) Unadvected cloud (with two layers of pyroclastic displacement). (b) Cloud in (a) advected with one small time step ($\Delta t = 0.035$) over a noise velocity field. (c) Advected with time step $2\Delta t$. (d) Advected time step $3\Delta t$.



(a) (b)

(c) (d)

Figure 13: Comparison of strategies to apply advection. (a) Semi-Lagrangian advection for a single step of time $3\Delta t$; (b) Three sequential semi-lagrangian advections, each with time step Δt; (c) Modified-MacCormack advection for a single step of time $3\Delta t$; (d) Modified-MacCormack log-advection for 5 log-steps (32 subframe steps) with total time $3\Delta t$. See Tessendorf [2015] and Appendix E for definitions of various advection schemes.

6.2.5 Bishop Workflow

For creating a single cloud, the task begins with a *base shape* for the cloud, in one of two forms: (1) a simple closed-volume polygonal geometry that is converted into an sdf, or (2) a collection of particles with attached attributes for a spherical radius and noise parameters.

For a base shape composed of polygonal geometry, all three modification processes are available as options to generate the cloud density. For a collection of particles, both noise stamping and advection are options. In all cases, choices of modification processes and parameter settings are made based on the desired outcome for the cloud shape. The cloud data can be stored in grids at any step of the processing, and grids can be written to disk for later use.

A cloud system can be assembled from a collection of individual clouds that have been accumulated in a library. If the individual clouds overlap, their density and temperature are combined by using the values at any point in space from the cloud with the largest density.

For GPU applications, the cloud fields (density, temperature, etc) are generated as sparse grids on the CPU, then transfered to the GPU for rendering.

6.3 Modeling Techniques for the Temperature Field

For IR wavelengths, the temperature field in the cloud will be important. Gilligan will include two modeling components for the temperature field. The basic level is a hydrostatic atmosphere behavior, in which the temperature decreases with altitude in the cloud volume. The second component is a temperature fluctuation within the cloud structure. The spatial structure of the temperature fluctuation will be assembled from the three methods of modifying fields discussed in this section. The scaling magnitude of the fluctuations is a user input.

6.4 Evolution of Cloud Structure

Clouds evolve. If the spatial structure is thought of as a random process evolving in time, the decoherence time of a clouds can be as small at tens of seconds, up to many minutes. To capture this evolution, several mechanisms will be available in Gilligan:

1. The base shape can evolve via animation of the polygonal geometry or animation of the particle collection.
2. The parameters used in pyroclastic displacement can evolve.
3. The parameters in noise stamping can evolve.
4. The amount of advection can increase over time.
5. The advection velocity field could evolve based on the underlying procedure used to create it.

Simple animations of one or a few noise parameters can induce very natural evolution of the structure of a cloud.

7 Rendering Clouds

Clouds are rendered in Gilligan using ray marching. This algorithm is sufficiently simple to run on both the CPU and GPU. Both emission and scattering are captured by ray marching, but the scattering process currently in Gilligan is just single scattering.

7.1 Ray Marching

The ray marching algorithm in Gilligan is described in Appendix D. In addition, Gilligan uses acceleration methods based on axis-aligned bounding boxes and kdtrees to accelerate the rendering by efficiently stepping through empty portions of the volume. The scattering calculation is accelerated by pre-computing the transmissivity to the light source and storing it in a *Deep Shadow Map* (DSM) grid (appendix D). Light sources are modeled as point lights (directional lights set up as point lights at extreme distance). If the point light is exterior to the volume, the DSM is stored in a frustum-shaped grid that is adapted to the size of the render volume. If the light is interior to the volume, the DSM grid is rectangular and the same size as the render volume. The resolution of the DSM grid affects the softness of the shadows created by the transmissivity, i.e. lower number of grid points leads to softer shadows, but much faster computation of the transmissivity. To improve the quality of the linear interpolation when the DSM is sampled, the values actually stored in the grid are proportional to the log of the transmissivity. These values are interpolated then exponentiated to recover the transmissivity. In all cases, the DSM grid is a sparse grid, and values are not computed at grid points where the cloud density is zero, because the DSM value is not needed. This approach speeds up the DSM generation and allows higher resolution grids because of the memory-savings of the grid sparseness.

7.2 Material Shading

Material shading within the ray marcher generates the components of the spectral radiance at any point along the ray march, based the local properties of the cloud. The two critical models are the choice of phase function, and blackbody emission.

7.2.1 Single Scattering Phase Function

The ray marcher will have several pre-built choices for phase functions:

1. Uniform: a constant value of $1/4\pi$ for all scattering angles.
2. Henyey-Greenstein to capture strong forward scattering
3. Double Henyey-Greenstein to capture for strong forward scattering and a secondary backscatter peak.
4. Fournier-Forand to capture more features of natural materials than the others.

None of these phase functions are capable of producing rainbows or sundogs. More sophisticated phase functions must be used for those phenomena.

7.2.2 Blackbody Emission

For blackbody emission (the C^E term in equation (A.3) in Appendix D), the formula for blackbody radiance will be used in Gilligan based on the local value of the temperature field and the particular wavelengths selected for rendering.

8 Rendering Platforms

Gilligan supports rendering on CPUs with a raytrace renderer, and rendering on GPUs with a hybrid of OpenGL rendering and CUDA-based volumetric raymarching. Both rendering platforms feed the image data to an image viewing package for display.

8.1 Rendering viewer: Skipper

Gilligan supports two methods of handling rendered images. One is to write the image to the filesystem in OpenEXR format. This format supports float image data, any number of channels, and open metadata inclusion. Routine experience with OpenEXR with images exceeding 40+ channels has been very good. To aid efficiency, OpenEXR losslessly compresses each channel using zip. The metadata will include all of the input parameters to the simulation and render, user info, timing and date information. To simplify access to OpenEXR, Gilligan uses the OSS package OpenImageIO, which provides a common image read/write framework to many file formats (e.g. JPG, TIFF, and PNG), while hiding the complexities and picadillos of those image formats.

The second output mechanism is to pipe the image data directly to an image viewer, bypassing filesystems. The Skipper image viewer will be based on pyside or glut, and use OpenGL to display the image, update the image when a new or revised one becomes available, and allow scaling, scrubbing, gamma and brightness corrections.

8.2 Rendering on CPU: MaryAnn

The CPU-based ray trace renderer in Gilligan is a repackaging and extension of a student-built renderer called Ash. It is an efficient ray tracer that uses BVH to accelerate tracing, and C++-based shaders to control many aspects of the rendering process, including surface shading. Although Ash is a straight-up ray tracer, students have built Global Illumination renders from Ash based on path tracing, bi-directional path tracing, and point-based global illumination by crafting suitable Ash shaders, without modifying the ray tracing code. Scene assembly, material attachment, camera definition, and any other processing logic is conducted in python scripts. All C++ features of Ash are accessble from python using binding generated using Swig (Simple Wrapper Interface Generator).

MaryAnn is a repurposing of the Ash code base, and differs from the Ash code in several ways:

1. The Ash code base includes a number of student projects using Ash. These are absent from the MaryAnn code.

2. MaryAnn includes a collection of python modules to streamline the generation and rendering of a scene in a manner appropriate for the goals of Gilligan.

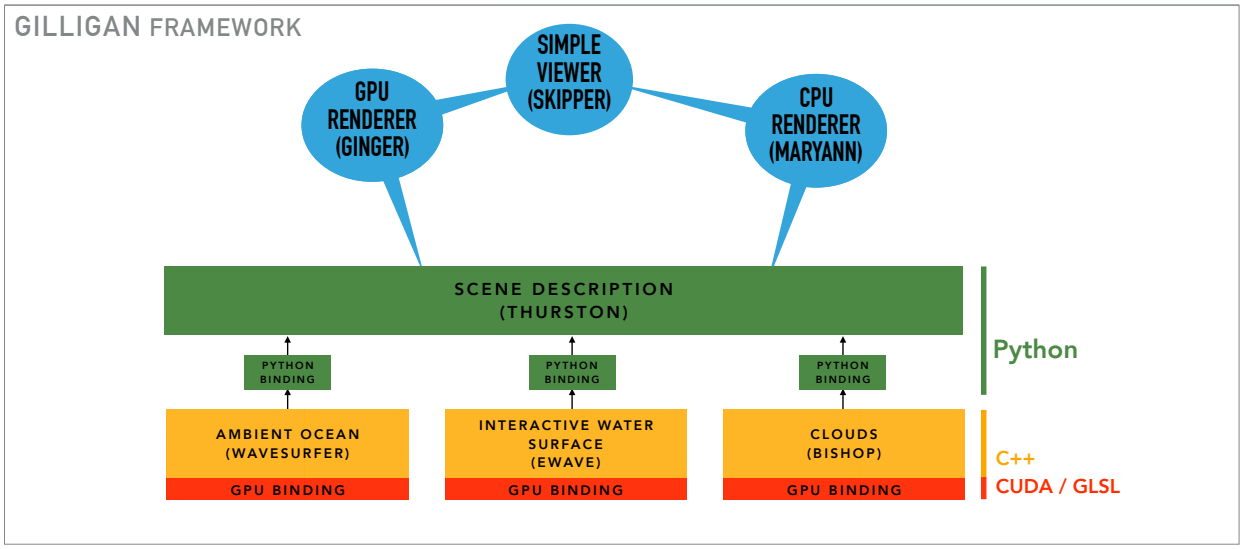


Figure 14: Visual illustration of the data and conceptual connectivity of the component features of Gilligan. Parentheses indicate the name of existing or planned software modules.

8.3 Rendering on GPU: Ginger

Water surfaces rendered on the GPU will be accomplished via the traditional OpenGL pipeline. GLSL shaders for surface shading will be written. Vertex data will come into the OpenGL pipeline either via vertex buffers (when the water surface is generated on the CPU), or via CUDA access to the vertex buffer memory space when the water surface is generated on the GPU with CUDA. The combination of python scripts, GLSL shaders, and CUDA to accomplish this is called Ginger.

9 Simulation and Rendering Framework

Figure 14 illustrates the connectivity of the various components of Gilligan. The clouds portion of the framework is described in Part II of this document. The components labelled WaveSurfer, eWave, and Bishop existed individually as C++ code prior to the creating of Gilligan. Much of the content of the CPU-based renderer MaryAnn existed as a C++ renderer named Ash. The Python-based scene description component called Thurston is a rewrite and refactoring of existing Python code. The completely new components in Gilligan are the GPU renderer Ginger, the viewer Skipper, and the GPU portions of WaveSurfer, eWave, and Bishop.

10 Existing Code

Priority is a numeric ranking, with higher values meaning higher importance and/or degree of difficulty.

10.1 WaveSurfer

Description: Computes FFT-based spectral model ocean waves at any input time. Dispersion includes parameters for deep, shallow, and capillary waves.

Platform: C++ and Python

Modifications Needed: (1) Add the various spectral models in section 2.3. (2) Add drift velocity option.

Modification Priority: 2

10.2 eWave

Description: Simulates dispersive water waves in response to sources and obstructions intersecting the water surface. Simulates scattered waves from incident ambient waves scattering from obstructions.

Platform: C++ and Python

Modifications Needed: Wrapper code to encapsulate as a WaveSurface object.

Modification Priority: 1

10.3 Bishop

Description: A nearly-comprehensive collection of volume modeling and rendering tools. Modeling is structured as a procedural process that also supports gridded and particle data, and simulation of any quantities. A volume renderer performs single-scatter raymarching. This is also a research tool for studying approaches to rendering multiple scattering in radiative transfer.

Platform: C++ and Python

Modifications Needed: Add indexed sparse grid data structure (see Appendix C)

Modification Priority: 4

10.4 Ash

Description: Student-built efficient ray-trace renderer. Very flexible control via C++-based shaders for material and other computations. Scenes are assembled in python. Ash will be cleaned of extraneous student projects, bundled with new python scripts for scene simulation (see Thurston in section 11.9), and renamed MaryAnn.

Platform: C++ and Python

Modifications Needed: New hyperspectral shader for water surfaces including reflection, refraction, and blackbody emission.

Modification Priority: 3

11 New Code

Priority is a numeric ranking, with higher values meaning higher importance and/or degree of difficulty.

11.1 Device-Agnostic C++ Model for Data and Processing

Description: Multiple components of Ginger must have the option of doing their processing on either the CPU or the GPU. Data may exist on one device and need to be transfered to the other. This is an attempt to develop a generic capability to standardize these common activities, and encapsulate them in a small C++ framework. This will simplify the development of GPU functionality for each relevant component, and provide a simple C++ interface for Python bindings to handle.

Priority: 10 (very high because it is a new development. In the event it is late or unworkable, there is a backup plan to assure that GPU functionality happens, albeit in a less systematic and efficient fashion.)

Platform: C++ and CUDA

11.2 Fast Meshing of Water Surface

Description: The existing method in WaveSurfer and eWave to create a displaced mesh and import it into the renderer is via an obj file created on the filesystem. The code for generating the obj data and writing the file is in Python. Both of these aspects severely limit the speed of the complete similuation-render-display cycle. A faster C++-based meshing routine is being written that will transfer the mesh data directly to MaryAnn or Ginger without the intermediate filesystem delay. An alternative technique that could be considered at a later date reference Tevs et al. [2008].

Priority: 5

Platform: C++ and CUDA

11.3 OpenGL Render of Water Surface

Description: When rendering in OpenGL on the GPU, the surface material properties are best handled using a GLSL fragment shader. This is applicable when rendering on the GPU, regardless of whether the displaced surface originates on the GPU or the CPU.

Priority: 4

Platform: C++ and OpenGL

11.4 CUDA Simulation of FFT Ocean Surface

Description: Implementation of WaveSurfer code in CUDA to construct the Fourier amplitudes, and generate wave height and horizontal displacements using the CUDA FFT routine. These displacements will be optionally combined with other displacements, and the combined surface will be applied to a mesh already stored in a GPU vertex buffer. The CUDA computation is encapsulated within a purely C++ interface, which in turn has Python bindings, so that the user does not directly interact with CUDA.

Priority: 3

Platform: C++ and CUDA

11.5 CUDA Simulation of Interactive Water Surface

Description: Implementation of eWave code in CUDA to construct the simulate wave height and velocity potential, generating wave height and horizontal displacements using the CUDA FFT routine. These displacements will be optionally combined with other displacements, and the combined surface will be applied to a mesh already stored in a GPU vertex buffer. The CUDA computation is encapsulated within a purely C++ interface, which in turn has Python bindings, so that the user does not directly interact with CUDA.

Priority: 3

Platform: C++ and CUDA

11.6 Skipper Image Viewer

Description: Basic viewer of rendered images. Runs an OpenGL display of the image. The image is represented in OpenGL as a texture on a flat rectangular surface, allowing simple and responsive zoom and translate, and adjustment of gamma and brightness interactively. User selects the channels to be displayed. Display is refreshed when the image is updated or replaced, allowing it to act as a realtime playback. Option to write an image to the filesystem in OpenEXR format.

Priority: 5

Platform: C++, Python, and Pyside or GLUT

11.7 Ginger GPU Renderer

Description: The assembled collection of python, C++, CUDA, and GLSL components for rendering a scene using OpenGL rendering.

Priority: 3

Platform: Python, C++ and CUDA

11.8 MaryAnn CPU Renderer

Description: A repackaging of the Ash raytrace renderer and controling python scripts.

Priority: 2

Platform: Python, C++

11.9 Thurston Scene Assembly and Control

Description: A collection of python scripts and C++-bound components to assemble geometry, simulations, materials, light(s) and renderer (MaryAnn or Ginger), execute the assembled processes, render the scene, and pass the rendered image to the Skipper image viewer.

Priority: 7

Platform: Python

12 Support

This work is supported by SPAWAR Systems Center Pacific under award number N66001-16-P-6865.

References

Siggraph course notes: Production volume rendering, 2011. URL http://magnuswrenninge.com/productionvolumerendering. [Online; accessed 10-August-2016].

Oliver Deusen, David S. Ebert, Ron Fedkiw, F. Kenton Musgrave, Przemyslaw Prusinkiewicz, Doug Roble, Jos Stam, and Jerry Tessendorf. *The elements of nature: Interactive and realistic techniques*. Association for Computing Machinery, Inc, 8 2004. ISBN 9780000000002. doi: 10.1145/1103900.1103932.

David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002. ISBN 1558608486.

Kara Gundersen. Ocean surface shader. Master's thesis, School of Computing, Clemson University, December 2015. http://tigerprints.clemson.edu/all_theses/2316/.

IAPWS. Release on the refractive index of ordinary water substance as a function of wavelength, temperature and pressure. 1997. URL http://www.iapws.org/relguide/rindex.pdf.

Stanislaw R. Massel. *Ocean Surface Waves Their Physics and Prediction*, volume 36 of *Advanced Series on Ocean Engineering*. World Scientific, 2 edition, 2013.

Trent Polack and Andre LaMothe. *Focus on 3D Terrain Programming*. Permier Press, 2003. ISBN 1-59200-028-2.

Donald A. Stentz. A Chronological Study of the Measurement of the Optical Properties of Ocean Water and an Atlas of the Diffuse Attentuation Coefficient, k, of Tropical Atlantic Ocean Waters. Technical report, June 1975.

Jerry Tessendorf. eWave: Using an Exponential Solver on the iWave Problem. Technical report, March 2014. URL https://people.cs.clemson.edu/~jtessen/reports.html.

Jerry Tessendorf. Advection Solver Performance with Long Time Steps, and Strategies for Fast and Accurate Numerical Implementation. Technical report, February 2015. URL https://people.cs.clemson.edu/~jtessen/reports.html.

Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 183–190, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-983-8. doi: 10.1145/1342250.1342279. URL http://doi.acm.org/10.1145/1342250.1342279.

Egbert S. Tse, John McGill, and Robert L. Kelly. Coherent whitecap and glitter simulation model, 1990. URL http://dx.doi.org/10.1117/12.21468.

Wikipedia. Gilligan's island, 2016. URL https://en.wikipedia.org/wiki/Gilligan%27s_Island. [Online; accessed 10-August-2016].

Appendix A Siggraph Course Notes

This appendix contains the relevant content from reference Deusen et al. [2004].

Simulating Ocean Water

Jerry Tessendorf

1 Introduction and Goals

These notes are intended to give computer graphics programmers and artists an introduction to methods of simulating, animating, and rendering ocean water environments. CG water has become a common tool in visual effects work at all levels of computer graphics, from print media to feature films. Several commercial products are available for nearly any computer platform and work environment. A few visual effects companies continue to extend and improve these tools, seeking to generate higher quality surface geometry, complex interatctions, and more compelling imagery. In order for an artist to exploit these tools to maximum benefit, it is important that he or she become familiar with concepts, terminology, a little oceanography, and the present state of the art.

As demonstrated by the pioneering efforts in the films *Waterworld* and *Titanic*, as well as several other films made since about 1995, images of cg water can be generated with a high degree of realism. However, this level of realism has been mostly limited to relatively calm, nice ocean conditions. Conditions with large amounts of spray, breaking waves, foam, splashing, and wakes are improving and approaching the same realistic look.

Three general approaches are currently popular in computer graphics for simulating fluid motion, and water surfaces in particular. The three methods are all related in some way to the basic Navier-Stokes equations at the heart of many applications. Computational Fluid Dynamics (CFD) is one of the methods which has recieved a great amount of attention lately. While many versions of this technology have been in existence for some time, recent papers by Stam [1], Fedkiw and Foster [2], and many others have demonstrated that the numerical computation discipline and the computer graphics discipline have connected well enough to produce useful, interesting, and sometimes beautiful results. The primary drawback of CFD methods is that the computations are performed on data structured in a 2D or 3D grid, called an Eulerian framework. This gridded architecture limits the combined extent and flow detail that can be computed. At present for example, it is practical to simulate with CFD waves breaking as they approach a shallow beach. However, that simulation is not able to simulate a bay-sized region of ocean while also simulating the breaking down to the detail of spray formation, simply because the number of grid points needed is prohibitively large.

A second method that is shows great promise is called Smoothed Particle Hydrodynamics (SPH). This is a completely different approach to solving the Navier-Stokes equations. SPH imagines that the volume of a fluid is composed of small overlapping regions, the center of each region carrying some amount of mass and momentum. The regions are allowed to move about within the fluid and experience forces due to pressure, strain, gravity, and others. But now the center of each region acts as a particle, and the Navier-Stokes equations are converted into equations of motion for discrete particles. This approach is called a Lagragian framework (as opposed to the Eulerian framework in CFD). The fluid volume is not bound to a grid geometry. SPH is a very useful method of simulation for situations in which there is significant splashing or explosions, and has even been used to simulated cracking in solids [3]. Using implicit methods to construct a surface for the fluid, standard computer graphics applications such as pouring water have been achieved [4].

Finally, the third method is the one that is the focus of these notes, and unlike the CFD and SPH methods, this one is focused on the more narrow goal of simulating the motion of the surface of a body of water. Surface simulations are commonly generated from the CFD and SPH methods, but the surface is generated by algorithms that are added to those fluid simulations, for example by tracking a type of implicit surface called a level set. In choosing to focus on the surface structure and motion, we eliminate much of the computation and resolution limits in the CFD and SPH methods. Of course when we eliminate those computations we have lost certain types of realistic motion, most notably the breakup of the surface with strong changes in topology. In place of that computation we substitute a mixture of knowledge of oceanographic phenomenology and computational flexibility to achieve realistic types of surfaces that cannot practically be achieved by the others. In that sense, this phemonological approach should be considered complementary – not competitive – to the CFD and SPH methods, since all three work best in different regimes of fluid motion.

Broadly, the reader should come away from this material with

1. an understanding of the important physical concepts for ocean surface propagation, most notably the concept of dispersion and types of dispersion relationships.

2. an understanding of some algorithms that generate/animate water surface height fields suitable for modeling waves as big as storm surges and as small as tiny capillaries;

3. an understanding of the basic optical processes of reflection and refraction from a water surface;

4. an introduction to the color filtering behavior of ocean water;

5. an introduction to complex lighting effects known as caustics and godrays, produced when sunlight passes through the rough surface into the water volume underneath; and

6. some rules of thumb for which choices make nice looking images and what are the tradeoffs of quality versus computational resources. Some example shaders are provided, and example renderings demonstrate the content of the discussion.

Before diving into it, I first want to be more concrete about what aspect of the ocean environment we cover (or not cover) in these notes. Figure 1 is a rendering of an oceanscape produced from models of water, air, and clouds. Light from the clouds is reflected from the surface. On the extreme left, sun glitter is also present. The generally bluish color of the water is due to the reflection of blue skylight, and to light coming out of the water after scattering from the volume. Although these notes do not tackle the modeling and rendering of clouds and air, there is a discussion of how skylight from the clouds and air is reflected from, or refracted through, the water surface. These notes will tell you how to make a height-field displacement-mapped surface for the ocean waves with the detail and quality shown in the figure. The notes also discuss several effects of the underwater environment and how to model/render them. The primary four effects are sunbeams (also called godrays), caustics on underwater surfaces, blurring by the scattering of light, and color filtering.

There are also many other complex and interesting aspects of the ocean environment that will not be covered. These include breaking waves, spray, foam, wakes around objects in the water, splashes from bodies that impact the surface, and global illumination of the entire ocean-atmosphere environment. There is substantial research underway on these topics, and so it is possible that future versions of this or other lecture notes will include them. I have included a brief section on advanced modifications to the basic wave height algorithm that produce choppy waves. The modification could feasibly lead to a complete description of the surface portion of breaking waves, and possibly serve to drive the spray and foam dynamics as well.

There is, of course, a substantial body of literature on ocean surface simulation and animation, both in computer graphics circles and in oceanography. One of the first descriptions of water waves in computer graphics was by Fournier and Reeves[12] , who modeled a shoreline with waves coming up on it using a water surface model called *Gerstner waves*. In that same issue, Darwin Peachey[13]

Figure 1: Rendered image of an oceanscape.

presented a variation on this approach using basis shapes other than sinusoids.

In the oceanographic literature, ocean optics became an intensive topic of research in the 1940s. S.Q. Duntley published[17] in 1963 papers containing optical data of relevance to computer graphics. Work continues today. The field of optical oceanography has grown into a mature quantitative science with subdisciplines and many different applications. One excellent review of the state of the science was written by Curtis Mobley[18].

In these lectures the approach we take to creating surface waves is close to the one outlined by Masten, Watterberg, and Mareda[11], although the technique had been in use for many years prior to their paper in the optical oceanography community. This approach synthesizes a patch of ocean waves from a Fast Fourier Transform (FFT) prescription, with user-controllable size and resolution, and which can be tiled seamlessly over a larger domain. The patch contains many octaves of sinusoidal waves that all add up at each point to produce the synthesized height. The mixture of sinusoidal amplitudes and phases however, comes from statistical, emperically-based models of the ocean. What makes these sinusoids look like waves and not just a bunch of sine waves is the large collection of sinusoids that are used, the relative amplitudes of the sinusoids, and their animation using the dispersion relation. We examine the impact of the number of sinusoids and resolution on the quality of the rendered image.

In the next section we begin the discussion of the ocean environment with a broad introduction to the global illumination problem. The radiosity equations for this environment look much like those of any other radiosity problem, although the volumetric character of some of the environmental components complicate a general implementation considerably. However, we simplify the issues by ignoring some interactions and replacing others with models generated by remote sensing data.

Practical methods are presented in section 4 for creating realizations of ocean surfaces. We present two methods, one based on a simple model of water structure and movement, and one based on summing up large numbers of sine waves with amplitudes that are related to each other based on experimental evidence. This second method carries out the sum using the technique of Fast Fourier Transformation (fft), and has been used effectively in projects for commercials, television, and motion pictures.

After the discussion of the structure and animation of the water surface, we focus on the optical properties of water relevant to the graphics problem. First, we discuss the interaction at the air-water interface: reflection and refraction. This leaves us with a simple but effective Renderman-style shader suitable for rendering water surfaces in BMRT, for example. Next, the optical characteristics of the underwater environment are explored.

Finally, please remember that these notes are a living document. Some of the discussion of the various topics is still very limited and incomplete. If you find a problem or have additional questions, please feel free to contact me at jerry@finelightvisualtechnology.com. The latest version of this course documents are hosted at http://www.finelightvisualtechnology.com

2 Radiosity of the Ocean Environment

The ocean environment, for our purposes, consists of only four components: The water surface, the air, the sun, and the water volume below the surface. In this section we trace the flow of light through the environment, both mathematically and schematically, from the light source to the camera. In general, the radiosity equations here are as coupled as any other radiosity problem. To a reasonable degree, however, the coupling can be truncated and the simplified radiosity problem has a relatively fast solution.

The light seen by a camera is dependent on the flow of light energy from the source(s) (i.e. the sun and sky) to the surface and into the camera. In addition to specular reflection of direct sunlight and skylight from the surface, some fraction of the incident light is transmitted through the surface. Ultimately, a fraction of the transmitted light is scattered by the water volume back up through the interface and into the air. Some of the light that is reflected or refracted at the surface may strike the surface a second time, producing more reflection and refraction events. Under some viewing conditions, multiple reflections and refractions can have a noticeable impact on images. For our part however, we will ignore more than one reflection or refraction from the surface at a time. This not only makes the algorithms and computation easier and faster, but also is reasonably accurate in most viewing conditions and produces visually realistic imagery.

At any point in the environment above the surface, including at the camera, the total light intensity (radiance) coming from any direction has three contributions:

$$L_{ABOVE} = rL_S + rL_A + t_U L_U \ , \tag{1}$$

with the following definitions of the terms:

- r is the Fresnel reflectivity for reflection from a spot on the surface of the ocean to the camera.
- t_U is the transmission coefficient for the light L_U coming up from the ocean volume, refracted at the surface into the camera.
- L_S is the amount of light coming directly from the sun, through the atmosphere, to the spot on the ocean surface where it is reflected by the surface to the camera.
- L_A is the (diffuse) atmospheric skylight
- L_U is the light just below the surface that is transmitted through the surface into the air.

Equation 1 has intentionally been written in a shorthand way that hides the dependences on position in space and the direction the light is traveling.

While equation 1 appears to have a relatively simple structure, the terms L_S, L_A, and L_U can in principle have complex dependencies on each other, as well on the reflectivity and transmissivity.

There is a large body of research literature investigating these dependencies in detail [19], but we will not at this point pursue these quantitative methods. But we can elaborate further on the coupling while continuing with the same shorthand notation. The direct light from the sun L_S is

$$L_S = L_{TOA} \exp\{-\tau\} , \tag{2}$$

where L_{TOA} is the intensity of the direct sunlight at the top of the atmosphere, and τ is the "optical thickness" of the atmosphere for the direction of the sunlight and the point on the earth. Both the diffuse atmospheric skylight L_A and the upwelling light L_U can be written as the sum of two terms:

$$L_A = L_A^0(L_S) + L_A^1(L_U) \tag{3}$$
$$L_U = L_U^0(L_S) + L_U^1(L_A) \tag{4}$$

These equations reveal the potential complexity of the problem. While both L_A and L_U depend on the direct sunlight, they also depend on each other. For example, the total amount of light penetrating into the ocean comes from the direct sunlight and from the atmospheric sunlight. Some of the light coming into the ocean is scattered by particulates and molecules in the ocean, back up into the atmosphere. Some of that upwelling light in turn is scattered in the atmosphere and becomes a part of the skylight shining on the surface, and on and on. This is a classic problem in radiosity. It is not particularly special for this case, as opposed to other radiosity problems, except perhaps for the fact that the upwelling light is difficult to compute because it comes from volumetric multiple scattering.

Our approach, for the purposes of these notes, to solving this radiosity problem is straightforward: take the skylight to depend only on the light from the sun, since the upwelling contribution represents a "tertiary" dependence on the sunlight; and completely replace the equation for L_U with an empirical formula, based on scientific observations of the oceans, that depends only on the direct sunlight and a few other parameters that dictate water type and clarity.

Under the water surface, the radiosity equation has the schematic form

$$L_{BELOW} = tL_D + tL_I + L_{SS} + L_M , \tag{5}$$

with the meaning

- t is the Fresnel transmissivity for transmission through the water surface at each point and angle on the surface.
- L_D The "direct" light from the sun that penetrates into the water.
- L_I The "indirect" light from the atmosphere that penetrates into the water.
- L_{SS} The single-scattered light, from both the sun and the atmosphere, that is scattered once in the water volume before arriving at any point.
- L_M The multiply-scattered light. This is the single-scattered light that undergoes more scattering events in the volume.

Just as for the above water case, these terms are all related to each other is relative complex ways. For example, the single scattered light depends on the direct and indirect light:

$$L_{SS} = P(tL_I) + P(tL_D) \tag{6}$$

with the quantity P being a linear functional operator of its argument, containing information about the single scattering event and the attenuation of the scattered light as it passes from the scatter point to the camera. Similarly, the multiply-scattered light is dependent on the single scattered:

$$L_M = G(tL_I) + G(tL_D) . \tag{7}$$

The functional schematic quantities P and G are related, since multiple scattering is just a series of single scatters. Formally, the two have an operator dependence that has the form

$$\begin{aligned} G &\sim P \otimes P \otimes \left\{1 + P + \frac{1}{2!}P \otimes P + \frac{1}{3!}P \otimes P \otimes P + \ldots\right\} \\ &\sim P \otimes P \otimes \exp(P) . \end{aligned} \tag{8}$$

At this point, the schematic representation may have outlived its usefullness because of the complex (and not here defined) meaning of the convolution-like operator $\otimes$, and because the expression for G in terms of P has created an even more schematic view in terms of an exponentiated P. So for now we will leave the schematic representation, and journey on with more concrete quantities the rest of the way through.

The formal schematic discussion put forward here does have a mathematically and physically precise counterpart. The field of study in Radiative Transfer has been applied for some time to water optics, by a large number researchers. The references cited are excellent reading for further information.

As mentioned, there is one additional radiosity scenario that can be important to ocean rendering under certain circumstances, but which we will not consider. The situation is illustrated in figure 2. Following the trail of the arrows, which track the direction light is travelling, we see that sometimes light coming to the surface (from above or below), can reflect and/or transmit through the surface more than once. The conditions which produce this behavior in significant amounts are: the wave heights must be fairly high, and the direction of viewing the waves, or the direction of the light source must be nearly grazing the surface. The higher the waves are, the less grazing the light source or camera need to be. This phenonmenon has been examined experimentally and in computer simulations. It is reasonably well understood, and we will ignore it from this point on.

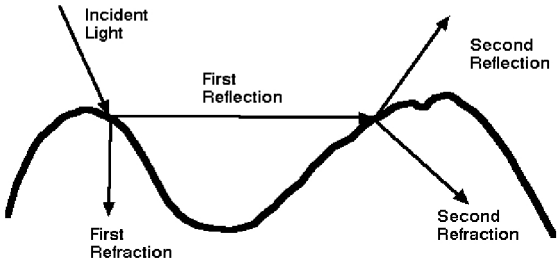


Figure 2: Illustration of multiple reflections and transmission through the air-water interface.

3 Mathematics, Physics, and Experiments on the Motion for the Surface

In this section we take a look at the mathematical problem we are trying to solve. We simplify the mathematics considerably by applying a series of approximations. How do we know these approximations are any good? There are decades of oceanographic research in which the ocean surface motion has been characterized

by measurements, simulations, mathematical analysis, and experimentation. The approximations we apply in this section are not perfect, and there are many circumstances in the real world in which they break down. But they work extraordinarily well for most conditions at sea. To give you some idea of how well they work, we show some experimental work in section 4 that has been done which clearly shows these approximations at work in the real world.

3.1 Bernoulli's Equation

The starting point of the mathematical formulation of ocean surface motion is the incompressible Navier-Stokes equations. Chapter 2 of [14] provides a thorough derivation of Bernoulli's equation from Navier-Stokes. We provide here the short version, and refer you to Kinsman's or some other textbook for details.

The incompressible Navier-Stokes equations for the velocity $\mathbf{u}(\mathbf{x}, t)$ of a fluid a any position $\mathbf{x}$ at any time t are

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla\, \mathbf{u} = -\nabla p + \mathbf{F} \quad (9)$$

$$\nabla \cdot \mathbf{u}(\mathbf{x}, t) = 0 \quad (10)$$

In these equations, $p(\mathbf{x}, t)$ is the pressure on the fluid, and $\mathbf{F}(\mathbf{x}, t)$ is the force applied to the fluid. In our application, the force is conservative, so $\mathbf{F} = -\nabla U$ for some potential energy function $U(\mathbf{x}, t)$.

For ocean surface dynamics due to conservative forces like gravity, it turns out to be worthwhile to restrict the type of motion to a class called *potential flow*. This is a situation in which the velocity has the form of a gradient:

$$\mathbf{u}(\mathbf{x}, t) = \nabla\, \phi(\mathbf{x}, t) \quad (11)$$

This restriction has the important effect of reducing the number degrees of freedom of the flow from the three components of velocity $\mathbf{u}$ to the single one of the *potential velocity* function ϕ. In fact, using this gradient form, the four Navier-Stokes equations transform into the two equations

$$\frac{\partial \phi}{\partial t} + \frac{1}{2}\left(\nabla \phi\right)^2 = -p - U \quad (12)$$

$$\nabla^2\, \phi(\mathbf{x}, t) = 0 \quad (13)$$

Equation 12 is called Bernoulli's equation. As a fully nonlinear reduction of the Navier-Stokes equations, Bernoulli's equation is capable of simulating a variety of surface dynamics effects, including wave breaking in shoaling shallow water (the bottom rises from deep water up to a beach). For more detail on numerical simulations of Bernoulli's equation in 3D, see [23].

3.2 Linearization

For our purposes, we want to reduce the complexity of Bernoulli's equation even further by applying two restrictions: linearize the equations of motion, and limit evaluation of the equations to just points on the surface itself, ignoring the volume below the surface. This may seem like an extreme restriction, but when combined with some phenomenolical knowledge of the ocean, this restrictions work very well.

The first restriction is to linearize Bernoulli's equation. This is simply the task of removing the quadratic term $1/2(\nabla\phi)^2$. Eliminating this term means that we are most likely restricted to surface waves that are not extremely violent in their motion, at least in principle. So Bernoulli's equation is reduced further to

$$\frac{\partial \phi}{\partial t} = -p - U \quad (14)$$

All of the quantities ϕ, p, and U are still evaluated at 3D points $\mathbf{x}$ on the surface and in the water volume.

The second restriction is to evaluation quantities only on the water surface. To do this we have to first characterize what we mean by the surface. We will take the surface to be a dynamically changing height field, $h(\mathbf{x}_\perp, t)$, that is a function of only the horizontal position $\mathbf{x}_\perp$ and time t. For convenience, we define the mean height of the wate surface as the zero value of the height. With this definition of wave height, the gravity-induced potential energy term U is

$$U = g\, h \quad (15)$$

and g is the gravity constant, usually $9.8\ m/sec^2$ in metric units.

Restricting to just the water surface has several important consequences. One of the first consequences is for mass conservation. In the incompressible Navier-Stokes equation, mass is conserved via the mass flux equation

$$\nabla \cdot \mathbf{u}(\mathbf{x}, t) = 0 \quad (16)$$

When we chose to consider only potential flow, this mass conservation equation became

$$\nabla^2\, \phi(\mathbf{x}, t) = 0 \quad (17)$$

If we label the horizontal portion of the position vector as $\mathbf{x}_\perp$, so that $\mathbf{x} = (\mathbf{x}_\perp, y)$, and y is the coordinate pointing down into the water volume, then the mass conservation equation restricted to the surface looks in more detail like

$$\left\{ \nabla_\perp^2 + \frac{\partial^2}{\partial y^2} \right\} \phi(\mathbf{x}_\perp, t) = 0 \quad (18)$$

Now, when you look at this equation and see that ϕ now depends only on the $\mathbf{x}_\perp$ on the surface, you might be tempted to throw out the $\partial^2/\partial y^2$ part of the equation, because there does not appear to be a dependence. That would produce useless results. Instead, what works better in this odd world of partial differential equations is to allow ϕ to be an arbitrary function (at least with respect to this mass conservation equation) and to define the y-derivative operator to be

$$\frac{\partial}{\partial y} = \pm\sqrt{-\nabla_\perp^2} \quad (19)$$

so that the operator ∇^2 is zero. We will use this approach for any quantity evaluated on the water surface whenever we need a vertical derivative. Of course, this introduces an unusual operator that contains a square root function.

Another consequence of restricting ourselves to just the surface is that the pressure remains essentially constant, and we can choose to have that constant be 0. With this and the rest of the restrictions, Bernoulli's equation has been linearized to

$$\frac{\partial \phi(\mathbf{x}_\perp, t)}{\partial t} = -g\, h(\mathbf{x}_\perp, t) \quad (20)$$

There is one final equation that must be rewritten for this situation. Recall that the velocity potential ϕ is used to compute the 3D fluid velocity as a gradient, $\mathbf{u} = \nabla\phi$. The vertical component of the velocity must now use equation 19. In addition, the vertical velocity of the fluid is the same at the speed of the surface height. Combining these we get

$$\frac{\partial h(\mathbf{x}_\perp, t)}{\partial t} = \sqrt{-\nabla_\perp^2}\, \phi(\mathbf{x}_\perp, t) \quad (21)$$

These last two equations, 20 and 21, are the final equations of motion that are needed to solve for the surface motion. They can

also be converted into a single equation. For example, if we take a derivative with respect to time of equation 21, and use 20 to substitute for the time derivative of the velocity potential, we get the single equation for the evolution of the surface height.

$$\frac{\partial^2 h(\mathbf{x}_\perp, t)}{\partial t^2} = -g\sqrt{-\nabla_\perp^2}\; h(\mathbf{x}_\perp, t) \tag{22}$$

This still involves the unusual operator $\sqrt{-\nabla_\perp^2}$. However, taking two more time derivatives converts it to a more normal two-dimensional Laplacian for the equation

$$\frac{\partial^4 h(\mathbf{x}_\perp, t)}{\partial t^4} = g^2\nabla_\perp^2\; h(\mathbf{x}_\perp, t) \tag{23}$$

This form is frequently the starting point for building mathematical solutions to the surface wave equation.

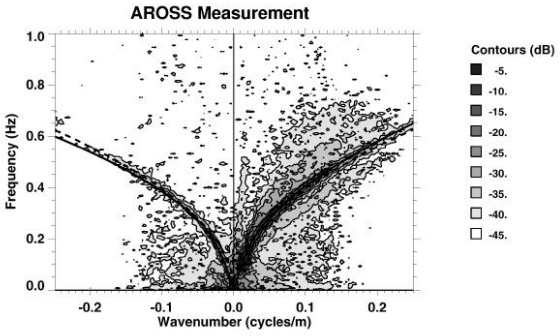


Figure 3: A slice through a 3D PSD showing that the observed wave energy follows the deep water dispersion relation very well.

3.3 Dispersion

It turns out that the equations we have built for surface height – whether in the form of equations 20 and 21, or equation 22, or equation23 – reduce to one primary lesson about surface wave propagation. This lesson is embodied in a simple mathematical relationship called the *Dispersion Relation*, which is the focus of this section. Our goal here is to obtain that simple expression from the mathematics above, understand some of its meaning, and demonstrate that, even though it appears unrealistically simplistic, the Dispersion Relation is in fact present in natural ocean waves and can be measured experimentally.

Lets just use the version of the surface evolution equation in equation 23 for convenience. The other two versions could be used and arrive at the same answer using a slightly different set of manipulations. Note that the equation of motion for the surface height is linear in the surface height. So as with any linear differential equation, the general solution of the equation is obtained by adding up any number of specific solutions. So lets find a specific solution. It turns out that all specific solutions have the form

$$h(\mathbf{x}_\perp, t) = h_0 \exp\{i\mathbf{k}\cdot\mathbf{x}_\perp - i\omega t\} \tag{24}$$

The 2D vector $\mathbf{k}$ and the numbers ω and h_0 are generic parameters at this point. If we use this form of a solution in equation 23, it turns into an algebraic equation like this:

$$h_0\;\left\{\omega^4 - g^2k^2\right\} = 0 \tag{25}$$

(and k is the magnitude of the vector $\mathbf{k}$). For this solution, there are only two possibilities:

1. $h_0 = 0$. Then the surface height is flat, and the solution is not very interesting.

2. $\omega = \pm\sqrt{gk}$ and h_0 can be anything. This is the interesting solution.

What we have found here is that the entire Navier-Stokes fluid dynamics problem, reduced to an evolution equation for the water surface and approximated to something that can be solved relatively easily, amounts to a single equation imposing the constraint that the temporal frequency ω of surface height movements is connected to the spatial extent of a the propagating wave $k = |\mathbf{k}|$. This relationship, $\omega = \pm\sqrt{gk}$ is the *Dispersion Relation* mentioned earlier.

Note in particular that there is no constraint placed on the amplitude h_0. But if the Navier-Stokes equation does not have anything to say about the amplitude, how do we give it a value? One way is by imposing initial conditions on the height and on its vertical speed. For ocean surface simulation in the next section, we will use an alternate method, a statistical procedure to generate random realizations of the amplitude, guided by measurements of the variance properties of wave height on the open ocean.

So the question remaining is just how reasonable is the dispersion relation for modelling realistic ocean surface waves? This is where lots of experimental research can come into play. Although there have been many decades of research on ocean wave properties using devices placed in the water to directly measure the wave motion at a point, here we look at some relatively new research that involves measuring wave properties remotely with a camera in a plane.

The AROSS [24] is a panchromatic camera mounted in a special hosing on the nose of a small airplane. Attached to the camera is navigation and GPS instrumentation which allow the camera position, viewing direction, and orientation to be measured for each frame. After the plane flys a circular orbit around a spot over the ocean, this data can be used to remap images of the ocean into a common reference frame, so that the motion of the aircraft has been removed (except for lighting variations). This remapping allows the researchers to use many frames of ocean imagery, typically 1-2 minutes worth, in some data processing to look for the dispersion relation.

The data processing that AROSS imagery is subjected to generates something called a 3D Power Spectral Density (PSD). This is obtained by taking the Fourier Transform of a time series of imagery in time, as well as Fourier Transforms in the two spatial directions of images. The output of these 3 Fourier Transforms is a quantity that is closely related to the amplitudes $h_0(\mathbf{k}, \omega)$ for each spatial and temporal frequency. These are then absolute squared and smoothed or averaged in some way so that the output is a numerical approximation of a statistical average of $|h_0(\mathbf{k}, \omega)|^2$.

But how does a 3D PSD help us decide whether the dispersion relation appears in nature? If the imagery found only dispersion constrained surface waves, then the 3D PSD should have the value 0 for all values of $\mathbf{k}$, ω that do not satisfy the dispersion relation. So mostly we would expect the 3D PSD to only have significant values in a narrow set of $\mathbf{k}$, ω values.

Figure 3 show a plot of the 3D PSD generated from AROSS images [24]. From the 3D volumetric PSD, this plot figure is a plane sliced through the volume. When sliced like this, the dispersion relation is a curve on the slice, shown as two dotted curves. The data is plotted as contours of PSD intensity, color-coded by the key on the right. PSD levels following the dispersion relation curves are much higher than in other regions. This shows that the motion of the surface waves on all scales includes a very strong dispersion relation style of motion. There are other types of motion certainly, which the PSD figure shows as intensity levels away from the dis-

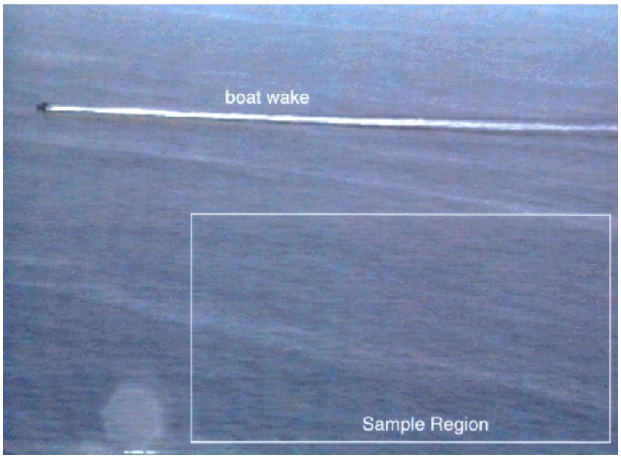


Figure 4: Site at which video data was collected in 1986, near Zuma Beach, California.

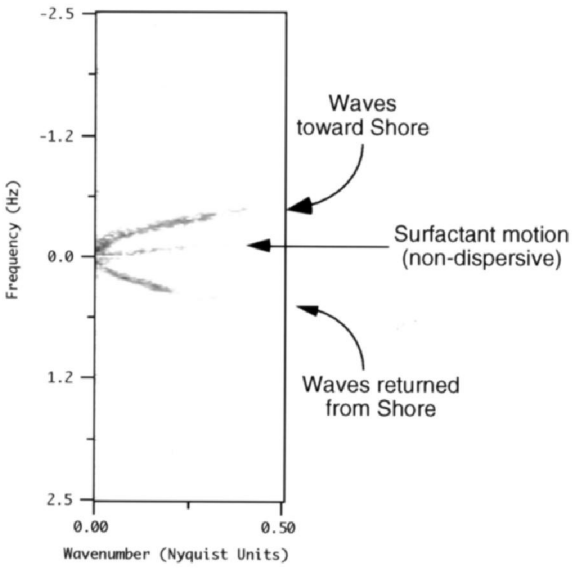


Figure 5: Slice from a 3D Power Spectral Density grayscale plot, from processed video data.

persion curves. But the dispersion motion is the strongest feature of this data.

Relatively simple experiments can be done by anyone with access to a video camera and a hilltop overlook of an ocean. For example, figure 4 is a frame from a video segment showing water coming into the beach near Zuma Beach, California. The video camera was located on hill overlooking the beach, in 1986. In 1993, the region of video frames indicated in the figure was digitized, to produce a time series of frames containing just water surface.

Figure 5 shows the actual 3D PSD from the image data. There are two clear branches along the dispersion relationship we have discussed, with no apparent modification by shallow water affects. There is also a third branch that is approximately a straight line lying between the first two. Examination of the video shows that this branch comes from a surfactant layer floating on the water in part of the video frame, and moving with a constant speed. Excluding the surface layer, this data clearly demonstrates the validity of the dispersion relationship, and demonstrates the usefulness of the linearized model of surface waves.

4 Practical Ocean Wave Algorithms

In this section we focus on algorithms and practical steps to building height fields for ocean waves. Although we will be occupied mostly by a method based on Fast Fourier Transforms (FFTs), we begin by introducing a simpler description called Gerstner Waves. This is a good starting point for several reasons: the mathematics is relatively light compared to FFTs, several important oceanographic concepts can be introduced, and they give us a chance to discuss wave animation. After this discussion of Gerstner waves, we go after the more complex FFT method, which produces wave height fields that are more realistic. These waves, called "linear waves" or "gravity waves" are a fairly realistic representation of typical waves on the ocean when the weather is not too stormy. Linear waves are certainly not the whole story, and so we discuss also some methods by which oceanographers expand the description to "nonlinear waves", waves passing over a shallow bottom, and very tiny waves about one millimeter across called capillary waves.

In the course of this discussion, we will see how quantities like windspeed, surface tension, and gravitational acceleration come into the practical implementation of the algorithms.

4.1 Gerstner Waves

Gerstner waves were first found as an approximate solution to the fluid dynamic equations almost 200 years ago. There first application in computer graphics seems to be the work by Fournier and Reeves in 1986 (cited previously). The physical model is to describe the surface in terms of the motion of individual points on the surface. To a good approximation, points on the surface of the water go through a circular motion as a wave passes by. If a point on the undisturbed surface is labelled $\mathbf{x}_0 = (x_0, z_0)$ and the undisturbed height is $y_0 = 0$, then as a single wave with amplitude A passes by, the point on the surface is displaced at time t to

$$\mathbf{x} = \mathbf{x}_0 - (\mathbf{k}/k)A\sin(\mathbf{k}\cdot\mathbf{x}_0 - \omega t) \quad (26)$$

$$y = A\cos(\mathbf{k}\cdot\mathbf{x}_0 - \omega t)\,. \quad (27)$$

In these expressions, the vector $\mathbf{k}$, called the wavevector, is a horizontal vector that points in the direction of travel of the wave, and has magnitude k related to the length of the wave (λ) by

$$k = 2\pi/\lambda \quad (28)$$

The frequency w is related to the wavevector, as discussed later.

Figure 6 shows two example wave profiles, each with a different value of the dimensionless amplitude kA. For values $kA < 1$, the wave is periodic and shows a steepening at the tops of the waves as kA approaches 1. For $kA > 1$, a loop forms at the tops of the wave, and the "insides of the wave surface are outside", not a particularly desirable or realistic effect.

As presented so far, Gerstner waves are rather limited because they are a single sine wave horizontally and vertically. However, this can be generalized to a more complex profile by summing a set of sine waves. One picks a set of wavevectors $\mathbf{k}_i$, amplitudes A_i, frequencies ω_i, and phases ϕ_i, for $i = 1, \ldots, N$, to get the expressions

$$\mathbf{x} = \mathbf{x}_0 - \sum_{i=1}^{N}(\mathbf{k}_i/k_i)A_i\sin(\mathbf{k}_i\cdot\mathbf{x}_0 - \omega_i t + \phi_i) \quad (29)$$

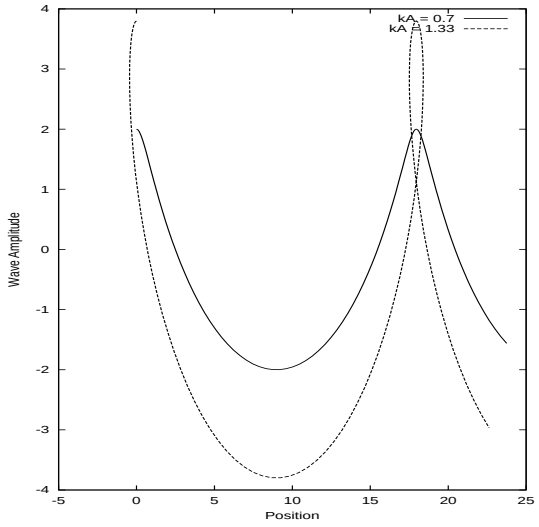


Figure 6: Profiles of two single-mode Gerstner waves, with different relative amplitudes and wavelengths.

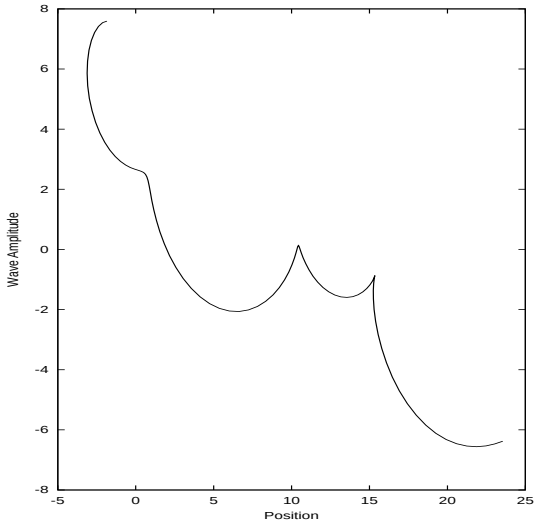


Figure 7: Profile of a 3-mode Gerstner wave.

$$y \quad = \quad \sum_{i=1}^{N} A_i \cos(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i) \; . \tag{30}$$

Figure 7 shows an example with three waves in the set. Interesting and complex shapes can be obtained in this way.

4.2 Animating Waves: The Dispersion Relation

The animated behavior of Gerstner waves is determined by the set of frequencies ω_i chosen for each component. For water waves, there is a well-known relationship between these frequencies and the magnitude of their corresponding wavevectors, k_i. In deep water, where the bottom may be ignored, that relationship is

$$\omega^2(k) = gk \; . \tag{31}$$

The parameter g is the gravitational constant, nominally $9.8m/sec^2$. This dispersion relationship holds for Gerstner waves, and also for the FFT-based waves introduced next.

There are several conditions in which the dispersion relationship is modified. When the bottom is relatively shallow compared to the length of the waves, the bottom has a retarding affect on the waves. For a bottom at a depth D below the mean water level, the dispersion relation is

$$\omega^2(k) = gk \tanh(kD) \tag{32}$$

Notice that if the bottom is very deep, the behavior of the $\tanh$ function reduces this dispersion relation to the previous one.

A second situation which modifies the dispersion relation is surface tension. Very small waves, with a wavelength of about 1 cm or less, have an additional term:

$$\omega^2(k) = gk(1 + k^2 L^2) \; , \tag{33}$$

and the parameter L has units of length. Its magnitude is the scale for the surface tension to have effect.

Using these dispersion relationships, it is very difficult to create a sequence of frames of water surface which for a continuous loop. In order to have the sequence repeat after a certain amount of time T for example, it is necessary that all frequencies be multiples of the basic frequence

$$\omega_0 \equiv \frac{2\pi}{T} \; . \tag{34}$$

However, when the wavevectors $\mathbf{k}$ are distributed on a regular lattice, itis impossible to arrange the dispersion-generated frequencies to also be on a uniform lattce with spacing ω_0.

The solution to that is to not use the dispersion frequences, but instead a set that is close to them. For a given wavenumber k, we use the frequency

$$\bar{\omega}(k) = \left[\left[\frac{\omega(k)}{\omega_0} \right] \right] \omega_0 \; , \tag{35}$$

where $[[a]]$ means take the integer part of the value of a, and $\omega(k)$ is any dispersion relationship of interest. The frequencies $\bar{\omega}(k)$ are a *quantization* of the dispersion surface, and the animation of the water surface loops after a time T because the quantized frequencies are all integer multiples of ω_0. Figure 8 plots the original dispersion curve, along with quantized dispersion curves for two choices of the repeat time T.

4.3 Statistical Wave Models and the Fourier Transform

Oceanographic literature tends to downplay Gerstner waves as a realistic model of the ocean. Instead, statistical models are used, in combination with experimental observations. In the statistical models, the wave height is considered a random variable of horizontal position and time, $h(\mathbf{x}, t)$.

Statistical models are also based on the ability to decompose the wave height field as a sum of sine and cosine waves. The value of this decomposition is that the amplitudes of the waves have nice mathematical and statistical properties, making it simpler to build models. Computationally, the decomposition uses Fast Fourier Transforms (ffts), which are a rapid method of evaluating the sums.

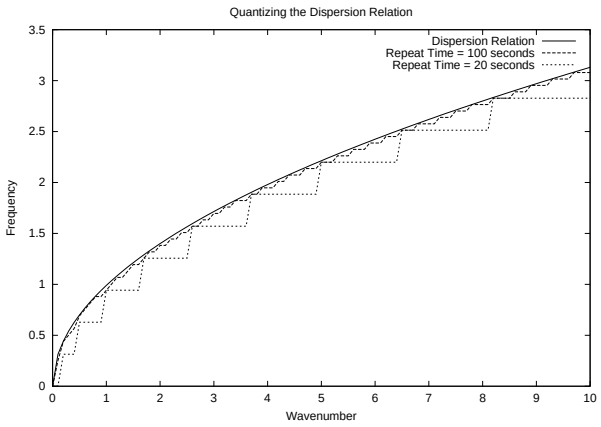


Figure 8: A comparison of the continuous dispersion curve $\omega = \sqrt{gk}$ and quantized dispersion curves, for repeat times of 20 seconds and 100 seconds. Note that for a longer repeat time, the quantized is a closer approximation to the original curve.

The fft-based representation of a wave height field expresses the wave height $h(\mathbf{x}, t)$ at the horizontal position $\mathbf{x} = (x, z)$ as the sum of sinusoids with complex, time-dependent amplitudes:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t)\ \exp(i\mathbf{k} \cdot \mathbf{x}) \tag{36}$$

where t is the time and $\mathbf{k}$ is a two-dimensional vector with components $\mathbf{k} = (k_x, k_z)$, $k_x = 2\pi n/L_x$, $k_z = 2\pi m/L_z$, and n and m are integers with bounds $-N/2 \le n < N/2$ and $-M/2 \le m < M/2$. The fft process generates the height field at discrete points $\mathbf{x} = (nL_x/N, mL_z/M)$. The value at other points can also be obtained by switching to a *discrete* fourier transform, but under many circumstances this is unnecessary and is not applied here. The height amplitude Fourier components, $\tilde{h}(\mathbf{k}, t)$, determine the structure of the surface. The remainder of this subsection is concerned with generating random sets of amplitudes in a way that is consistent with oceanographic phenomenology.

For computer graphics purposes, the slope vector of the wave-height field is also needed in order to find the surface normal, angles of incidence, and other aspects of optical modeling as well. One way to compute the slope is though a finite difference between fft grid points, separated horizontally by some 2D vector $\Delta\mathbf{x}$. While a finite difference is efficient in terms of memory requirements, it can be a poor approximation to the slope of waves with small wavelength. An exact computation of the slope vector can be obtained by using more ffts:

$$\epsilon(\mathbf{x}, t) = \nabla h(\mathbf{x}, t) = \sum_{\mathbf{k}} i\mathbf{k}\, \tilde{h}(\mathbf{k}, t)\ \exp(i\mathbf{k} \cdot \mathbf{x})\ . \tag{37}$$

In terms of this fft representation, the finite difference approach would replace the term $i\mathbf{k}$ with terms proportional to

$$\exp(i\mathbf{k} \cdot \Delta\mathbf{x}) - 1 \tag{38}$$

which, for small wavelength waves, does not well approximate the gradient of the wave height. Whenever possible, slope computation via the fft in equation 37 is the prefered method.

The fft representation produces waves on a patch with horizontal dimensions $L_x \times L_z$, outside of which the surface is perfectly periodic. In practical applications, patch sizes vary from 10 meters to 2 kilometers on a side, with the number of discrete sample points as high as 2048 in each direction (i.e. grids that are 2048×2048, or over 4 million waves). The patch can be tiled seamlessly as desired over an area. The consequence of such a tiled extension, however, is that an artificial periodicity in the wave field is present. As long as the patch size is large compared to the field of view, this periodicity is unnoticeable. Also, if the camera is near the surface so that the effective horizon is one or two patch lengths away, the periodicity will not be noticeable in the look-direction, but it may be apparent as repeated structures across the field of view.

Oceanographic research has demonstrated that equation 36 is a reasonable representation of naturally occurring wind-waves in the open ocean. Statistical analysis of a number of wave-buoy, photographic, and radar measurements of the ocean surface demonstrates that the wave height amplitudes $\tilde{h}(\mathbf{k}, t)$ are nearly statistically stationary, independent, gaussian fluctuations with a spatial spectrum denoted by

$$P_h(\mathbf{k}) = \left\langle \left| \tilde{h}^*(\mathbf{k}, t) \right|^2 \right\rangle \tag{39}$$

for data-estimated ensemble averages denoted by the brackets $\langle\ \rangle$.

There are several analytical semi-empirical models for the wave spectrum $P_h(\mathbf{k})$. A useful model for wind-driven waves larger than capillary waves in a fully developed sea is the *Phillips* spectrum

$$P_h(\mathbf{k}) = A \frac{\exp\left(-1/(kL)^2\right)}{k^4} |\hat{\mathbf{k}} \cdot \hat{w}|^2\ , \tag{40}$$

where $L = V^2/g$ is the largest possible waves arising from a continuous wind of speed V, g is the gravitational constant, and $\hat{w}$ is the direction of the wind. A is a numeric constant. The cosine factor $|\hat{\mathbf{k}} \cdot \hat{w}|^2$ in the Phillips spectrum eliminates waves that move perpendicular to the wind direction. This model, while relatively simple, has poor convergence properties at high values of the wavenumber $|\mathbf{k}|$. A simple fix is to suppress waves smaller that a small length $\ell \ll L$, and modify the Phillips spectrum by the multiplicative factor

$$\exp\left(-k^2\ell^2\right)\ . \tag{41}$$

Of course, you are free to "roll your own" spectrum to try out various effects.

4.4 Building a Random Ocean Wave Height Field

Realizations of water wave height fields are created from the principles elaborated up to this point: gaussian random numbers with spatial spectra of a prescribed form. This is most efficiently accomplished directly in the fourier domain. The fourier amplitudes of a wave height field can be produced as

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_h(\mathbf{k})}\ , \tag{42}$$

where ξ_r and ξ_i are ordinary independent draws from a gaussian random number generator, with mean 0 and standard deviation 1. Gaussian distributed random numbers tend to follow the experimental data on ocean waves, but of course other random number distributions could be used. For example, log-normal distributions could be used to produce height fields that are vary "intermittent", i.e. the waves are very high or nearly flat, with relatively little in between.

Given a dispersion relation $\omega(k)$, the Fourier amplitudes of the wave field realization at time t are

$$\begin{aligned} \tilde{h}(\mathbf{k}, t) &= \tilde{h}_0(\mathbf{k}) \exp\{i\omega(k)t\} \\ &+ \tilde{h}_0^*(-\mathbf{k}) \exp\{-i\omega(k)t\} \end{aligned} \tag{43}$$

This form preserves the complex conjugation property $\tilde{h}^*(\mathbf{k}, t) = \tilde{h}(-\mathbf{k}, t)$ by propagating waves "to the left" and "to the right". In addition to being simple to implement, this expression is also efficient for computing $h(\mathbf{x}, t)$, since it relies on ffts, and because the wave field at any chosen time can be computed without computing the field at any other time.

In practice, how big does the Fourier grid need to be? What range of scales is reasonable to choose? If you want to generate wave heights faster, what do you do? Lets take a look at these questions.

How big should the Fourier grid be? The values of N and M can be between 16 and 2048, in powers of two. For many situations, values in the range 128 to 512 are sufficient. For extremely detailed surfaces, 1024 and 2048 can be used. For example, the wave fields used in the motion pictures *Waterworld* and *Titanic* were 2048×2048 in size, with the spacing between grid points at about 3 cm. Above a value of 2048, one should be careful because the limits of numerical accuracy for floating point calculations can become noticeable.

What range of scales is reasonable to choose? The answer to this question comes down to choosing values for L_x, L_z, M, and N. The smallest facet in either direction is $dx \equiv L_x/M$ or $dz \equiv L_z/N$. Generally, dx and dz need never go below 2 cm or so. Below this scale, the amount of wave action is small compared to the rest of the waves. Also, the physics of wave behavior below 2 cm begins to take on a very different character, involving surface tension and "nonlinear" processes. From the form of the spectrum, waves with a wavelength larger than V^2/g are suppressed. So make sure that dx and dz are smaller than V^2/g by a substantial amount (10 - 1000) or most of the interesting waves will be lost. The secret to realistic looking waves (e.g. figure 12 (a) compared to figure 12 (c)) is to have M and N as large as reasonable.

How do you generate wave height fields in the fastest time? The time consuming part of the computation is the fast fourier transform. Running on a 1+ GHz cpu, 512×512 FFTs can be generated at nearly interactive rates.

4.5 Examples: Height Fields and Renderings

We now turn to some examples of waves created using the fft approach discussed above. We will show waves in two formats: as greyscale images in which the grey level is proportional to wave height; and renderings of oceanscapes using several different rendering packages to illustrate what is possible.

In the first set of examples, the grid size is set to $M = N = 512$, with $L_x = L_z = 1000$ meters. The wind speed is a gale force at $V = 31$ meters/second, moving in the x-direction. The small-wave cutoff of $\ell = 1$ meter was also used. Figure 9 is a greyscale representation of the wave height: brighter means higher and darker means lower height. Although produced by the fft algorithms described here, figure 9 is not obviously a water height field. It may help to examine figure 10, which is a greyscale depiction of the x-component of the slope. This looks more like water waves that figure 9. What is going on?

Figures 9 and 10 demonstrate a consequence of water surface optics, discussed in the next section: the visible qualities of the surface structure tend to be strongly influenced by the slope of the waves. We will discuss this in quantitative detail, but for now we willl summarize it by saying that the reflectivity of the water is a strong function of the slope of the waves, as well as the directions of the light(s) and camera.

To illustrate a simple effect of customizing the spectrum model, figure 11 is the greyscale display of a height field identical to figure

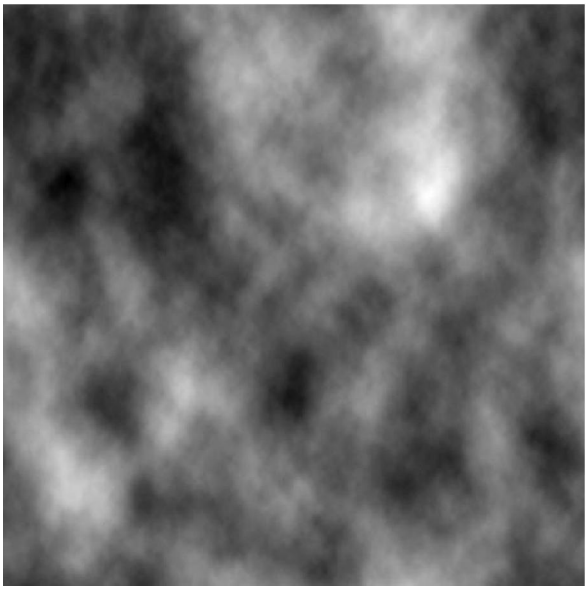


Figure 9: A surface wave height realization, displayed in greyscale.

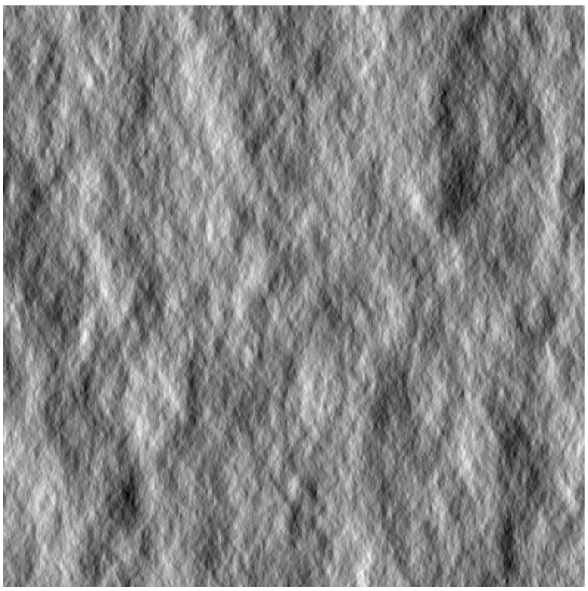


Figure 10: The x-component of the slope for the wave height realization in figure 9.

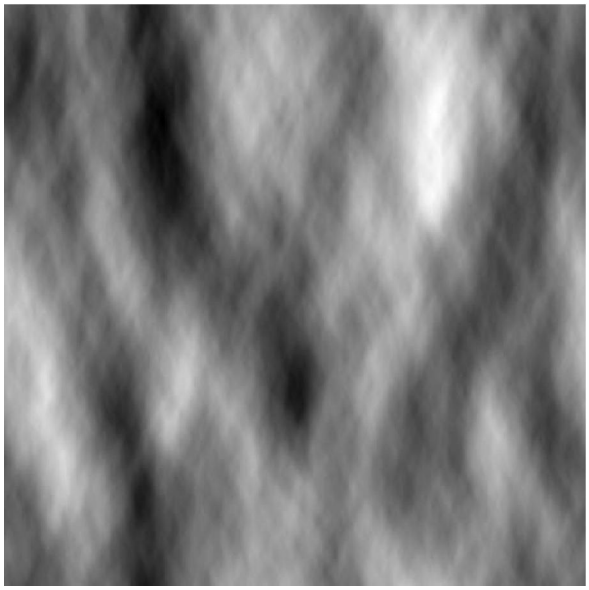


Figure 11: Wave height realization with increased directional dependence.

9, with the exception that the directional factor $|\hat{\mathbf{k}} \cdot \hat{w}|^2$ in equation 40 has been changed to $|\hat{\mathbf{k}} \cdot \hat{w}|^6$. The surface is clearly more aligned with the direction of the wind.

The next example of a height field uses a relatively simple shader in BMRT, the Renderman-compliant raytracer. The shader is shown in the next section. Figure 12 shows three renderings of water surfaces, varying the size of the grid numbers M and N and making the facet sizes dx and dz proportional to $1/M$ and $1/N$. So as we go from the top image to the bottom, the facet sizes become smaller, and we see the effect of increasing amount of detail in the renderings. Clearly, more wave detail helps to build a realistic-looking surface.

As a final example, figure 13 is an image rendered in the commercial package RenderWorld by Arete Entertainment. This rendering includes the effect of an atmosphere, and water volume scattered light. These are discussed in the next section. But clearly, wave height fields generated from random numbers using an fft prescription can produce some nice images.

4.6 Choppy Waves

We turn briefly in this section to the subject of creating choppy looking waves. The waves produced by the fft methods presented up to this point have rounded peaks and troughs that give them the appearance of fair-weather conditions. Even in fairly good weather, and particularly in a good wind or storm, the waves are sharply peaked at their tops, and flattened at the bottoms. The extent of this chopping of the wave profile depends on the environmental conditions, the wavelengths and heights of the waves. Waves that are sufficiently high (e.g. with a slope greater than about 1/6) eventually break at the top, generating a new set of physical phenonema in foam, splash, bubbles, and spray.

The starting point for this method is the fundamental fluid dynamic equations of motion for the surface. These equations are expressed in terms of two dynamical fields: the surface elevation and the velocity potential on the surface, and derive from the Navier-Stokes description of the fluid throughout the volume of the water and air, including both above and below the interface. Creamer

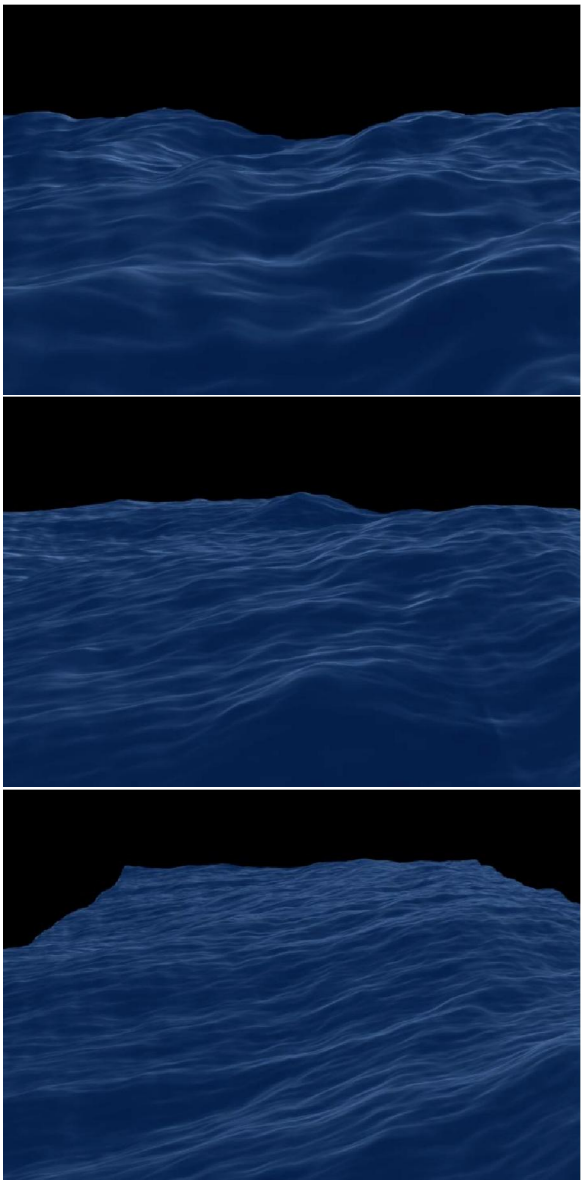


Figure 12: Rendering of waves with (top) a fairly low number of waves (facet size = 10 cm), with little detail; (middle) a reasonably good number of waves (facet size = 5 cm); (bottom) a high number of waves with the most detail (facet size = 2.5 cm).

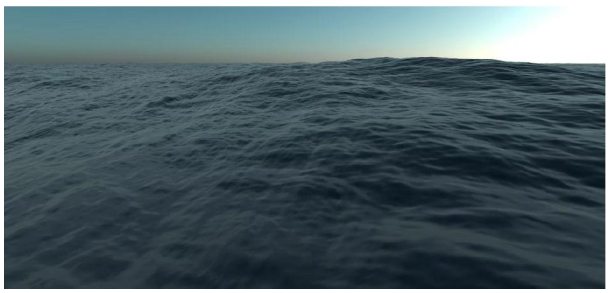


Figure 13: An image of a wave height field rendered in a commercial package with a model atmosphere and sophisticated shading.

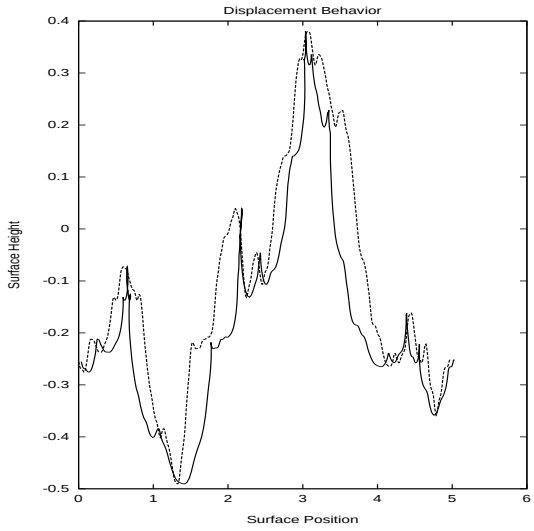


Figure 14: A comparison of a wave height profile with and without the displacement. The dashed curve is the wave height produced by the fft representation. The solid curve is the height field displaced using equation 44.

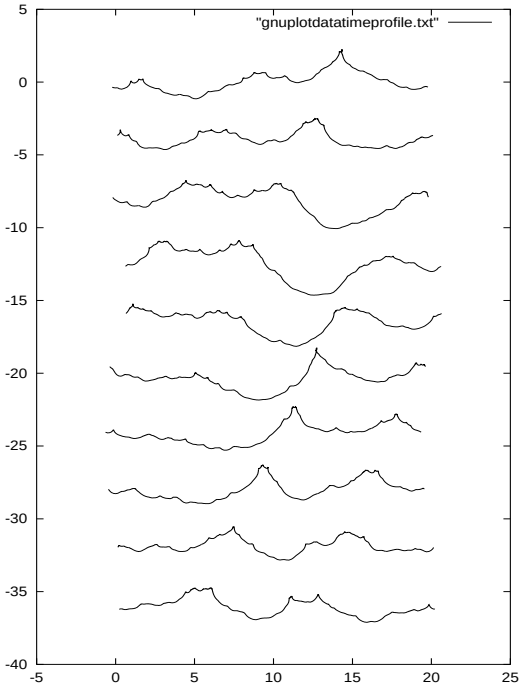


Figure 15: A time sequence of profiles of a wave surface. From top to bottom, the time between profiles is 0.5 seconds.

et al[16] set out to apply a mathematical approach called the "Lie Transform technique" to generate a sequence of "canonical transformations" of the elevation and velocity potential. The benefit of this complex mathematical procedure is to convert the elevation and velocity potential into new dynamical fields that have a simpler dynamics. The transformed case is in fact just the simple ocean height field we have been discussing, including evolution with the same dispersion relation we have been using in this paper. Starting from there, the inverse Lie Transform in principle converts our phenomenological solution into a dynamically more accurate one. However, the Lie Transform is difficult to manipulate in 3 dimensions, while in two dimensions exact results have been obtained. Based on those exact results in two dimensions, an extrapolation for the form of the 3D solution has been proposed: a horizontal displacement of the waves, with the displacement locally varying with the waves.

In the fft representation, the 2D displacement vector field is computed using the Fourier amplitudes of the height field, as

$$\mathbf{D}(\mathbf{x},t) = \sum_{\mathbf{k}} -i\frac{\mathbf{k}}{k}\,\tilde{h}(\mathbf{k},t)\,\exp\left(i\mathbf{k}\cdot\mathbf{x}\right) \tag{44}$$

Using this vector field, the horizontal position of a grid point of the surface is now $\mathbf{x} + \lambda\mathbf{D}(\mathbf{x},t)$, with height $h(\mathbf{x})$ as before. The parameter λ is not part of the original conjecture, but is a convenient method of scaling the importance of the displacement vector. This conjectured solution does not alter the wave heights directly, but instead warps the horizontal positions of the surface points in a way that depends on the spatial structure of the height field. The particular form of this warping however, actually sharpens peaks in the height field and broadens valleys, which is the kind of nonlinear behavior that should make the fft representation more realistic. Figure 14 shows a profile of the wave height along one direction in a simulated surface. This clearly shows that the "displacement conjecture" can dramatically alter the surface.

The displacment form of the this solution is similar to the algorithm for building Gerstner waves [12] discussed in section 4. In that case however, the displacement behavior, applied to sinusoid shapes, was the principle method of characterizing the water surface structure, and here it is a modifier to an already useful wave height representation.

Figure 15 illustrates how these choppy waves behave as they evolve. The tops of waves form a sharp cusp, which rounds out and disappears shortly afterward.

One "problem" with this method of generating choppy waves can be seen in figure 14. Near the tops of some of the waves, the surface actally passes through itself and inverts, so that the outward normal to the surface points inward. This is because the amplitudes of the wave components can be large enough to create large displacements that overlap. This is easily defeated simply by reducing the magnitude of the scaling factor λ. For the purposes of computer graphics, this might actually be a useful effect to signal the production of spray, foam and/or breaking waves. We will not discuss here how to carry out such an extension, except to note that in order to use this region of overlap, a simple and quick test is needed for deciding that the effect is taking place. Fortunately, there is such a simple test in the form of the Jacobian of the transformation from $\mathbf{x}$ to $\mathbf{x} + \lambda\mathbf{D}(\mathbf{x},t)$. The Jacobian is a measure of the uniqueness of the transformation. When the displacement is zero, the Jacobian is 1. When there is displacement, the Jacobian has the form

$$J(\mathbf{x}) = J_{xx}J_{yy} - J_{xy}J_{yx}\ , \tag{45}$$

with individual terms

$$\begin{aligned} J_{xx}(\mathbf{x}) &= 1 + \lambda\frac{\partial D_x(\mathbf{x})}{\partial x} \\ J_{yy}(\mathbf{x}) &= 1 + \lambda\frac{\partial D_y(\mathbf{x})}{\partial y} \end{aligned}$$

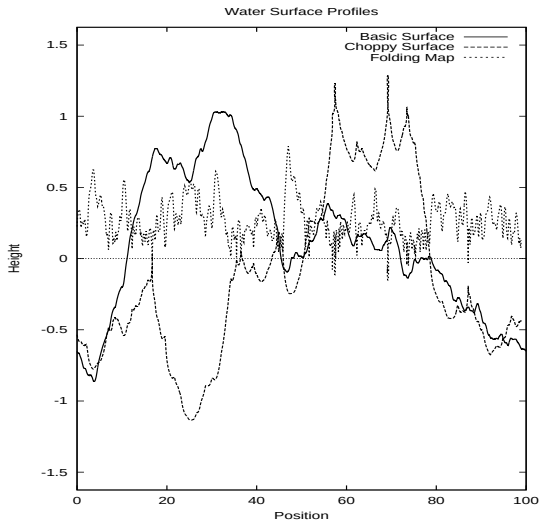


Figure 16: Wave height profile with and without the displacement. Also plotted is the Jacobian map for choppy wave profile.

$$\begin{aligned} J_{yx}(\mathbf{x}) &= \lambda \frac{\partial D_y(\mathbf{x})}{\partial x} \\ J_{xy}(\mathbf{x}) &= \lambda \frac{\partial D_x(\mathbf{x})}{\partial y} = J_{yx} \end{aligned}$$

and $\mathbf{D} = (D_x, D_y)$. The Jacobian signals the presence of the overlapping wave bacause its value is less than zero in the overlap region. For example, figure 16 plots a profile of a basic wave without displacement, the wave with displacement, and the value of J for the choppy wave (labeled "Folding Map"). The "folds" or overlaps in the choppy surface are clearly visible, and align with the regions in which $J < 0$. With this information, it should be relatively easy to extract the overlapping region and use it for other purposes if desired.

But there is more that can be learned from these folded waves from a closer examination of this folding criterion. The Jacobian derives from a 2×2 matrix which measures the local uniqueness of the choppy wave map $\mathbf{x} \rightarrow \mathbf{x} + \lambda \mathbf{D}$. This matrix can in general be written in terms of eigenvectors and eigenvalues as:

$$J_{ab} = J_- \hat{e}_a^- \hat{e}_b^- + J_+ \hat{e}_a^+ \hat{e}_b^+ \ , \ \ (a, b = x, y) \tag{46}$$

where J_- and J_+ are the two eigenvalues of the matrix, ordered so that $J_- \leq J_+$. The corresponding orthonormal eigenvectors are $\hat{e}^-$ and $\hat{e}^+$ respectively. From this expression, the Jacobian is just $J = J_- J_+$.

The criterion for folding that $J < 0$ means that $J_- < 0$ and $J_+ > 0$. So the minimum eigenvalue is the actual signal of the onset of folding. Further, the eigenvector $\hat{e}^-$ points in the horizontal direction in which the folding is taking place. So, the prescription now is to watch the minimum eigenvalue for when it becomes negative, and the alignment of the folded wave is parallel to the minimum eigenvector.

We can illustrate this phenomenon with an example. Figures 17 and 18 show two images of an ocean surface, one without choppy waves, and the other with the choppy waves strongly applied. These two surfaces are identical except for the choppy wave algorithm. Figure 19 shows the wave profiles of both surfaces along a slice through the surfaces. Finally, the profile of the choppy wave is plotted together with the value of the minimum eigenvalue in figure 20, showing the clear connection between folding and the negative value of J_-.

Incidentally, computing the eigenvalues and eigenvectors of this matrix is fast because they have analytic expressions as

$$J_\pm = \frac{1}{2}(J_{xx} + J_{yy}) \pm \frac{1}{2} \left\{ (J_{xx} - J_{yy})^2 + 4 J_{xy}^2 \right\}^{1/2} \tag{47}$$

for the eigenvalues and

$$\hat{e}^\pm = \frac{(1, q_\pm)}{\sqrt{1 + q_\pm^2}} \tag{48}$$

with

$$q_\pm = \frac{J_\pm - J_{xx}}{J_{xy}} \tag{49}$$

for the eigenvectors.

5 Interactive Waves from Disturbances

The Fourier based approach to water surface evolution described in the section 4 has several limitations that make in unworkable for really interactive applications. Fundamentally, the Fourier method computes the wave height everywhere in one FFT computation. You cannot choose to compute wave height in limited areas of a surface grid without completely altering the calculation. You either get the whole surface, or you don't compute it. This makes it very hard to customize the wave propagation problem from one location to another. So it you want to put some arbitrarily-shaped object in the water, move it around some user-constructed path, the FFT method makes it difficult to compute the wave response to the shape and movement of the object. So if you want to have an odd looking craft moving around in the water, and maybe going up and down and changing shape, the FFT method is not the easiest thing to use. Also, if you want to have a shallow bottom, with a sloping beach or underwater sea mound or some other variability in the depth of the water, the FFT method again is a challenge use.

Fortunately in the last few years an alternative scheme has emerged which allows fast construction of a water surface in response to interactions with objects in the water and/or variable bottom depth. The heart of the method is an approach to computing the propagation which does not use the FFT method at all. The fundamental mathematics of this interactive method is described in the reference [22], and is refered to as *iWave*. Here we very quickly run through the approach and show some examples in action.

At its heart the iWave method returns to the linearized equation for the wave height, 22. We can turn the time derivatives into finite time differences for a time step Δt, and get an explicit expression for the wave height:

$$\begin{aligned} h(\mathbf{x}, t + \Delta t) &= 2h(\mathbf{x}, t) - h(\mathbf{x}, t - \Delta t) \\ &- g(\Delta t)^2 \sqrt{-\nabla^2}\, h(\mathbf{x}, t) \end{aligned} \tag{50}$$

This form can be used to explicitly advance the surface wave height from one frame to the next. The hard part of course is figuring out how to calculate the last term, with the square root of the Laplacian operator.

The solution is to use convolution. The wave height is kept on a rectangular grid of points (the dimensions of which do *not* have to be powers of two), and we make use of the fact that any linear operation on a grid of data values can be converted into a convolution of some sort. The details of the numerical implementation are contained in [22].

But the real, amazing, property of iWave is that wave interaction with objects on the water surface is evaluated with a very simple

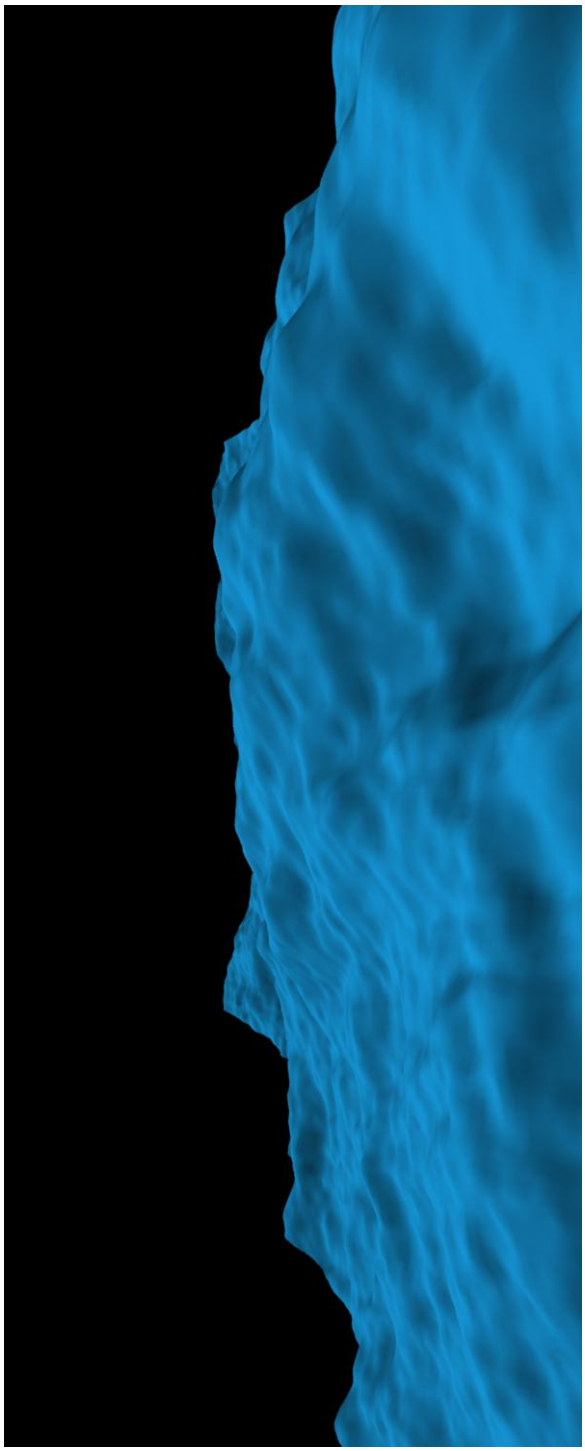


Figure 17: Simulated wave surface without the choppy algorithm applied. Rendered in BMRT with a generic plastic shader.

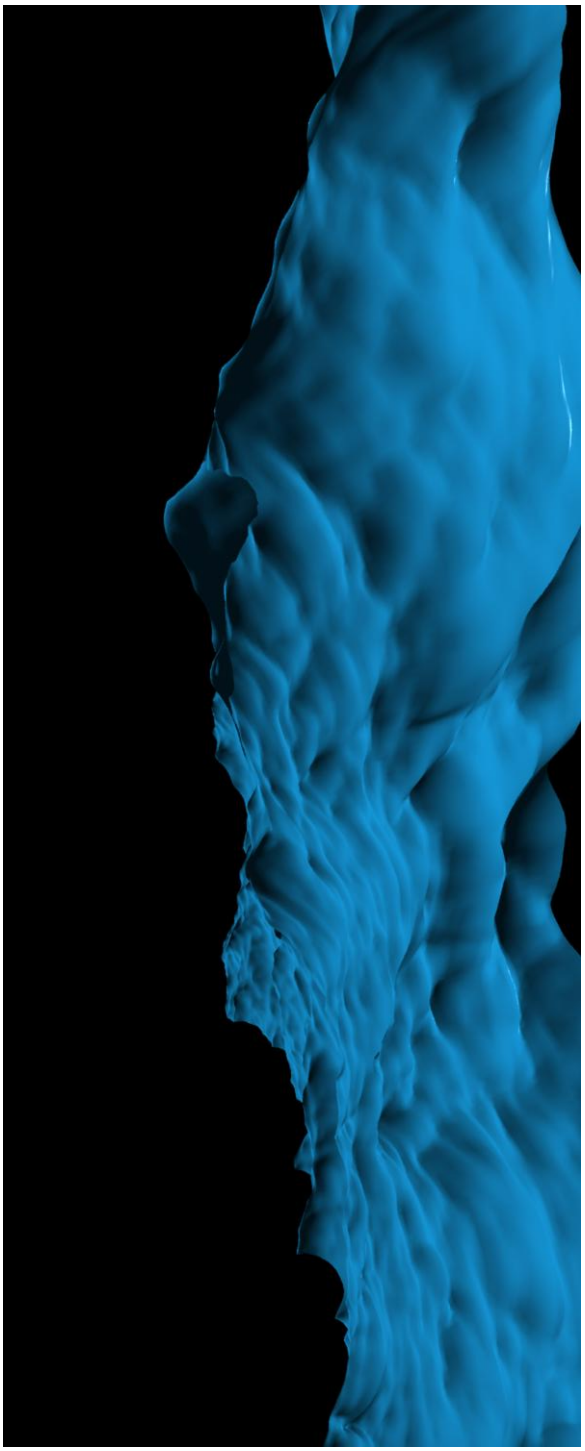


Figure 18: Same wave surface with strong chop applied. Rendered in BMRT with a generic plastic shader.

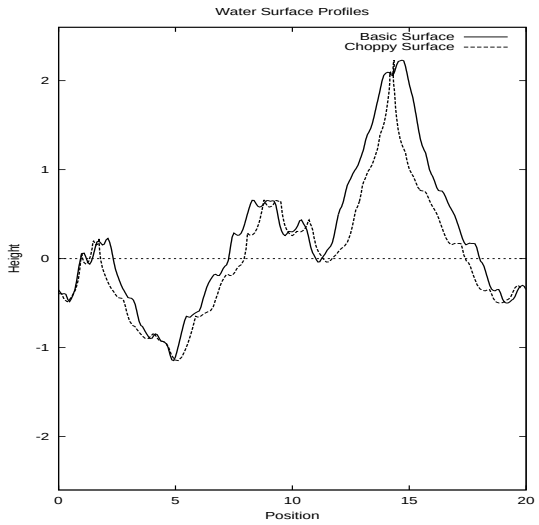


Figure 19: Profiles of the two surfaces, showing the effect of the choppy mapping.

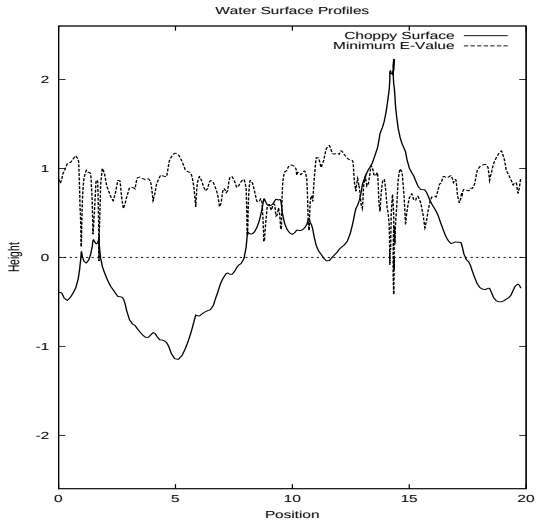


Figure 20: Plot of the choppy surface profile and the minimum eigenvalue. The locations of folds of the surface are clearly the same as where the eigenvalue is negative.

linear operation that amount to an image masking. For this purpose, an object in the water is described as a grid of mask values, zero meaning the object is present, 1 if the object is not present, and along the edges of the object values between 0 and 1 are useful as an antialiasing. Applying this mask to the wave height grid, the waves are effectively removed at grid points that the object is located at. With just this simple procedure, waves that are incident on the object reflect off of it in a realistic way. Figure 21 is a frame from a sequence, showing the wave height computed using iWave with several irregularly shaped objects.

Ordinarily, simulating waves that interact with an object on the water surface should involve a careful treatment of boundary conditions, matching water motion to the object on the boundary of the geometry, and enforcing no-slip conditions. This is a complex and time consuming computation that involves a certain amount of numerical black art. It is suprising that a simple process like masking the wave height, as described above, should not be sufficient. Yet, with the iWave procedure, correct-looking reflection/refraction waves happen in the simulation. It is not yet clear just how quantitatively accurate these interactions are. Figure 22 shows a rendered scene with a high resolution calculation of waves interacting with the hull of the ship.

5.1 Modifications for shallow water

Wave simulations based on the FFT method can simulate shallow water effects by using the dispersion relationship in equation 32. This only applies to a *flat* bottom. It would be nice to simulate wave propagation onto a beach, or pasta shallow subsurface sea mount, or over a submarine that is just below the surface. The iWave method is a great way of doing those things. Recall that the iWave method converts the mathematical operation

$$g\sqrt{-\nabla^2}\ h \tag{51}$$

into a convolution, so that effectively $g\sqrt{-\nabla^2}$ becomes a convolution kernel. A shallow bottom with depth D changes this term to (compare with the dispersion relationship)

$$g\sqrt{-\nabla^2}\ \tanh\left(\sqrt{-\nabla^2}D\right)\ h \tag{52}$$

This operation can also be converted into a convolution, with $g\sqrt{-\nabla^2}\ \tanh\left(\sqrt{-\nabla^2}D\right)$ becomes a convolution kernel.

How is this applied to a variable-depth bottom? The convolution kernel is a 13x13 matrix[22]. So we could build a collection of kernels over a range of depth values D. At each point on the grid, a custom kernel is constructed for the actual depth at that grid point by interpolating from the set of prebuilt values. Figure 23 is the wave height from a simulation using this technique. The bottom slopes from deep on the right hand side, to a depth of 0 on the left edge. In addition, there is a subsurface sea mount on the right.

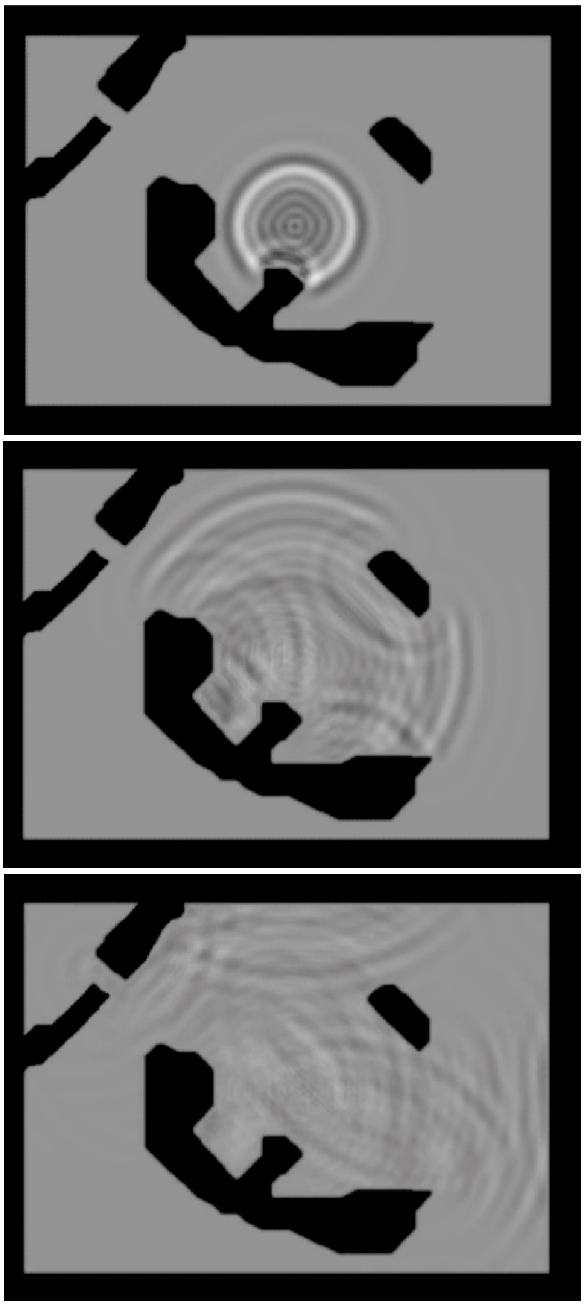
There are three important behaviors in this simulation that occur in real shallow water propagation:

1. Waves in shallow regions have large amplitudes that in deep regions.

2. As waves approach a beach, they pile up together and have a higher spatial frequency.

3. The subsurface sea mount causes a diffraction of waves.

In addition to these capabilities, iWave can also compute other quantities, such as cuspy waves and surface velocity.



(a) (b) (c)

Figure 21: A sequence of frames from an iWave simulation, showing waves reflecting off of objects in the water. The objects are the black regions. In the upper left there is also diffraction taking place as waves propagate the narrow channel and emerge in the corner region.

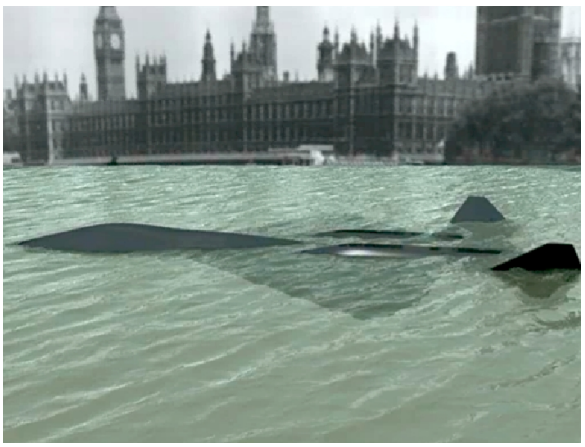


Figure 22: Frame from a simulation and rendering showing waves interacting with a ship.

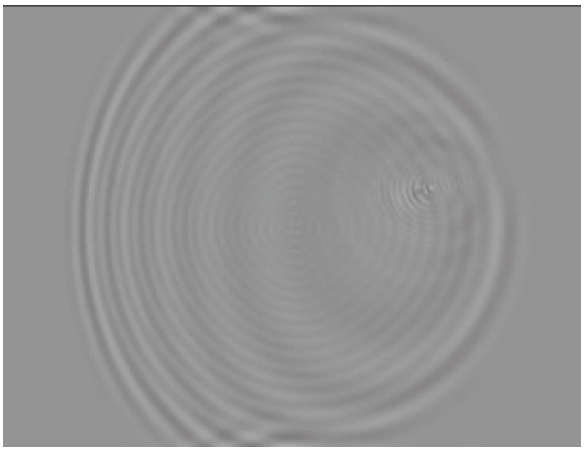


Figure 23: Frame from an iWave simulation with a variable depth shallow bottom.

6 Surface Wave Optics

The optical behavior of the ocean surface is fairly well understood, at least for the kinds of quiescent wave structure that we consider in these notes. Fundamentally, the ocean surface is a near perfect specular reflector, with well-understand relectivity and transmissity functions. In this section these properties are summarized, and combined into a simple shader for Renderman. There are circumstances when the surface does not appear to be a specular reflector. In particular, direct sunlight reflected from waves at a large distance from the camera appear to be spread out and made diffuse. This is due to the collection of waves that are smaller than the camera can resolve at large distances. The mechanism is somewhat similar to the underlying microscopic reflection mechnanisms in solid surfaces that lead to the Torrance-Sparrow model of BRDFs. Although the study of glitter patterns in the ocean was pioneered by Cox and Munk many years ago, the first models of this BRDF behavior that I am aware of were developed in the early 1980's. At the end of this section, we introduce the concepts and conditions, state the results, and ignore the in-between analysis and derivation.

Throughout these notes, and particularly in this section, we ignore one optical phenomenon completely: polarization. Polarization effects can be strong at a boundary interface like a water surface. However, since most of computer graphics under consideration ignores polarization, we will continue in that long tradition. Of course, interested readers can find literature on polarization effects at the air-water interface.

6.1 Specular Reflection and Transmission

Rays of light incident from above or below at the air-water interface are split into two components: a transmitted ray continuing through the interface at a refracted angle, and a reflected ray. The intensity of each of these two rays is diminished by reflectivity and transmissivity coefficients. Here we discussed the directions of the two outgoing rays. In the next subsection the coefficients are discussed.

6.1.1 Reflection

As is well known, in a perfect specular reflection the reflected ray and the incident ray have the same angle with respect to the surface normal. This is true for all specular reflections (ignoring roughening effects), regardless of the material. We build here a compact expression for the outgoing reflected ray. First, we need to build up some notation and geometric quantities.

The three-dimensional points on the ocean surface can be labelled by the horizontal position $\mathbf{x}$ and the waveheight $h(\mathbf{x}, t)$ as

$$\mathbf{r}(\mathbf{x}, t) = \mathbf{x} + \hat{\mathbf{y}} h(\mathbf{x}, t) \ , \tag{53}$$

where $\hat{\mathbf{y}}$ is the unit vector pointing straight up. At the point $\mathbf{r}$, the normal to the surface is computed directly from the surface slope $\epsilon(\mathbf{x}, t) \equiv \nabla h(\mathbf{x}, t)$ as

$$\hat{\mathbf{n}}_S(\mathbf{x}, t) = \frac{\hat{\mathbf{y}} - \epsilon(\mathbf{x}, t)}{\sqrt{1 + \epsilon^2(\mathbf{x}, t)}} \tag{54}$$

For a ray intersecting the surface at $\mathbf{r}$ from direction $\hat{\mathbf{n}}_i$, the direction of the reflected ray can depend only on the incident direction and the surface normal. Also, as mentioned before, the angle between the surface normal and the reflected ray must be the same as the angle between incident ray and the surface normal. You can verify for yourself that the reflected direction $\hat{\mathbf{n}}_r$ is

$$\hat{\mathbf{n}}_r(\mathbf{x}, t) = \hat{\mathbf{n}}_i - 2\hat{\mathbf{n}}_S(\mathbf{x}, t) \left(\hat{\mathbf{n}}_S(\mathbf{x}, t) \cdot \hat{\mathbf{n}}_i\right) \ . \tag{55}$$

Note that this expression is valid for incident ray directions on either side of the surface.

6.1.2 Transmission

Unfortunately, the direction of the transmitted ray is not expressed as simply as for the reflected ray. In this case we have two guiding principles: the transmitted direction is dependent only on the surface normal and incident directions, and Snell's Law relating the sines of the angles of the incident and transmitted angles to the indices of refraction of the two materials.

Suppose the incident ray is coming from one of the two media with index of refraction n_i (for air, $n = 1$, for water, $n = 4/3$ approximately), and the transmitted ray is in the medium with index of refraction n_r. For the incident ray at angle θ_i to the normal,

$$\sin\theta_i = \sqrt{1 - (\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S)^2} = |\hat{\mathbf{n}}_i \times \hat{\mathbf{n}}_S| \tag{56}$$

the transmitted ray will be at an angle θ_t with

$$\sin\theta_t = |\hat{\mathbf{n}}_t \times \hat{\mathbf{n}}_S| \ . \tag{57}$$

Snell's Law states that these two angles are related by

$$n_i \sin\theta_i = n_t \sin\theta_t \ . \tag{58}$$

We now have all the pieces needed to derive the direction of transmission. The direction vector can only be a linear combination of $\hat{\mathbf{n}}_i$ and $\hat{\mathbf{n}}_S$. It must satisfy Snell's Law, and it must be a unit vector (by definition). This is adequate to obtain the expression

$$\hat{\mathbf{n}}_t(\mathbf{x}, t) = \frac{n_i}{n_t}\hat{\mathbf{n}}_i + \Gamma(\mathbf{x}, t)\, \hat{\mathbf{n}}_S(\mathbf{x}, t) \tag{59}$$

with the function Γ defined as

$$\begin{aligned} \Gamma(\mathbf{x}, t) \quad &\equiv \quad \frac{n_i}{n_t}\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S(\mathbf{x}, t) \\ &\pm \quad \left\{1 - \left(\frac{n_i}{n_t}\right)^2 |\hat{\mathbf{n}}_i \times \hat{\mathbf{n}}_S(\mathbf{x}, t)|^2\right\}^{1/2} \ . \end{aligned} \tag{60}$$

The plus sign is used in Γ when $\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S < 0$, and the minus sign is used when $\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S > 0$.

6.2 Fresnel Reflectivity and Transmissivity

Accompanying the process of reflection and transmission through the interface is a pair of coefficients that describe their efficiency. The reflectivity R and transmissivity T are related by the constraint that no light is lost at the interface. This leads to the relationship

$$R + T = 1 \ . \tag{61}$$

The derivation of the expressions for R and T is based on the electromagnetic theory of dielectrics. We will not carry out the derivations, but merely write down the solution

$$R(\hat{\mathbf{n}}_i, \hat{\mathbf{n}}_r) = \frac{1}{2}\left\{\frac{\sin^2(\theta_t - \theta_i)}{\sin^2(\theta_t + \theta_i)} + \frac{\tan^2(\theta_t - \theta_i)}{\tan^2(\theta_t + \theta_i)}\right\} \tag{62}$$

Figure 24 is a plot of the reflectivity for rays of light traveling down onto a water surface as a function of the angle of incidence to the surface. The plot extends from a grazing angle of 0 degrees to perpendicular incidence at 90 degrees. As should be clear, variation of the reflectivity across an image is an important source of the "texture" or feel of water. Notice that reflectivity is a function of the angle of incidence relative to the wave normal, which in turn is directly related to the slope of the surface. So we can expect that a strong contributor to the texture of water surface is the pattern of slope, while variation of the wave height serves primarily as a wave hiding mechanism. This is the quantitative explanation of why the

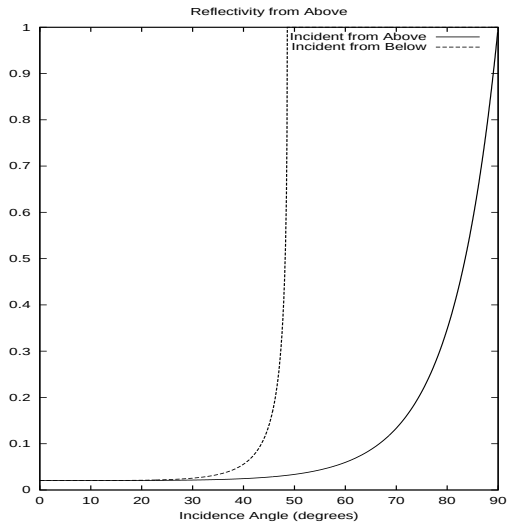


Figure 24: Reflectivity for light coming from the air down to the water surface, as a function of the angle of incidence of the light.

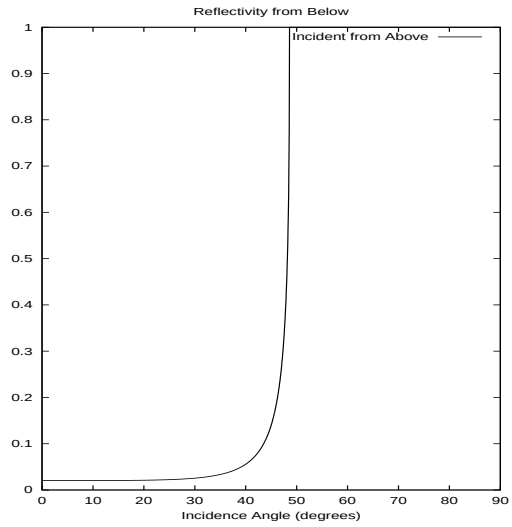


Figure 25: Reflectivity for light coming from below the water surface, as a function of the angle of incidence of the light.

surface slope more closely resembles rendered water than the wave height does, as we saw in the previous section when discussing figure 10.

When the incident ray comes from below the water surface, there are important differences in the reflectivity and transmissivity. Figure 25 shows the reflectivity as a function of incidence angle again, but this time for incident light from below. In this case, the reflectivity reaches unity at a fairly large angle, near 41 degrees. At incidence angles below that, the reflectivity is one and so there is no transmission of light through the interface. This phenomenon is *total internal reflection*, and can be seen just by swimming around in a pool. The angle at which total internal reflection begins is called Brewster's angle, and is given by, from Snell's Law,

$$\sin\theta_i^B = \frac{n_t}{n_i} = 0.75 \tag{63}$$

or $\theta_i^B = 48.6\,\mathrm{deg}$. In our plots, this angle is $90 - \theta_i^B = 41.1\,\mathrm{deg}$.

6.3 Building a Shader for Renderman

From the discussion so far, one of the most important features a rendering must emulate is the textures of the surface due to the strong slope-dependence of reflectivity and transmissivity. In this section we construct a simple Renderman-compliant shader using just these features. Readers who have experience with shaders will know how to extend this one immediately.

The shader exploits that fact that the Renderman interface already provides a built-in Fresnel quantity calculator, which provides R, T, $\hat{\mathbf{n}}_r$, and $\hat{\mathbf{n}}_t$ using the surface normal, incident direction vector, and index of refraction. The shader for the air-to-water case is as follows:

```
surface watercolorshader(
    color upwelling = color(0, 0.2, 0.3);
    color sky       = color(0.69,0.84,1);
    color air       = color(0.1,0.1,0.1);
    float nSnell    = 1.34;
    float Kdiffuse  = 0.91;
    string envmap   = "";
                          )
{
  float reflectivity;
  vector nI = normalize(I);
  vector nN = normalize(Ng);
  float costhetai = abs(nI . nN);
  float thetai = acos(costhetai);
  float sinthetat = sin(thetai)/nSnell;
  float thetat = asin(sinthetat);
  if(thetai == 0.0)
  {
    reflectivity = (nSnell - 1)/(nSnell + 1);
    reflectivity = reflectivity * reflectivity;
  }
  else
  {
    float fs = sin(thetat - thetai)
             / sin(thetat + thetai);
    float ts = tan(thetat - thetai)
             / tan(thetat + thetai);
    reflectivity = 0.5 * ( fs*fs + ts*ts );
  }
  vector dPE = P-E;
  float dist = length(dPE) * Kdiffuse;
  dist = exp(-dist);

  if(envmap != "")
  {
    sky = color environment(envmap, nN);
  }
  Ci =  dist * ( reflectivity * sky
                + (1-reflectivity) * upwelling )
     +  (1-dist)* air;
}
```

There are two contributions to the color: light coming downward onto the surface with the default color of the sky, and light coming upward from the depths with a default color. This second term will be discussed in the next section. It is important for incidence angles that are high in the sky, because the reflectivity is low and transmissivity is high.

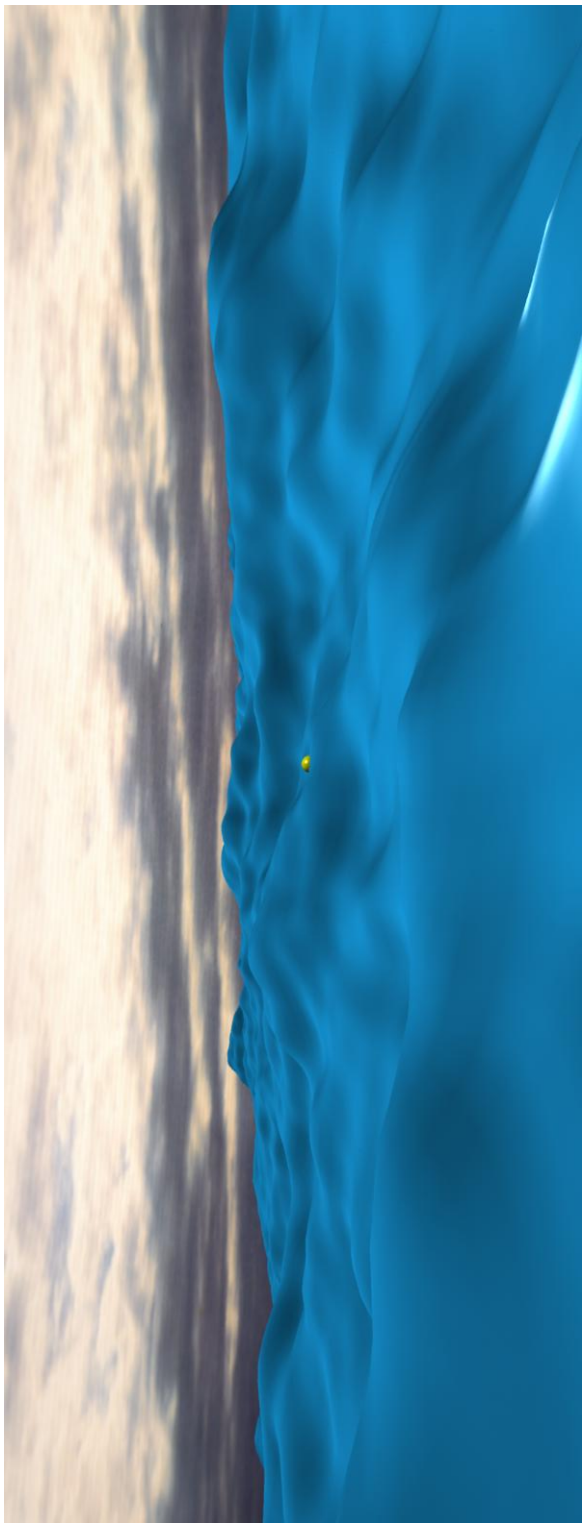


Figure 26: Simulated water surface with a generic plastic surface shader. Rendered with BMRT.

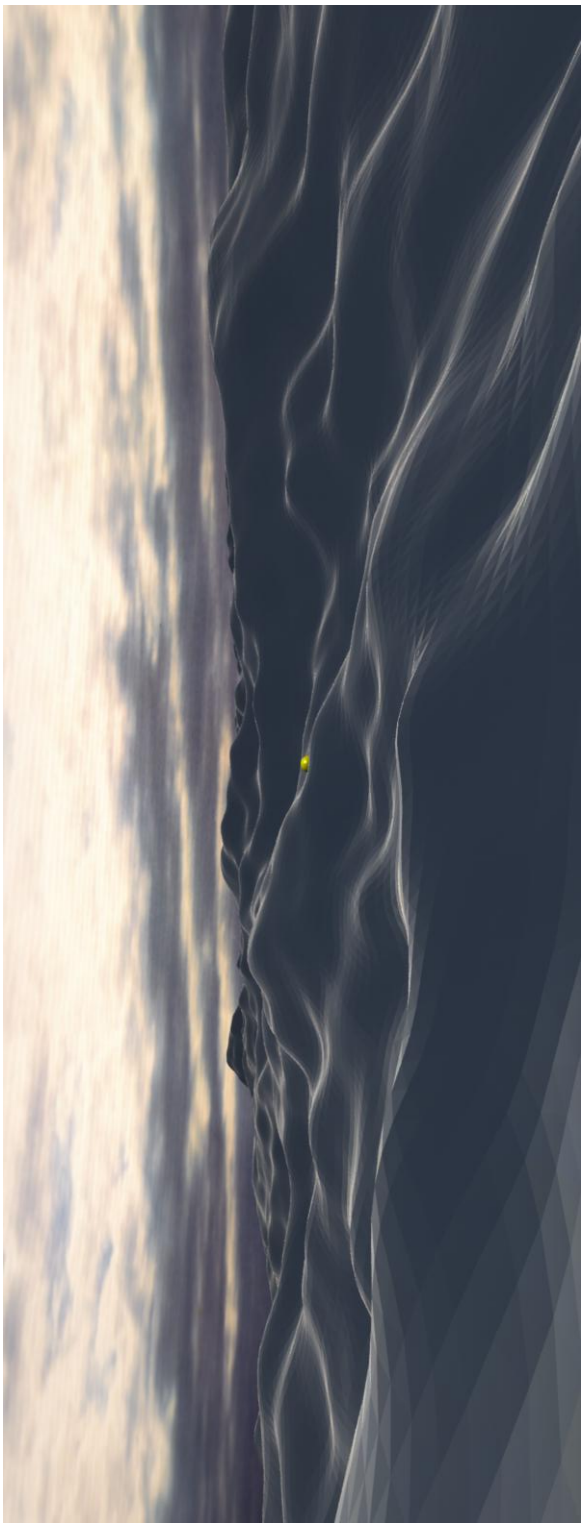


Figure 27: Simulated water surface with a realistic surface shader. Rendered with BMRT.

This shader was used to render the image in figure 27 using the BMRT raytrace renderer. For reference, the exact same image has been rendered in 26 with a generic plastic shader. Note that the realistic water shader tends to highlight the tops of the the waves, where the angle of incidence is nearly 90 degrees grazing and the reflectivity is high, while the sides of the waves are dark, where angle of incidence is nearly 0 that the reflectivity is low.

7 Water Volume Effects

The previous section was devoted to a discussion of the optical behavior of the surface of the ocean. In this section we focus on the optical behavior of the water volume below the surface. We begin with a discussion of the major optical effects the water volume has on light, followed by an introduction to color models researchers have built to try to connect the ocean color on any given day to underlying biological and physical processes. These models are built upon many years of in-situ measures off of ships and peers. Finally, we discuss two important effects, caustics and sunbeams, that sometimes are hard to grasp, and which produce beautiful images when properly simulated.

7.1 Scattering, Transmission, and Reflection by the Water Volume

In the open ocean, light is both scattering and absorbed by the volume of the water. The sources for these events are of three types: water molecules, living and dead organic matter, and non-organic matter. In most oceans around the world, away from the shore lines, absorption is a fairly even mixture of water molecules and organic matter. Scattering is dominated by organic matter however.

To simulate the processes of volumetric absorption and scattering, there are five quantities that are of interest: absorption coefficient, scattering coefficient, extinction coefficient, diffuse extinction coefficient, and bulk reflectivity. All of these coefficients have units of inverse length, and represent the exponential rate of attenuation of light with distance through the medium. The absorption coefficient a is the rate of absorption of light with distance, the scattering coefficient b is the rate of scattering with length, the extinction coefficient c is the sum of the two previous ones $c = a + b$, and the diffuse extinction coefficient K describes the rate of loss of intensity of light with distance after taking into account both absorption and scattering processes. The connection between K and the other parameters is not completely understood, in part because there are a variety of ways to define K in terms of operational measurements. Different ways change the details of the dependence. However, there is a condition called the *asymptotic* limit at very deep depths in the water, at which all operational definitions of K converge to a single value. This asymptotic value of K has been modeled in a variety of ways. There is a mathematically precise result that the ratio K/c depends only on b/c, the single scatter albedo, and some details of the angular distribution of individual scattering distributions. Figure 28 is an example of a model of K/c for reasonable water conditions. Models have been generated for the color dependence of K, most notably by Jerlov. In 1990, Austin and Petzold performed a revised analysis of spectral models, including new data, to produce refined models of K as a function of color. For typical visible light conditions in the ocean, K ranges in value from 0.03/meter to 0.1/meter. It is generally true that $a < K < c$.

One way to interpret these quantities for a simulation of water volume effects is as follows:

1. A ray of sunlight enters the water with intensity I (after losing some intensity to Fresnel transmission). Along a path underwater of a length s, the intensity at the end of the path is $I \exp(-cs)$, i.e. the ray of direct sunlight is attenuated as fast a possible.

2. Along the path through the water, a fraction of the ray is scattered into a distribution of directions. The strength of the scattering per unit length of the ray is b, so the intensity is proportional to $bI \exp(-cs)$.

3. The light that is scattered out of the ray goes through potentially many more scattering events. It would be nearly impossible to track all of them. However, the sum whole outcome of this process is to attenuate the ray along the path from the original path to the camera as $bI \exp(-cs) \exp(-K s_c)$, where s_c is the distance from the scatter point in the ocean to the camera.

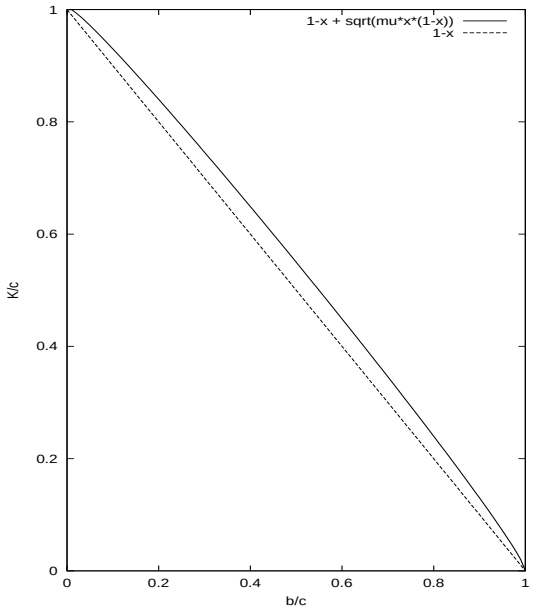


Figure 28: Dependence of the Diffuse Extinction Coefficient on the Single Scatter Albedo, normalized to the extinction.

A fifth quantity of interest for simulation is the bulk reflectivity of the water volume. This is a quantity that is intended to allow us to ignore the details of what is going on, treat the volume as a Lambertian reflector, and compute a value for bulk reflectivity. That number is sensitive to many factors, including wave surface conditions, sun angle, water optical properties, and details of the angular spread. Nevertheless, values of reflectivity around 0.04 seem to agree well with data.

7.2 The Underwater POV: Refracted Skylight, Caustics, and Sunbeams

Now that we have underwater optical properties at hand, we can look at two important phenomena in the ocean: caustics and sunbeams.

7.2.1 Caustics

Caustics, in this context, are a light pattern that is formed on surfaces underwater. Because the water surface is not flat, groups of

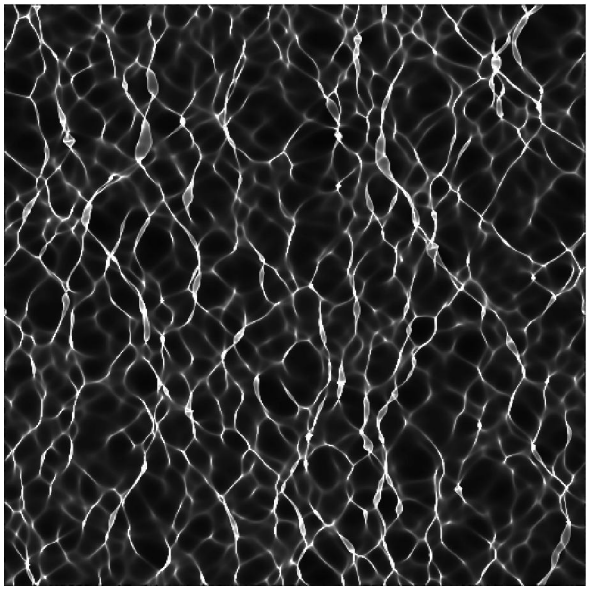


Figure 29: Rendering of a caustic pattern at a shallow depth (5 meters) below the surface.

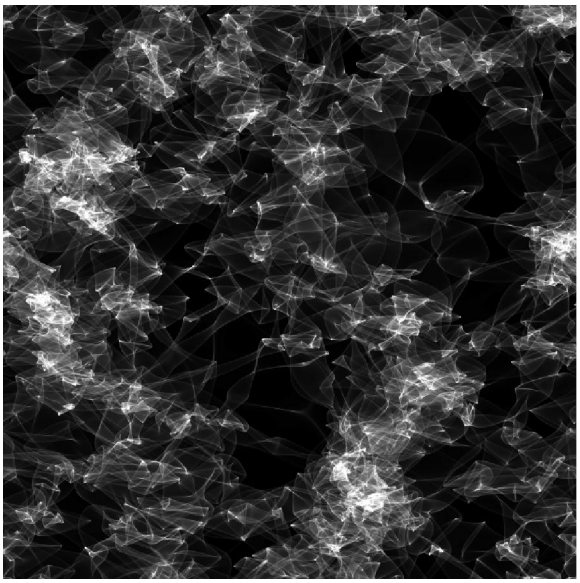


Figure 30: Rendering of a caustic pattern at great depth (100 meters) below the surface.

light rays incident on the surface are either focussed or defocussed. As a result, a point on a fictitious plane some depth below the ocean surface receives direct sunlight from several different positions on the surface. The intensity of light varies due to the depth, original contrast, and other factors. For now, lets write the intensity of the pattern as $I = Ref\ I_0$, with I_0 as the light intensity just above the water surface. The quantity Ref is the scaling factor that varies with position on the fictitious plane due to focussing and defocussing of waves, and is called a *caustic pattern*. Figure 29 shows an example of the caustic pattern Ref. Notice that the caustic pattern exhibits filaments and ring-like structure. At a very deep depth, the caustic pattern is even more striking, as shown in figure 30.

One of the important properties of underwater light that produce caustic patterns is conservation of flux. This is actually a simple idea: suppose a small area on the ocean surface has sunlight passing through it into the water, with intensity I at the surface. As we map that area to greater depths, the amount of area within it grows or shrinks, but most likely grows depending on whether the area is focussed or defocussed. The intensity at any depth within the water is proportional to inverse of the area of the projected region. Another way of saying this is that if a bundle of light rays diverges, their intensities are reduced to keep the product of intensity time area fixed.

Simulated caustic patterns can actually be compared (roughly) with real-world data. In a series of papers published throughout the 1970's, 1980's, and into the 1990's, Dera and others collected high-speed time series of light intensity[21]. As part of this data collection and analysis project, the data was used to generate a probability distribution function (PDF) for the light intensity. Figure 31 shows two PDFs taken from one of Dera's papers. The two PDF's were collected for different surface roughness conditions: rougher water tended to suppress more of the high magnitude fluctuations in intensity.

Figure 32 shows the pdf at two depths from a simulation of the ocean surface. These two sets do not match Dera's measurements because of many factors, but most importantly because we have not simulated the environmental conditions and instrumentation in Dera's experiments. Nevertheless, the similarity of figure 32 with

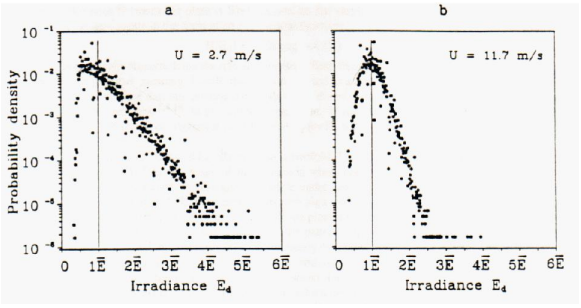


Figure 31: PDF's as measured by Dera in reference [21].

Dera's data is an encouraging point of information for the realism of the simulation.

7.2.2 Godrays

Underwater sunbeams, also called godrays, have a very similar origin to caustics. Direct sunlight passes into the water volume, focussed and defocussed at different points across the surface. As the rays of light pass down through the volume, some of the light is scattered in other directions, and a fraction arrives at the camera. The accumulated pattern of scattered light apparent to the camera are the godrays. So, while caustics are the pattern of direct sunlight that penetrates down to the floor of a water volume, sunbeams are scattered light coming from those shafts of direct sunlight in the water volume. Figure 33 demonstrates sunbeams as seen by a camera looking up at it.

References

[1] Jos Stam, "Stable Fluids," *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp 121-128, (1999).

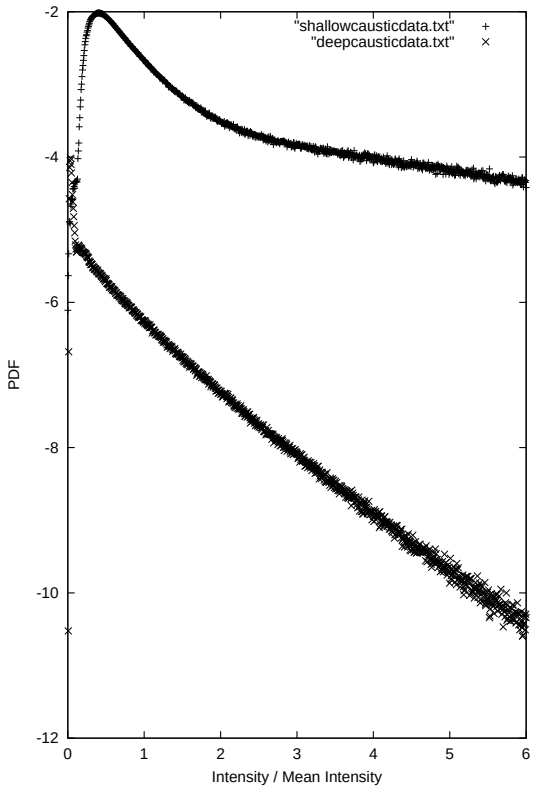


Figure 32: Computed Probability Density Function for light intensity fluctuations in caustics. (upper curve) shallow depth of 2 meters; (lower curve) deep depth of 10 meters.

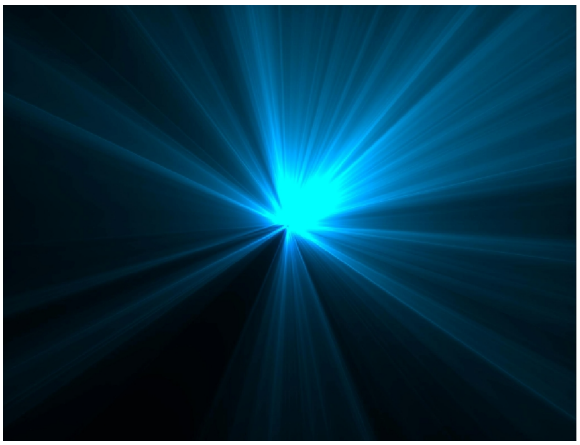


Figure 33: Rendering of sunbeams, or godrays, as seen looking straight up at the light source.

[2] Nick Foster and Ronald Fedkiw, "Practical animation of liquids," *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp 23-30, (2001).

[3] B.A. Cheeseman and C.P.R. Hoppel,"SIMULATING THE BALLISTIC IMPACT OF COMPOSITE STRUCTURAL ARMOR," http://www.asc2002.com/summaries/h/HP-08.pdf

[4] Simon Premoze, "Particle-Based Simulation of Fluids," *Eurographics 2003*, Vol 22, No. 3.

[5] Jeff Odien, "On the Waterfront", Cinefex, No. 64, p 96, (1995)

[6] Ted Elrick, "Elemental Images", Cinefex, No. 70, p 114, (1997)

[7] Kevin H. Martin, "Close Contact", Cinefex, No. 71, p 114, (1997)

[8] Don Shay, "Ship of Dreams", Cinefex, No. 72, p 82, (1997)

[9] Kevin H. Martin, "Virus: Building a Better Borg", Cinefex, No. 76, p 55, (1999)

[10] Simon Premože and Michael Ashikhmin, "Rendering Natural Waters," Eighth Pacific Conference on Computer Graphics and Applications, October 2000.

[11] Gary A. Mastin, Peter A. Watterger, and John F. Mareda, "Fourier Synthesis of Ocean Scenes", IEEE CG&A, March 1987, p 16-23.

[12] Alain Fournier and William T. Reeves, "A Simple Model of Ocean Waves", Computer Graphics, Vol. 20, No. 4, 1986, p 75-84.

[13] Darwyn Peachey, "Modeling Waves and Surf", Computer Graphics, Vol. 20, No. 4, 1986, p 65-74.

[14] Blair Kinsman, *Wind Waves, Their Generation and Propagation on the Ocean Surface*, Dover Publications, 1984.

[15] S. Gran, *A Course in Ocean Engineering*, *Developments in Marine Technology No. 8*, Elsevier Science Publishers B.V. 1992. See also `http://www.dnv.no/ocean/bk/grand.htm`

[16] Dennis B. Creamer, Frank Henyey, Roy Schult, and Jon Wright, "Improved Linear Representation of Ocean Surface Waves." J. Fluid Mech, **205**, pp. 135-161, (1989).

[17] Seibert Q. Duntley, "Light in the Sea," J. Opt. Soc. Am., **53**,2, pg 214-233, 1963.

[18] Curtis D. Mobley, *Light and Water: Radiative Transfer in Natural Waters*, Academic Press, 1994.

[19] *Selected Papers on Multiple Scattering in Plane-Parallel Atmospheres and Oceans: Methods*, ed. by George W. Kattawar, SPIE Milestones Series, **MS 42**, SPIE Opt. Eng. Press., 1991.

[20] R.W. Austin and T. Petzold, "Spectral Dependence of the Diffuse Attenuation Coefficient of Light in Ocean Waters: A Reexamination Using New Data," *Ocean Optics X*, Richard W. Spinrad, ed., SPIE **1302**, 1990.

[21] Jerzy Dera, Slawomir Sagan, Dariusz Stramski, "Focusing of Sunlight by Sea Surface Waves: New Measurement Results from the Black Sea," *Ocean Optics XI*, SPIE Proceedings, 1992.

[22] Jerry Tessendorf, “Interactive Water Surfaces,” *Game Programming Gems 4* , ed. Andrew Kirmse, Charles River Media, (2004).

[23] Stéphan T. Grilli home page, `http://131.128.105.170/ grilli/`.

[24] AROSS - Airborne Remote Optical Spotlight System, `http://www.aross.arete-dc.com`

8 Appendix: Sample code for interactive water surfaces

The code listed in this appendix is a working implementation of the iWave algorithm. As listed below, *iwave_paint* looks like a crude paint routine. The user can paint in two modes. When *iwave_paint* starts up, it begins running an iWave simulation on a small grid (200x200 with the settings listed). This grid is small enough that the iWave simulation runs interactively on cpus around 1 GHz and faster. The running of the simulation is not apparent since there are no disturbing waves at start up. The interface looks like figure 34. In the default mode at start up, the painting generates obstructions that show up in black and block water waves. Figure 35 shows an example obstruction that has been painted. Hitting the ’s’ key changes the painting mode to painting a source disturbance on the water surface. As you paint the disturbance, *iwave_paint* propagates the disturbance, reflecting off of any obstructions you may have painted. Figure 36 shows a frame after source has been painted inside the obstructed and allows a brief period of time to propagate inside the obstruction and exit, creating a diffraction pattern at the mouth of the obstruction.

A few useful keyboard options:

o Puts the paint mode into obstruction painting. This may be selected at any time.

s Puts the paint mode into source painting. This may be selected at any time.

```
//--------------------------------------------------
//
//  iwave_paint
//
//  demonstrates the generation and interaction of
//  waves around objects by allowing the user to
//  paint obstructions and source, and watch iwave
//  propagation.
//
//  author: Jerry Tessendorf
//          jerry@finelightvisualtechnology.com
//  August, 2003
//
//  This software is in the public domain.
//
//--------------------------------------------------

//--------------------------------------------------
//
//  usage:
//
//  iwave_paint is an interactive paint program
//  in which the user paints on a water surface and
//  the waves evolve and react with obstructions
//  in the water.
//
//  There are two paint modes.  Typing ’o’ puts the
//  program in obstruction painting mode. When you
//  hold down the left mouse button and paint, you
//  will see a black obstruction painted.  This
//  obstruction may be any shape.
//
//  Typing ’s’ puts the program in source painting
//  mode.  Now painting with the left mouse button
//  down generates a source disturbance on the water
//  surface.  The waves it produces evolve if as you
//  continue to paint.  The waves bounce of off any
//  obstructions that have been painted or are
//  subsequently painted.
//
//  Typing ’b’ clears all obstructions and waves.
//
//  Typing ’=’ and ’-’ brightens and darkens the display
//  of the waves.
//
//  Pressing the spacebar starts and stops the wave
//  evolution. While the evolution is stopped, you
//  can continue painting obstructions.
//
//
//  This code was written and runs under Linux. The
//  compile command is
//
//  g++ iwave_paint.C -O2 -o iwave_paint -lglut -lGL
//
//
//--------------------------------------------------


#include <cmath>
```

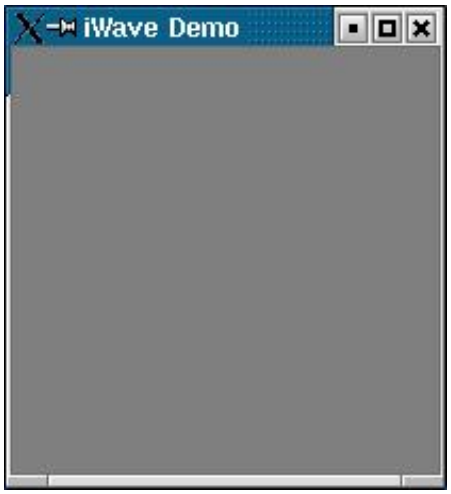


Figure 34: The *iwave_paint* window at startup.

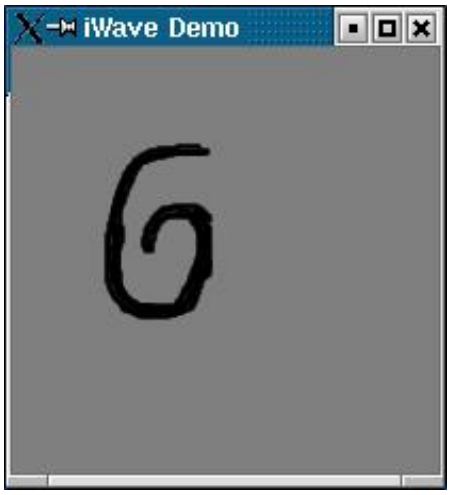


Figure 35: The *iwave_paint* window with obstruction painted.

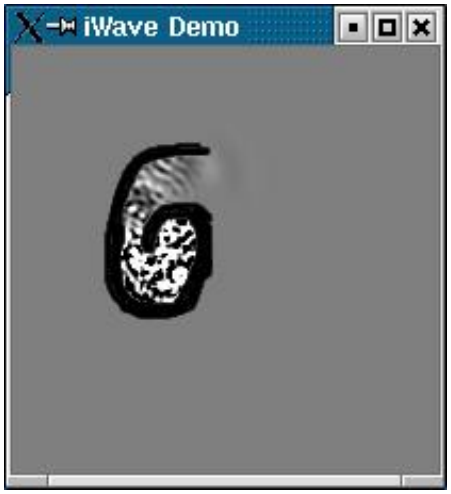


Figure 36: The *iwave_paint* window after painting source inside the obstruction and letting it propagate out.

```
#ifdef __APPLE__
  #include <GLUT/glut.h>
#else
  #include <GL/gl.h>   // OpenGL itself.
  #include <GL/glu.h>  // GLU support library.
  #include <GL/glut.h> // GLUT support library.
#endif

#include <iostream>

using namespace std;

int iwidth, iheight, size;
float *display_map;
float *obstruction;
float *source;

float *height;
float *previous_height;
float *vertical_derivative;

float scaling_factor;
float kernel[13][13];

int paint_mode;
enum{ PAINT_OBSTRUCTION, PAINT_SOURCE };

bool regenerate_data;
bool toggle_animation_on_off;

float dt, alpha, gravity;

float obstruction_brush[3][3];
float source_brush[3][3];

int xmouse_prev, ymouse_prev;


//------------------------------------------------------------
//
//  Initialization routines
//
//
// Initialize all of the fields to zero
void Initialize( float *data, int size, float value )
{
   for(int i=0;i<size;i++ ) { data[i] = value; }
}

// Compute the elements of the convolution kernel
void InitializeKernel()
{
    double dk = 0.01;
    double sigma = 1.0;
    double norm = 0;

    for(double k=0;k<10;k+=dk)
    {
        norm += k*k*exp(-sigma*k*k);
    }

    for( int i=-6;i<=6;i++ )
    {
for( int j=-6;j<=6;j++ )
{
   double r = sqrt( (float)(i*i + j*j) );
           double kern = 0;
   for( double k=0;k<10;k+=dk)
   {
       kern += k*k*exp(-sigma*k*k)*j0(r*k);
   }
         kernel[i+6][j+6] = kern / norm;
}
    }
}


void InitializeBrushes()
{
   obstruction_brush[1][1] = 0.0;
   obstruction_brush[1][0] = 0.5;
   obstruction_brush[0][1] = 0.5;
   obstruction_brush[2][1] = 0.5;
   obstruction_brush[1][2] = 0.5;
   obstruction_brush[0][2] = 0.75;
   obstruction_brush[2][0] = 0.75;
   obstruction_brush[0][0] = 0.75;
   obstruction_brush[2][2] = 0.75;

   source_brush[1][1] = 1.0;
   source_brush[1][0] = 0.5;
   source_brush[0][1] = 0.5;
   source_brush[2][1] = 0.5;
   source_brush[1][2] = 0.5;
   source_brush[0][2] = 0.25;
   source_brush[2][0] = 0.25;
   source_brush[0][0] = 0.25;
   source_brush[2][2] = 0.25;
}

void ClearObstruction()
{
    for(int i=0;i<size;i++ ){ obstruction[i] = 1.0; }
}

void ClearWaves()
{
    for(int i=0;i<size;i++ )
    {
```

```
      height[i] = 0.0;
      previous_height[i] = 0.0;
      vertical_derivative[i] = 0.0;
   }
}

//--------------------------------------------------

void ConvertToDisplay()
{
   for(int i=0;i<size;i++ )
   {
      display_map[i] = 0.5*( height[i]/scaling_factor + 1.0 )*obstruction[i];
   }
}

//--------------------------------------------------
//
//  These two routines,
//
//    ComputeVerticalDerivative()
//    Propagate()
//
//  are the heart of the iWave algorithm.
//
//  In Propagate(), we have not bothered to handle the
//  boundary conditions.  This makes the outermost
//  6 pixels all the way around act like a hard wall.
//

void ComputeVerticalDerivative()
{
   // first step:  the interior
   for(int ix=6;ix<iwidth-6;ix++)
   {
       for(int iy=6;iy<iheight-6;iy++)
{
   int index = ix + iwidth*iy;
   float vd = 0;
   for(int iix=-6;iix<=6;iix++)
   {
      for(int iiy=-6;iiy<=6;iiy++)
      {
  int iindex = ix+iix + iwidth*(iy+iiy);
         vd += kernel[iix+6][iiy+6] * height[iindex];
      }
   }
   vertical_derivative[index] = vd;
}
   }
}

void Propagate()
{
   // apply obstruction
   for( int i=0;i<size;i++ ) { height[i] *= obstruction[i]; }

   // compute vertical derivative
   ComputeVerticalDerivative();

   // advance surface
   float adt = alpha*dt;
   float adt2 = 1.0/(1.0+adt);
   for( int i=0;i<size;i++ )
   {
      float temp = height[i];
      height[i] = height[i]*(2.0-adt)-previous_height[i]-gravity*vertical_derivative[i];
      height[i] *= adt2;
      height[i] += source[i];
      height[i] *= obstruction[i];
      previous_height[i] = temp;
      // reset source each step
      source[i] = 0;
   }
}


//----------------------------------------
//
//  Painting and display code
//

void resetScaleFactor( float amount )
{
   scaling_factor *= amount;
}

void DabSomePaint( int x, int y )
{
   int xstart = x - 1;
   int ystart = y - 1;
   if( xstart < 0 ){ xstart = 0; }
   if( ystart < 0 ){ ystart = 0; }

   int xend = x + 1;
   int yend = y + 1;
   if( xend >= iwidth ){ xend = iwidth-1; }
   if( yend >= iheight ){ yend = iheight-1; }


   if( paint_mode == PAINT_OBSTRUCTION )
   {
      for(int ix=xstart;ix <= xend; ix++)
      {
         for( int iy=ystart;iy<=yend; iy++)
 {
            int index = ix + iwidth*(iheight-iy-1);
    obstruction[index] *= obstruction_brush[ix-xstart][iy-ystart];
 }
      }
   }
   else if( paint_mode == PAINT_SOURCE )
   {
      for(int ix=xstart;ix <= xend; ix++)
      {
         for( int iy=ystart;iy<=yend; iy++)
 {
            int index = ix + iwidth*(iheight-iy-1);
    source[index] += source_brush[ix-xstart][iy-ystart];
 }
      }
   }
   return;
}

//--------------------------------------------------
//
//  GL and GLUT callbacks
//
//--------------------------------------------------



void cbDisplay( void )
{
   glClear(GL_COLOR_BUFFER_BIT );
   glDrawPixels( iwidth, iheight, GL_LUMINANCE, GL_FLOAT, display_map );
   glutSwapBuffers();
}

// animate and display new result
void cbIdle()
{
   if( toggle_animation_on_off ) { Propagate(); }
   ConvertToDisplay();
   cbDisplay();
}


void cbOnKeyboard( unsigned char key, int x, int y )
{
   switch (key)
   {
      case '-': case '_':
      resetScaleFactor( 1.0/0.9 );
      regenerate_data = true;
      break;

      case '+': case '=':
      resetScaleFactor( 0.9 );
      regenerate_data = true;
      break;

      case ' ':
      toggle_animation_on_off = !toggle_animation_on_off;

      case 'o':
      paint_mode = PAINT_OBSTRUCTION;
      break;

      case 's':
      paint_mode = PAINT_SOURCE;
      break;

      case 'b':
      ClearWaves();
      ClearObstruction();
      Initialize( source, size, 0.0 );
      break;

      default:
      break;
   }
}

void cbMouseDown( int button, int state, int x, int y )
{
   if( button != GLUT_LEFT_BUTTON ) { return; }
   if( state != GLUT_DOWN ) { return; }
   xmouse_prev = x;
   ymouse_prev = y;
   DabSomePaint( x, y );
}



void cbMouseMove( int x, int y )
{
   xmouse_prev = x;
   ymouse_prev = y;
   DabSomePaint( x, y );
}



//--------------------------------------------------

int main(int argc, char** argv)
{
   // initialize a few variables
   iwidth = iheight = 200;
   size = iwidth*iheight;

   dt = 0.03;
   alpha = 0.3;
   gravity = 9.8 * dt * dt;

   scaling_factor = 1.0;
   toggle_animation_on_off = true;
```

```
   // allocate space for fields and initialize them
   height              = new float[size];
   previous_height     = new float[size];
   vertical_derivative = new float[size];
   obstruction         = new float[size];
   source              = new float[size];
   display_map         = new float[size];

   ClearWaves();
   ClearObstruction();
   ConvertToDisplay();
   Initialize( source, size, 0 );

   InitializeBrushes();

   // build the convolution kernel
   InitializeKernel();



   // GLUT routines
   glutInit(&argc, argv);

   glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
   glutInitWindowSize( iwidth, iheight );

   // Open a window
   char title[] = "iWave Demo";
   int Window_ID = glutCreateWindow( title );

   glClearColor( 1,1,1,1 );

   glutDisplayFunc(&cbDisplay);
   glutIdleFunc(&cbIdle);
   glutKeyboardFunc(&cbOnKeyboard);
   glutMouseFunc( &cbMouseDown );
   glutMotionFunc( &cbMouseMove );

   glutMainLoop();
   return 1;

};
```

Appendix B eWave Dynamics Algorithm

eWave is an improvement of the iWave algorithm Deusen et al. [2004]. In iWave the first order differential equations for the motion, the Bernoulli equation and continuity equation, are converted into one second order differential equation for the wave height, then solved by a finite-difference-in-time scheme. iWave is unstable to strong inputs. In eWave, the coupled first order equations are retained and solved exactly via an exponential map solver. With this approach, eWave is very stable and very accurate. The processing for interaction with objects intersecting the water remains unchanged from iWave. This appendix contains reference Tessendorf [2014] with the eWave solution.

eWave: Using an Exponential Solver on the iWave Problem

Jerry Tessendorf

March 16, 2014

1 iWave Dynamics

The iWave equations of motion for the surface displacement h and velocity potential ϕ are:

$$\frac{\partial h(\mathbf{x},t)}{\partial t} = \sqrt{-\nabla^2}\phi(\mathbf{x},t) \quad (1)$$

$$\frac{\partial \phi(\mathbf{x},t)}{\partial t} = -gh(\mathbf{x},t) \quad (2)$$

$$(3)$$

To see how the exponential solver approach helps, it would be good to set up a different way to express these equations in terms of a couplet

$$W(\mathbf{x},t) = \begin{bmatrix} h(\mathbf{x},t) \\ \phi(\mathbf{x},t) \end{bmatrix} \quad (4)$$

The couplet has the equation of motion

$$\frac{\partial W(\mathbf{x},t)}{\partial t} = \mathcal{M}\ W(\mathbf{x},t) \quad (5)$$

where $\mathcal{M}$ is the matrix

$$\mathcal{M} = \begin{bmatrix} 0 & \sqrt{-\nabla^2} \\ -g & 0 \end{bmatrix} \quad (6)$$

2 eWave: Exponential Solution

Equation 5 has the exact exponential solution

$$W(\mathbf{x},t) = \exp\{\mathcal{M}t\}\ W(\mathbf{x},0) \quad (7)$$

This can be built more explicitly, because if you work it out, you see the following identities:

$$\mathcal{M}^2 = -\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} g\sqrt{-\nabla^2} \tag{8}$$

$$\mathcal{M}^3 = -\mathcal{M}\, g\sqrt{-\nabla^2} \tag{9}$$

$$\mathcal{M}^{2n} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \left(-g\sqrt{-\nabla^2}\right)^n \tag{10}$$

$$\mathcal{M}^{2n+1} = \mathcal{M}\, \left(-g\sqrt{-\nabla^2}\right)^n \tag{11}$$

So, if we expand the exponential into a Taylor series:

$$\exp\{\mathcal{M}t\} = \sum_{n=0}^{\infty} \frac{(\mathcal{M})^n t^n}{n!} \tag{12}$$

and separate the powers into even and odd sets

$$\exp\{\mathcal{M}t\} = \sum_{n=0}^{\infty} \frac{(\mathcal{M})^{2n} t^{2n}}{(2n)!} + \sum_{n=0}^{\infty} \frac{(\mathcal{M})^{2n+1} t^{2n+1}}{(2n+1)!} \tag{13}$$

Using the identities above,

$$\exp\{\mathcal{M}t\} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \sum_{n=0}^{\infty} \frac{(-g\sqrt{-\nabla^2}t^2)^n}{(2n)!} + \mathcal{M}t \sum_{n=0}^{\infty} \frac{(-g\sqrt{-\nabla^2}t^2)^n}{(2n+1)!} \tag{14}$$

The first infinite series is the series for the cosine, and the second is the series for the sine. So if we define $\hat{\omega} \equiv \sqrt{g\sqrt{-\nabla^2}}$

$$\exp\{\mathcal{M}t\} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cos(\hat{\omega}t) + \frac{\mathcal{M}}{\hat{\omega}} \sin(\hat{\omega}t) \tag{15}$$

$$= \begin{bmatrix} \cos(\hat{\omega}t) & \frac{\sqrt{-\nabla^2}}{\hat{\omega}} \sin(\hat{\omega}t) \\ \frac{-g}{\hat{\omega}} \sin(\hat{\omega}t) & \cos(\hat{\omega}t) \end{bmatrix} \tag{16}$$

The nice thing about this is that, because g and ∇^2 are not time dependent quantities, this solution is exact for any value of time t. This is the exact solution for any time period:

$$h(\mathbf{x}, t+\Delta t) = \cos(\hat{\omega}\Delta t)\ h(\mathbf{x}, t) + \frac{\sqrt{-\nabla^2}}{\hat{\omega}} \sin(\hat{\omega}\Delta t)\ \phi(\mathbf{x}, t) \tag{17}$$

$$\phi(\mathbf{x}, t+\Delta t) = \cos(\hat{\omega}\Delta t)\ \phi(\mathbf{x}, t) - \frac{g}{\hat{\omega}} \sin(\hat{\omega}\Delta t)\ h(\mathbf{x}, t) \tag{18}$$

In real space, the tricky part of this is the complex way that ∇^2 is in the solution. Using convolution on this would be similar to the old iWave approach, but still better because the dynamics has been integrated already.

3 FFT Form

In Fourier space, the appearance of ∇^2 simplifies considerable, because now the operator $\hat{\omega}$ becomes the dispersion relation $\hat{\omega} \rightarrow \omega(k) = \sqrt{gk}$, where k is the absolute magnitude of the Fourier vector. For the initial Fourier amplitudes $\tilde{h}(\mathbf{k})$ and $\tilde{\phi}(\mathbf{k})$

$$\tilde{h}(\mathbf{k}, t + \Delta t) = \cos(\omega(k)\Delta t)\ \tilde{h}(\mathbf{k}, t) + \frac{k}{\omega(k)} \sin(\omega(k)\Delta t)\ \tilde{\phi}(\mathbf{k}, t) \quad (19)$$

$$\tilde{\phi}(\mathbf{k}, t + \Delta t) = \cos(\omega(k)\Delta t)\ \tilde{\phi}(\mathbf{k}, t) - \frac{g}{\omega(k)} \sin(\omega(k)\Delta t)\ \tilde{h}(\mathbf{k}, t) \quad (20)$$

Equations 19 and 20 are what is in the FFTDynamics code.

4 Convolution Form

If we want to evaluate the dynamics as a convolution, the FFT form can be used to construction the convolutions. In convolution form, there are three convolution kernels that apply as

$$h(\mathbf{x}, t) = C(\mathbf{x}) \otimes h(\mathbf{x}) + S(\mathbf{x}) \otimes \phi(\mathbf{x}) \quad (21)$$

$$\phi(\mathbf{x}, t) = C(\mathbf{x}) \otimes \phi(\mathbf{x}) - T(\mathbf{x}) \otimes h(\mathbf{x}) \quad (22)$$

where $\otimes$ denotes spatial convolution and the convolution kernels are:

$$C(\mathbf{x}) = \int \frac{d^2k}{(2\pi)^2}\ e^{i\mathbf{k}\cdot\mathbf{x}}\ \cos(\omega(k)\Delta t) \quad (23)$$

$$S(\mathbf{x}) = \int \frac{d^2k}{(2\pi)^2}\ e^{i\mathbf{k}\cdot\mathbf{x}}\ \sin(\omega(k)\Delta t)\ \frac{k}{\omega(k)} \quad (24)$$

$$T(\mathbf{x}) = \int \frac{d^2k}{(2\pi)^2}\ e^{i\mathbf{k}\cdot\mathbf{x}}\ \sin(\omega(k)\Delta t)\ \frac{g}{\omega(k)} \quad (25)$$

Explicitly in terms of integration, these convolutions are:

$$h(\mathbf{x}, t) = \int d^2y\ C(\mathbf{x} - \mathbf{y})\ h(\mathbf{y}) + S(\mathbf{x} - \mathbf{y})\ \phi(\mathbf{y}) \quad (26)$$

$$\phi(\mathbf{x}, t) = \int d^2y\ C(\mathbf{x} - \mathbf{y})\ \phi(\mathbf{y}) - T(\mathbf{x} - \mathbf{y})\ h(\mathbf{y}) \quad (27)$$

The practical implementation of convolution is as a moving window filter on the 2D gridded data for h and ϕ. Imagining these quantities have values h_{ij}, ϕ_{ij} on the rectangular grid, the time updates are

$$h_{ij}(t + \Delta t) = \sum_{k,l=-N/2}^{N/2} C_{kl}\ h_{i+k\ j+l}(t) + S_{kl}\ \phi_{i+k\ j+l}(t) \quad (28)$$

$$\phi_{ij}(t+\Delta t) \;=\; \sum_{k,l=-N/2}^{N/2} C_{kl}\ \phi_{i+k\ j+l}(t) \;-\; T_{kl}\ h_{i+k\ j+l}(t) \qquad (29)$$

Where N is the size of the square moving window, and C_{kl}, S_{kl}, and T_{kl} are the values our the kernels at discrete grid intervals. Strategies for obtaining these discrete kernels are discussed below.

Theoretically, if we take the limit that N is the size of the full simulation grid and apply periodic boundary conditions, the moving window filter produces identical results to the FFT approach, although the FFT approach is dramatically faster at it. But the moving window filter opens up many other strategies that can be applied when simple periodic boundary conditions are no longer feasible, and assuming that a small-enough window size N produces results that look good. In iWave, the smallest window size recommended was 13×13, but here a size that looks good will probably be larger, perhaps 30×30 or larger depending on the circumstances and desired visual qualities. In general, the larger the value of N, the better the quality of the propagation motion.

5 Boundary Conditions

The convolution approach breaks away from the constraint that the fields h and ϕ be periodic. That freedom also imposes a requirement that boundary conditions be specified when applying the convolution. Here are a few different strategies. Note that all of these strategies can be applied in a single simulation by choosing different ones on each boundary.

5.1 Periodic

Of course, periodic boundary conditions can still be imposed, simply by wrapping the indexing of $i+k$ and $j+l$ in the convolution sums.

5.2 Fixed Ghost Values

When the values of $i+k$ or $j+l$ extend beyond the bound of the grid, ghost values for h and ϕ can be prescribed.

5.3 Tiled Simulation Grids

If you are running independent eWave simulations on multiple grids that butt up agains each other, the simultions can be coupled via the boundary conditions. When the values of $i+k$ or $j+l$ extend beyond the bound of one grid, values for h and ϕ can be retrieved from the adjoining grid. This will naturally propagate waves and momentum between simulation grids without requiring extending each grid with gridpoints for ghost values (although ghost values is one approach to implementing the approach here).

6 Convolution Kernel Construction

The approach below is to find expressions for the convolution kernels C, S, and T. There are two methods we will look at. The first is a highly theoretical way that produces practically poor performance. The second is a brute-force method that is fast, flexible, and works.

6.1 The Impractical Theoretical Way

For the deep water case, using the symmetries of the integrands, we can define the dimensionless variable ξ as

$$\xi^2 \;=\; \frac{gt^2}{|\mathbf{x}|} \tag{30}$$

Using this variable, these convolution kernels can be written as

$$C(\mathbf{x}) = \frac{1}{|\mathbf{x}|^2}\ \mathcal{C}(\xi) \tag{31}$$

$$S(\mathbf{x}) = \frac{1}{|\mathbf{x}|^{5/2}\sqrt{g}}\ \mathcal{S}(\xi) \tag{32}$$

$$T(\mathbf{x}) = \frac{\sqrt{g}}{|\mathbf{x}|^{3/2}}\ \mathcal{T}(\xi) \tag{33}$$

where

$$\mathcal{C}(\xi) = \int_0^\infty du\ u\ J_0(u)\ \cos(\sqrt{u}\ \xi) \tag{34}$$

$$\mathcal{S}(\xi) = \int_0^\infty du\ u^{3/2}\ J_0(u)\ \sin(\sqrt{u}\ \xi) \tag{35}$$

$$\mathcal{T}(\xi) = \int_0^\infty du\ u^{1/2}\ J_0(u)\ \sin(\sqrt{u}\ \xi) \tag{36}$$

While this approach to generating the convolution kernels may prove interesting in the future, trying to apply these formula has not yet produced a practically useful approach.

6.2 The Practical Brute-Force Way

The simplest way to quickly and accurately construct C, S, and T as $N \times N$ moving window filters is to use FFTs to directly evaluate the integrals in equations 23 – 25. The procedure is as follows:

1. Create three grids, one each for C, S, and T. The dimensions of these grids should *not* be $N \times N$. They should be the dimensions of the simulation grid for h and ϕ. This is to ensure that spatial scales properly contribute to the filter values.

2. Assuming you are using an FFT package like FFTW, initialize the values in the three grids to 0, with the single gridpoint $i = 0, j = 0$ initialized to $\Delta x \Delta y$.

3. FFT the three grids to Fourier space. All of the grid points should now have identical values, and the exact value depends on (a) with FFT package you use, and (b) the number of grid points.

4. For each gridpoint in each grid, multiply the value in the gridpoint by the integrand in the corresponding equation. For the C grid, multiply by $\cos(\omega(k)\Delta t)$, the S grid by $\sin(\omega(k)\Delta t)\ \frac{k}{\omega(k)}$, and the T grid by $\sin(\omega(k)\Delta t)\ \frac{g}{\omega(k)}$.

5. Inverse FFT the three grids back to real space. Be sure to apply the normalization that the FFT package requires.

6. The three grids should now be filled with values of C, S, and T for any moving window size up the size of the full simulation grid.

7. Choose a value for N, create three $N \times N$ filter grids, and fill them with values from the larger C, S, T grids just generated.

8. You can now discard the full-sized C, S, and T grids.

Appendix C Sparse Grid Volume Rendering on a GPU

This appendix is a final report written by students for an honors undergraduate class.

Spare Grid Volume Rendering on a GPU

Sam Bryce
Clemson University

Zach Welch
Clemson University

Spring 2012

1 Introduction

Volume rendering is a method for creating images from a three dimensional set of particles. Volume rendering has a wide variety of applications. Medical technology uses volume rendering in Computer Tomography (CT) scans and Magnetic Resonance Imaging (MRI). Scientific visualization applications often use volume rendering techniques. Volume rendering is often used in the digital special effects to render amorphous and/or gaseous elements like clouds, dust, smoke, or flames 5. Something like a cloud can be more realistically modeled by a volume of particles than the traditional graphics paradigm of representing objects as surfaces. This more realistic modeling in turn leads to a more realistic image. Volume renderers can take into account factors such as how light travels through various mediums and the effects of light scattering. The elements to be rendered are modeled as sets of volumetric data within a 3D grid. This grid is conceptually made up of 1x1x1 cubes called voxels 2. Each voxel stores a density and a color of the particle at that specific voxel.

One of the key limitations of traditional volume rendering is memory size. If at each voxel we store a single 4 byte floating point number and three 4 byte floats representing the RGB color at this voxel, we end up storing 16 bytes for every voxel. A single voxel is inconsequential in terms of memory usage. However, volume rendering utilizes large grids of voxels to work. The larger the volume to be rendered and the more detailed the volume needs to be, the larger the array of voxels. Rendering a 1000^3 grid with 16 bytes allocated for each grid point would require a minimum of 15 GB of main

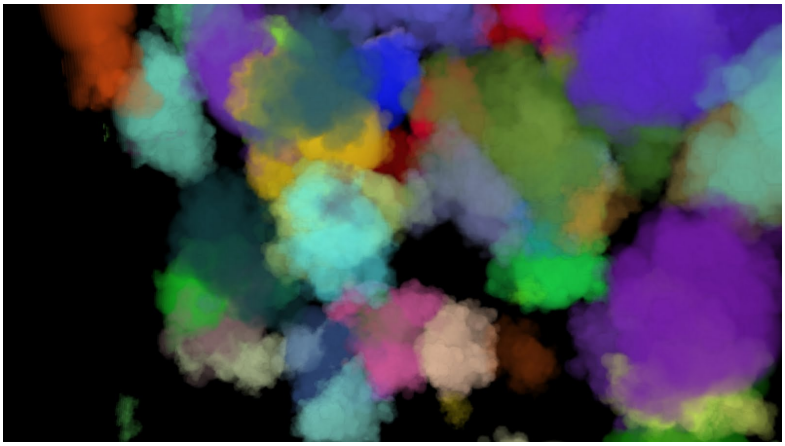


Figure 1: A set of volume rendered clouds

memory, well beyond most machines. Also, it is very likely that a substantial majority of voxels are empty, meaning that much of the memory allocated is unused and therefore redundant. An ideal situation would involve storing only non-empty voxels in memory, with the understanding that if a voxel is not in memory, then it contains specified default values. This is the basic idea upon which sparse grids are built. Sparse grids, as opposed to dense grids that allocate memory for each grid point, regardless of its value, only store a portion of the data 4. When the volume renderer attempts to access data at a grid point, the sparse grid must determine whether this grid point is actually allocated and return the appropriate values.

2 Background

Sparse Grids

Sparse grids are not appropriate for all situations. If there are no memory management concerns, or if the importance of shorter render times outweighs decreasing the size of the data, then sparse grids will not likely be of much use. Even in situations where using less memory is an important goal, sparse grids may not be a useful solution. There is a threshold of data a sparse grid can hold beyond which the sparse grid is less memory efficient. This threshold varies depending on the implementation. Since sparse grids store memory in a noncontinuous fashion, extra information must be stored to

retrieve the correct data. This information will end up being more of a burden on memory if too much of the grid is allocated. Sparse grids are useful for a specific class of problems. If the application contains a grid that is mostly empty and for which better memory usage justifies slower rendering speeds, sparse grids are appropriate. The added computation in looking up values is a sparse grid is what causes the slowdown in render time.

OpenCL

Specifically, we are interested in building and analyzing the performance of sparse grid implementations in OpenCL. OpenCL is a programming framework designed to take advantage of the heterogeneous nature of modern computer systems 1. One of the major benefits of OpenCL is the ability to execute code on graphical processing units (GPU). GPUs are highly parallel, with many threads of execution. This inherent parallelism combined with the fast floating point processing cores used in GPUs makes these devices very appealing for solving parallelizable problems (like volume rendering). OpenCL seeks to take advantage of the attributes of the GPU. Programs written for OpenCL must have a host and at least one kernel written in order to function. The host program, written in a language like C or C++, is tasked with setting up the kernel and handling interactions between the kernel and the rest of the system.

Common tasks performed by the kernel include discovering what OpenCL devices are in the system, loading a kernel, choosing the appropriate OpenCL device for the kernel to run on, and loading data onto the OpenCL device. OpenCL devices are usually CPUs and GPUs, and the host program can specify what type of OpenCL device a kernel is to run on. Kernels are written in OpenCL C, a C like language with some alterations. The kernels written for the volume renderers used in this paper utilize the data parallel programming model. In other words, a set of input data is split up and a sequence of operations is performed on each element of the input data in separate threads of execution. In this case, kernels contain the sequence of operations. This same kernel is run concurrently on many threads.

Each data element is given a unique global id, and using this id, each thread spawned is given a different elements id. The host program specifies the number of threads to spawn, which is often the number of data elements to be processed. One of the important factors in OpenCL is device memory. Since the CPU the host is executing on and the device executing the kernel are

almost always different, data the kernel needs must be loaded into the devices global memory. For most systems, this device memory is smaller than main memory, especially if the device in question is a GPU. Thus, efficient memory management is even more important when using OpenCL. OpenCL is an attractive language for volume renderers because of the speedup provided by GPU concurrency, but memory limitations are a serious concern. Sparse grids are a valuable extension that can potentially greatly expand the size of renderable grids.

Related Work

Several popular libraries already exist that utilize sparse grids. Field3D is an open source C++ library originally developed by Sony Pictures Imageworks 2. While Field3D uses dense grids by default (which are generally much faster but bigger), the library does have sparse grid options available. Field3D uses a scheme similar to the Block-partition sparse grid implementation described below. In a nutshell, Field3D divides the grid into multi-voxel blocks. These blocks are only allocated when a voxel within a given block is set. Field3D has some additional functionality not found in our implementations, such as deallocating blocks and iterating over blocks. This functionality was not necessary for our purposes and thus was not added.

Gigavoxel is a system developed by Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann 3. It uses N^3-trees to store and access voxel data. Each node of an N^3-tree can be subdivided into $N^3 3$ children. N^3-trees are a more generalized form of octrees, which are N^3 trees with N = 2. Different values of N can be used to achieve different performance goals. If N is small, the data structure will be more memory efficient. If N is large, the data structure will allow for quicker traversal. Each node in Gigavoxels N^3-tree stores either a block of pointers or a single value. The single value represents a value that is shared by all elements in the particular N^3 sized block. Usually this will be zero, representing an empty block. All node data and blocks of voxels are stored in texture memory. Gigavoxel is capable of rendering 2048^3 sized RGBA data in real time on GPUs.

3 C++/C implementations

Before attempting to build an OpenCL volume renderer, a number of implementations of sparse grids were created. These implementations were analyzed and compared based on memory usage, lookup time, and load time. The grids described below were implemented as C++ classes with a common API. The one C struct implemented was written to match this API as closely as possible. Several variations are made where appropriate, but the important methods are :

get(int,int,int) - given an index for each dimension, return the value at grid point
set(float,int,int,int) - given an index for each dimension, set the grid point to the given float value
init(int,int,int) - given dimensions for x, y, and z components, initialize the grid
In addition, all sparse implementations have an additional method
setDefVal(float) - sets the default value for the grid

Non Sparse Implementation

As a baseline for comparison, a dense grid class was implemented. Given X,Y, and Z dimensions for the grid, a 1D float array of size $X * Y * Z$ is dynamically allocated. Assuming an indexing scheme from 0 to N-1, grid point (i,j,k) would be accessed by indexing into the array

$$index = i + j * X + k * X * Y \tag{1}$$

This value is used both when getting and setting this grid point. Since there is very little computation and the memory is all allocated beforehand, both operations are performed very quickly.

C++STL Maps

Two implementation of sparse grids were created using the Map class from C++s Standard Template Library (STL). Abstractly, maps are a set of (key,value) pairs. A map can store a value with a unique key. If given a potential key, a map will either indicate that the key does not exist or return the corresponding value. The C++ STL map class is templated; a type

must be specified for both the key and the value. In both implementations described below, the value type is float, while the key type varies based on implementation. Note that the map class is not currently portable to OpenCL; these two implementations served as a baseline for completely sparse grids.

The first implementation has a key,value type of $< int, float >$; given an int key, there would potentially be a float value. This associative storage of data is very similar to the general idea of sparse grids and a sparse grid can easily be implemented using an $< int, float >$ map. First, a default float value must be defined. Though commonly 0.0 (grid points with no data would have no density) , the value is application specific. The grid must then be initialized with the X, Y, and Z dimensions of the grid. Though no memory is allocated, the grid dimensions must still be known for correct getting and setting. When a value F is to be set set at grid point (i,j,k), the key value pair to be inserted into the map will be $(i + j * X + k * X * Y, F)$. This insertion is only performed if $F \neq defVal$. In this way, only relevant data is stored in the grid. When getting a value from position (i,j,k) on the grid, if the map does not contain key i + j * X + k * X * Y, then the default value is returned. If $i + j * X + k * X * Y$ does have an associated value, it is returned. Whereas the dense grid had O(k) set and get methods, the complexity of the set and get methods for the map$< int, float >$ sparse grid is O(log n), where n is the number of elements in the map.

The second implementation of sparse grids using STL maps has a key,value pair type of $< int, map < int, map < int, float >>>$. Each of the three ints in the key,value pair type correspond to a dimension of the grid. The get method works by checking for (i,j,k) if any value has been stored with an X index of i, if so whether any value has also been stored with Y value of j, and so on. Setting works in a similar though reverse way. The benefits of this implementation may not be initially obvious behind the layers of abstraction. The main benefit of this implementation is that there is no need for predefined grid dimensions. This allows for practically infinite sized grids, as long as the indices are valid ints and there is available memory. Grid data can be positioned anywhere without having to specify an offset as in other grid implementations. The benefits described above also come at a large cost. The set and get methods of the map$< int, map < int, map < int, float >>>$ sparse grid are O((log n)3).

Red-Black Tree

Since the C++ STL map is not something that can be used in an OpenCL kernel, one of the potential avenues pursued by the authors was creating a sparse grid with an underlying data structure that would be compatible with OpenCL and also improve upon either the render time or the memory overhead. It was decided that the underlying data structure for storing non-empty grid points would be a binary search tree. The key for organizing the nodes of the tree is the $i + j * X + k * X * Y$, which is unique for each grid point. Binary search trees insert smaller keys into the left sub-tree and larger values into the right sub-tree. It is very possible that a user of the grid class would start at grid point (0,0,0) and loop over the acceptable values of i,j, and k to initialize their grid. This potentially leads to the worst case scenario for the tree, in which it essentially becomes a linked list of right sub-trees. Since this possibility could easily occur in use, it was decided to use a self balancing tree, specifically a red-black tree. While this may incur a hit in inserting new nodes, this should decrease the look up time, especially in the worst case. The red-black tree implementation of a sparse grid was written in C. Instead of a C++, a struct was used, and all methods had an additional SparseGrid * parameter. Notice that this implementation (as well as all sparse implementations) does introduce memory overhead for each non-empty value added to the grid. Pointers to the left and right subtrees and a key value must be stored for each float to be stored in the tree. This overhead means that for grids where greater than 25% of the grid points are non-empty, the sparse grid implementation is actually less memory efficient.

Block-Partition

The method for sparsely storing voxel data described here is similar to the scheme used by Field3D 2. This implementation partitions the grid in into cubes of one or more voxels. Often the grid is partitioned so that each block (the partition size) is a power of two in each dimension 4. This is implemented as an array of float* all originally set to null. When an element is to be set at a certain grid point, the correct block is found. If this block has not been allocated, the entire block containing the grid point will be allocated as an array of size $(partSize)^3$. The value is then set at the appropriate index in the appropriate block array. If the block has already been created, then the only step is having the correct element of the array is set. Looking up a value

involves indexing to the correct block. If this block is null, the default value should be returned. If the block is allocated, the correct value within the block should be returned.

Unlike previous spare grids implemented, the block-partition method does not attempt to only store filled voxels. Block-partition instead attempts to use the inherent spatial locality of particles within most grids. In the entities we are often trying to render, like clouds, if a specific voxel is filled, it is very likely that most if not all of its adjacent voxels will also be filled. This property does not hold along the outer edges of the entity, and it is possible there are non filled areas within the entity, but for the majority of particles spatial locality should apply. This coupled with the expensive cost of dynamically allocating space makes building grids using this algorithm generally faster than other implementations. Implementations like those using the STL maps and red-black trees carry a high cost in terms of the amount of data necessary to get to organize the grid (ex: the pointers for the left and sub-trees). In block-partition method, only one extra value is created for every $partSize^3$ voxels. While not all of these voxels may be used, the majority of them will likely be used for most volumes. It is not unlikely that for many sets of data the block-partition method may be more memory efficient than other sparse implementations discussed. Another very tangible benefit of this scheme is its handling of dynamically allocating memory. The first grid point to be set in a block should be about as slow as other implementations since the block must be allocated before the value can be set. However, later values placed into that block will be inserted almost as quickly as in the dense grid (there is some extra computation involved in finding the correct block), since the space has already been allocated. This is one of the ways that this implementation takes advantage of the spatial locality of most data. The computational complexity of setting and getting values in partition-block grids is O(k). This is the same order of complexity as dense grids. The multiplicative constant for dense grids is lower, so sets and gets on a dense grid will still run faster than the block-partition grid.

Comparison

With five different implementations of grids written, we were interested in seeing how these grids would work in use. The two important metrics of sparse grids are time and memory usage. Multiple grid sizes were used in testing; the tests were repeated with both a 256^3 grid and a 512^3 resolution

grid of a wisp The grids contained only floats, no color element was taken into account. The rendered 512^3 grid used for testing.

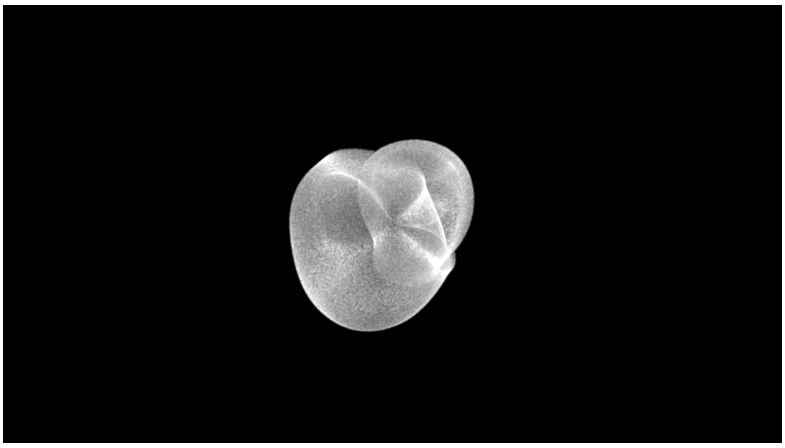


Figure 2: The rendered 512^3 grid used for testing

To get a sense of how long each grid takes to set and access data, the time it took for each implementation to build a grid and iteratively get every point in the grid was measured. Loading a grid is a test of the speed of each grids set method. Iterating over a grid in turn gives an idea of the relative time it takes each grid to look a value up. This test was repeated fifteen times, one immediately after the other. The results of the first five runs were discarded, since they almost always tended to be much less consistent than the latter runs. The load time and iteration time of the next ten runs were then averaged. The results of this testing were displayed below.

Several trends are worth discussing from this chart. Immediately apparent is the time inefficiency of the 3 nested STL maps implementation. This is not unexpected given its high computational complexity compared to the other grids, but its load and iteration times almost double that of the next slowest implementation. Unless the added benefit of not having to predefine maximum grid dimensions far outweigh the time costs displayed here, this implementation is not of much practical worth. The single STL map and the red-black implementation achieved similar results, which is not surprising. Maps like the ones found in the STL can be implemented using tree structures, and the two implementations were both O(log n) for setting and getting voxel data. The red-black tree performed slightly better, but this could potentially be attributed to a number of things like differences be-

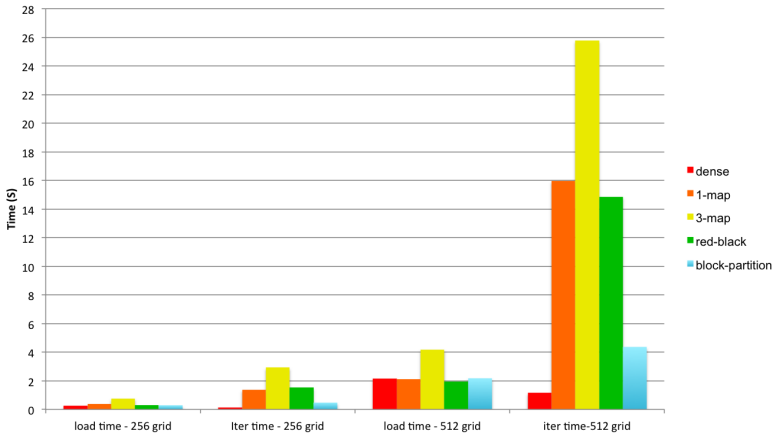


Figure 3: CPU Grid Implementations - Time

tween the gcc and g++ compilers, extra overhead by the templates used in STL maps, or that the red-black implementation was written specifically for grids. This chart also makes it clear that sparse grids take the bigger time hit in accessing data. The sparse grids (nested map excluded) all performed about as well as the dense grid when it came to loading data. The dense grid iterates over both lists faster than it loads them. The opposite is true for the sparse grid implementations. The block-partition grid, which is clearly the most efficient sparse grid from a time standpoint, is twice as slow in iterating over its data as loading it. This means that getting a value is about four times slower for block-partition grids than for a dense grid.

Time is only one metric to analyze sparse grids. The other important metric is memory usage. To track memory usage, a 256^3 and a 512^3 grid were again loaded into memory. Valgrind, a tool that tracks memory allocation of programs, was used to see how much memory was being allocated in the process of loading each grid. The results are displayed below.

It is obvious that all of the sparse grids are much more memory efficient than the dense grid at storing data. Interestingly, the most efficient sparse grid (for these two images, which are representative of the majority of images rendered) is the block-partition grid, which does not attempt to contain only non-empty values. This is likely the case because the overhead incurred by the STL map and red-black tree implementations place a far greater burden on memory than the array of pointers to blocks and the unused but still allocated voxels.

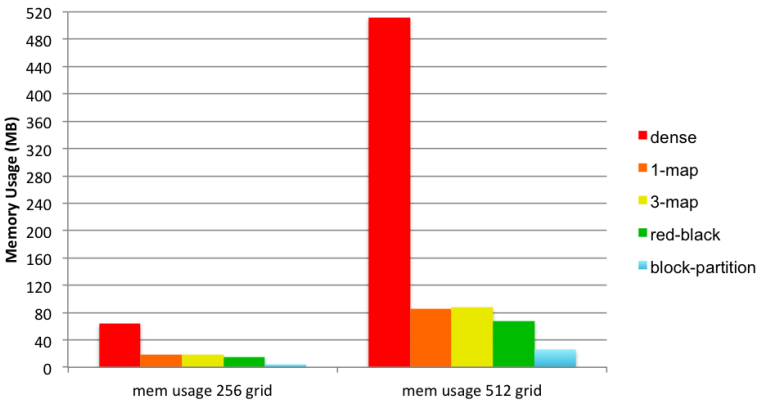


Figure 4: CPU Grid Implementations - Memory

Based on the tests we ran, the partition-block scheme of sparse grids is the clear winner. It is significantly faster and more memory efficient than any of its sparse counterparts. It is also clearly the most scalable, which is an important attribute for these grids considering that their purpose is in rendering larger grids of data. It is for these reasons that the sparse grids used in our volume renderers use block-partition grids.

4 OpenCL implementations

Non-sparse

To serve as a foundation for building volume renderers utilizing sparse grids, we started with volume renderer with a C++ host that set up the volume to be rendered and the appropriate OpenCL setup. The renderer used two OpenCL kernels. The first kernel calculates and builds a deep shadow map for the volume. The second kernel performs the ray marching 5 and creates the image data, which is then returned to the host and written to an image file in the requested format. The shadow map kernel was removed during development to allow for more space in GPU memory.

Single Level of Abstraction

The original sparse grid implementation of a volume renderer in OpenCL features a system based on the partition-block implementation described above.

The grid is built in the C++ code and loaded onto the GPU to be used by the kernel. Due to the nature of memory on a GPU, one major deviation is made from the C++ implementation. The C++ implementation kept an array of pointers to blocks of grid points, initially set to null. When a grid point was being set within a null block, space for the block would be dynamically allocated and the array of pointers would then point to their respective blocks. The problem with this implementation is that when the array is loaded into the graphics cards memory, the values of those array indexes are pointing to memory locations in a totally different address space, in main memory. To fix this problem, the initial array of pointers is replaced with an int array of the same size. Initially all values in the array are set to -1. Using the same indexing scheme described to index into the array of pointers with a given (i,j,k), whenever a location is accessed for the first time, its value is changed from -1 to one less than the total number of locations indexed to so far. The first location accessed will be given a value of 0, the second location accessed will be given a value of 1, etc. These new values will be used to index into a second array of floats containing the actual values. Each time a new block is indexed, the float array will have to be reallocated with partSize3 more floats to accommodate a new partition. All of the newly allocated floats must be initialized to the default value.

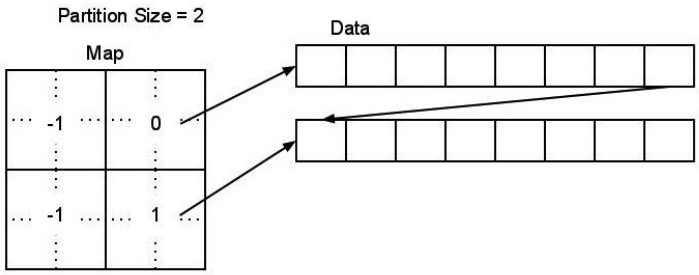


Figure 5: A $4x4x2$ grid with a partition size of 2

Once both arrays are built in this fashion, getting a value from the grid is a two step process. The first step involves looking up the indexed int array value. If this is -1, then the default value is immediately returned. Otherwise, the array value is used to index into the correct partition in the float array, and then the correct value within the partition is returned. This difference allows the spare grid to be loaded and correctly work on the GPU. The

sparse grid was updated so that OpenCLs float4 type could also be stored at each data point. A separate array of float4s was kept and values were added and gotten identically to the float array. In the volume renderer the first three values of the float4 stored RGB color information. The original volume renderer using dense grids was heavily modified to accommodate the sparse grid implementation. Aside from trivial API differences between the grid classes on the C++ side, the majority of the changes to the C++ code involved getting all of the extra data needed by the sparse grid loaded onto the GPU. The sparse grid needs the mapping int array, the partition size, and the default values for the float and float4 arrays in addition to what is required of the dense grid. The kernels were altered to handle the additional arguments and the sparse data. The original dense grids relied on direct access to the arrays and were rewritten to correctly access the appropriate data.

Multiple Levels of Abstraction

Most of the overhead associated with the block-partition scheme hinges on the size of each partition. Larger partitions mean a smaller int array for mapping, but also means that more potentially unused grid points are being allocated. Conversely, depending on the size of the grid, it is possible that making the partition size will cause the total memory usage of the system to go up because of the increased size of the mapping array. In addition, it is possible that many of the integer blocks will be unused, containing a default value of -1. One way to alleviate this issue is to extend the idea of sparse grids so that the mapping array is itself sparsely stored into partitions. Adding an additional array of ints adds a layer of complexity both conceptually and computationally, but given a sparse enough grid the improvements in in memory usage are substantial. When setting a value at a grid point, the class must first check if the correct mapping partition has been allocated, allocate if necessary, and then do the same with the data partition before setting the value. Value look up also has the extra step of checking if the mapping partition has been allocated. The doubly sparse grid implemented here has the same partition size for both levels, though a natural extension might be to have separate partition sizes for each level.

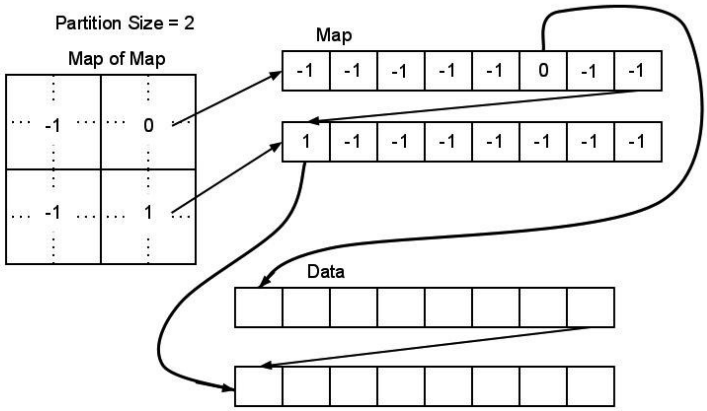


Figure 6: An $8x8x4$ double sparse grid

5 Results

Optimization

Several improvements were made to the two sparse renderers after achieving basic functionality. In the original version of the sparse renderer, only grids in which each dimension was a multiple of the partition size would correctly renderer. The double sparse renderer, in turn, would only correctly render with dimensions the multiple of the square of the partition size. Grids not fitting these requirements would produce images with noise around the edges. This noise was the result of the extra space created by dividing the grid into blocks; since the grid does not evenly divide into the blocks, a subset of the blocks will contain space for data that should not exist. This extra space would cause incorrect some indirect indexing into the sparse grid and thus distort the image. This was resolved by forcing non-multiple grid dimensions to the next highest multiple (a 101^3 grid will be forced to a 104^3 grid for single sparseness), ensuring the image produced will never have this issue.

Other changes were made to the renderers to improve performance in terms of both speed and memory usage. The deep shadow map kernel and its associated grid were removed from later versions of all three renderers. This change decreased both the render time of the image and the memory usage of the GPU since the shadow map data no longer had to be loaded on the card and there were fewer computations to perform in the ray march kernel. During the development and refinement of these renderers, we found

that a partition size of four seems to produce optimal results for the test volume. The optimal partition size varies between volumes and the value that seemed the best compromise between render time and memory size.

Testing

Memory

In addition to creating an image, each renderer returns pertinent information like the partition size used, amount of time spent on the GPU, and the number of blocks allocated (for the sparse grids). From this data it is easy to calculate the amount of memory necessary to successfully create and store a grid. To compare the three renderers, the same image was rendered on each renderer. If a render was successful, the grid size was increased in each dimension by 250 voxels. When a grid gets too large for the GPU to store, an allocation failure flag is thrown and the render terminates. The results of testing are shown below. The results discussed below were generated by an Nvidia GeForce 9600M GT with 256 MB of memory. Additional tests were run on WHATEVER CHORTLES CARD IS and comparisons between the gathered data can be found later in the text.

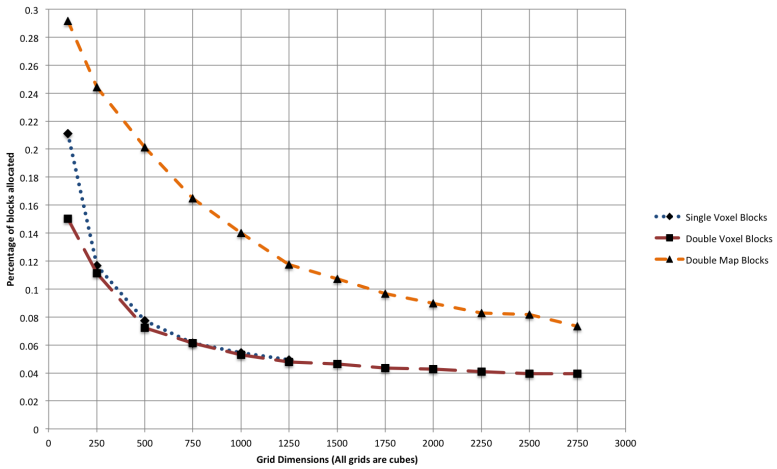


Figure 7: Voxel Allocation in Sparse Renderers

The non-sparse implementation was only able to successfully render a 100^3 grid before exhausting the memory. The renderer using single sparse grid

was able to successfully render grids up to 1250^3, over three thousand times larger than the non-sparse renderer. The double sparse grid implementation achieved 2750^3, six times larger than the single sparse renderer and over twenty thousand times larger than the non-sparse renderer. The sparse grid renderers are able to hold much greater volumes because they only allocate a very minute percentage of the voxels in the grid (< 0.1 % in most cases). As a result, grids can be orders of magnitude larger. There is also a substantial gain in the additional layer of sparseness added for the double sparse renderer. Notice that the percentage of voxels allocated is much closer between the two sparse renderers than between either sparse renderer and the non-sparse one. Often, they differ by only a few thousandths of a percent. This would seem to imply that the majority of the savings are coming from making the mapping sparse as well. As the number of allocated map blocks reported from the double sparse implementation indicates, the vast majority of blocks are empty in the single sparse implementation. Significant memory savings are made by making the map sparse as well. These savings only increase with the size of the actual grid, since the number of allocated blocks in the double sparse implementation stays roughly the same across grid sizes.

Render Time

The significant gains made by the sparse grid implementations in terms of memory footprint do come at the cost of render time. The non-sparse renderer may store all of its values, but because it does so, looking up a density value is as simple as directly indexing into an array. The single sparse renderer must perform a set of operations to see if the desired voxel has been allocated and if so what its values are. Having to perform these additional operations means a slower render. The double sparse renderer must essentially perform this set of operations twice to look up a value, and as a result is noticeably slower than even the single sparse render.

Several trends, however, do appear in the data. Immediately clear is the fact that there is an inverse relation between grid size and render time for sparse grids. This relation is especially pronounced in the double sparse implementation results, which decrease by approximately 40% over the range of grid sizes. In smaller grids, render times are larger and there is a large change in render time between grid sizes. As grid sizes get larger, render time starts to decrease and there is less of a difference between the render time of different sized grids. These results may initially seem counterintu-

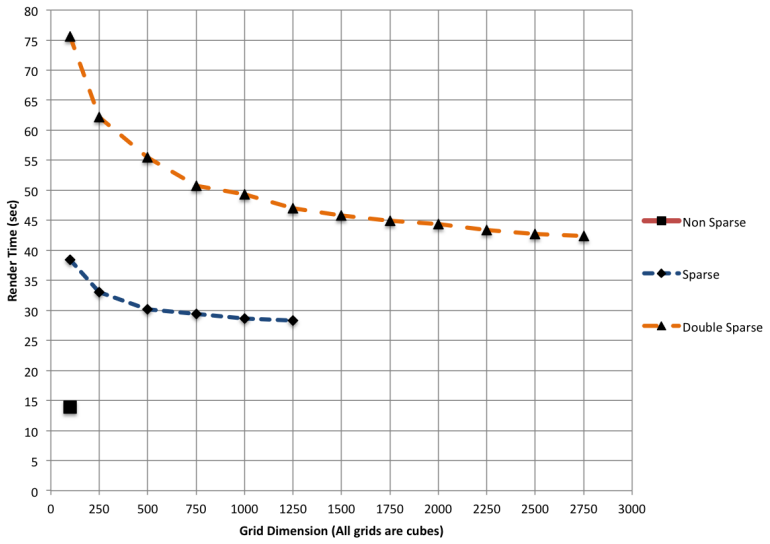


Figure 8: Render Times Using Different Grid Sizes

itive. Conventional wisdom would seem to imply that as grid size increased, so too should the time it takes to render the grid. Notice that render time trends are similar to the trend displayed percentage of map blocks allocated as grid sizes increase. Both start large with a fast rate of change and end with smaller values and a slow rate of change. A smaller percentage of allocated blocks with increasing grid size implies that the rate of filled blocks in the grid is increasing at a slower rate the number of unfilled blocks. In other words, the majority of blocks added with an increase in grid size will go unallocated. While this is not necessarily desirable from a memory management perspective, it is actually a source of speed up when rendering a volume. If in the course of the look up function, the look up function finds that the desired voxel is not allocated, the look up function immediately returns the default value. This means that the majority of the computations needed to actually find the correct value are skipped, and the lookup function is much faster as a result. This speedup, combined with the parallel nature of graphics cards, leads to the shorter render times displayed in the double sparse implementation results. This same basic principle also applies to the single sparse renderer.

Card Scalability

CHORTLE RESULTS WOULD GO HERE

Several tests were run also run on WHATEVER CHORTLES CARD IS with MEM MB of memory. While the results clearly show that this is a more powerful card that can store much larger sparse grids, the trends remain largely the same. The benefits of these grids translate with more powerful cards.

Other

These results show that for many volume rendering applications in which memory is the limiting factor, sparse grids are valuable tools that can greatly increase the functionality of said application. If the volume needs to be rendered as fast as possible, or if the volume is mostly filled, sparse grids are probably not appropriate. Our results show that a single layer of sparseness allows grids with orders of magnitude more voxels for a relatively modest increase in render time. These memory savings come occur because only a minute percentage of voxels in the grid are actually allocated. More time is needed to render because additional computations are needed to correctly index to the allocated voxels. Eventually memory in a single level of sparseness the limiting factor becomes the size of the map used to index to allocated voxels rather than the voxels themselves. This limitation is solved by making the map itself sparse, creating a double level of sparseness. This double sparse renderer is slower than its less sparse counterpart, but also can render much larger volumes. With increasing grid size, these sparse renderers actually take less time to render. This is because the vast majority of voxels added by the increased grid size will be unallocated. Unallocated voxels can be looked up much faster than allocated voxels, which helps speed up the render. We recommend that a volume be rendered with the lowest degree of sparseness possible. Rendering an image using a sparse grid which can rendered with a non sparse renderer only increases the render time without real benefits. The value in sparse grids is in rendering grids so large they could not normally be rendered.

6 Conclusion

There are many possible methods of reducing the memory size of large volume grids. We implemented sparse grids using C++ STL maps, red-black trees, and block-partitions. Overall, block-partition sparse grids seem to be the most efficient method of storing volumetric data. Implementing block-partition grids in OpenCL allows large volume grids to be rendered on GPUs, which typically do not have as much RAM as CPUs. Using block-partition grids reduces memory usage but increases render time. Potential future work involves the use of texture memory on the GPU. Texture memory provides faster access time than global memory. To use texture memory for sparse grids, we envision writing the one-dimensional, sparse RGBA data to the two-dimensional texture memory. We would choose an texture width that is some power of two in order to speed up indexing calculations. The logical extension of what has been presented in this paper would be to design a triply sparse grid. This could be useful for volumes that are extremely large but mostly empty. In general, however, there will be diminishing returns on adding extra levels of sparse mapping. Every extra layer of sparse mapping increases data access time significantly. Another potential improvement would be to find the optimal partition size or sizes computationally on the host before sending the data to the kernel. Generally, a partition size of around 4 is most memory efficient, but in some cases a larger partition size might save memory. We could develop an algorithm that analyzes that volume grid on the host and determines which partition size would result in the least possible memory usage. A drawback is that partition sizes that are powers of 2 make indexing calculations much faster. Using a partition size of 5 instead of 4 might save GPU memory, but it could double render time. Our current implementation coalesces a block into a single value only if every element in the block is exactly equal to one particular value. In other words, if the default value is zero, then a block of values will be allocated in the sparse grid if any of the values in the block are not exactly equal to zero. We could add a range of values that would be clamped to the default value. This would decrease memory usage but also decrease render quality.

1. Aaftab Munshi et al. "OpenCL Programming Guide", 1st ed. Upper Saddle River, NJ: Addison Wesley,2012

2. Magnus Wrenninge. (2011,11,15). Field3D: An Open Source File Format For Voxel Data[Online]. Available sites.google.com/site/field3d/

3. Cyril Crassin et al. "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering" in ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), Boston, MA, 2009 http://maverick.inria.fr/Publications/2009/CNLE09

4. Magnus Wrenninge et al. "Volumetric Methods in Visual Effects". SIGGRAPH 2010 Course Notes, pp. 6-70

5. Jerry Tessendorf and Michael Kowalski. "Resolution Independent Volumes". SIGGRAPH 2011 Course Notes

Appendix D Resolution Independent Volumes

This technical report was written for the Siggraph Course "Production Volume Rendering" (sig [2011]) offered in 2010 and 2011.

Resolution Independent Volumes

Jerry Tessendorf
School of Computing, Clemson University

Michael Kowalski
Rhythm and Hues Studios

October 5, 2016

i

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

This document can be found at https://people.cs.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

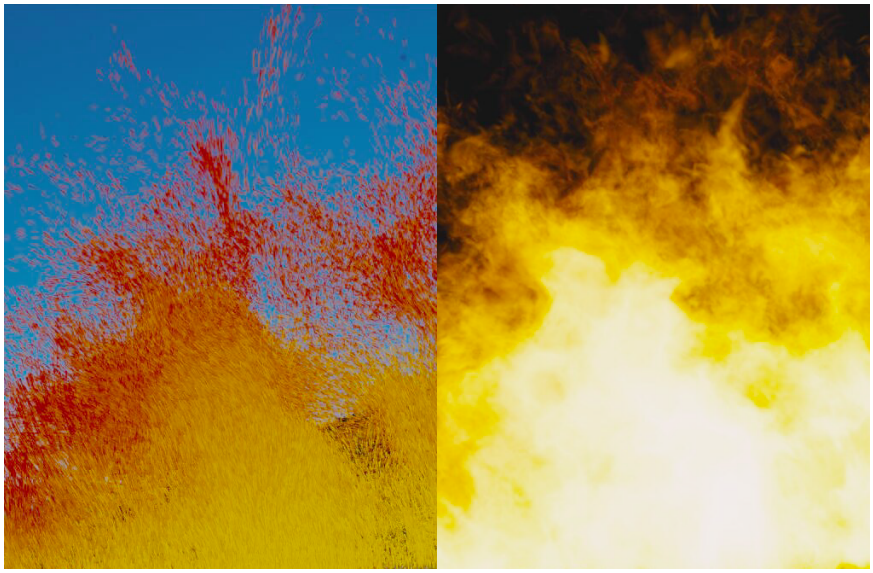


Figure 1: Early imagery showing the conversion of a particle system into a volumetric fire. The FELT algorithms used for this included early versions of gridless advection and SELMA.

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.

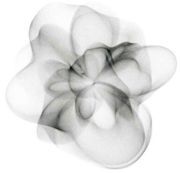
Cleaner coding of algorithms

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

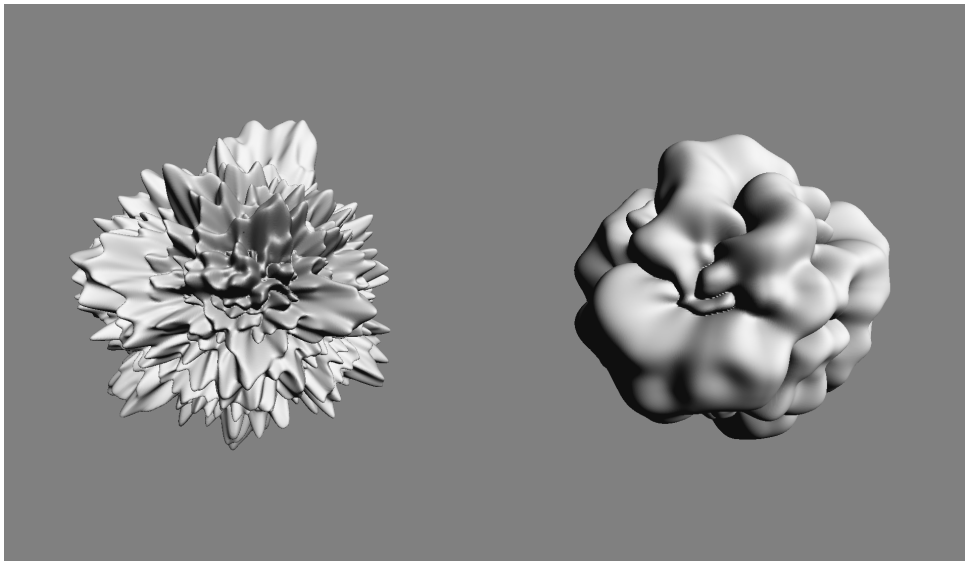


Figure 3.2: Examples of classic pyroclastically displaced spheres of density.

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:

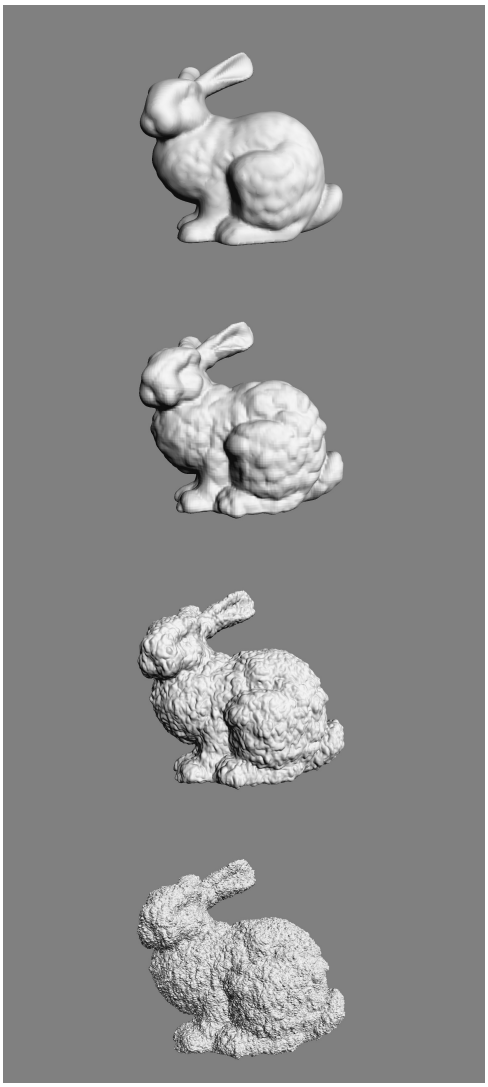


Figure 3.3: Illustration of layering of pyroclastic displacements. From top to bottom: No displacements; one layer of displacements; two layers; three layers. The displacements are applied to the levelset representation of the bunny, and the displaced bunny was converted into geometry for display.

```
// 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

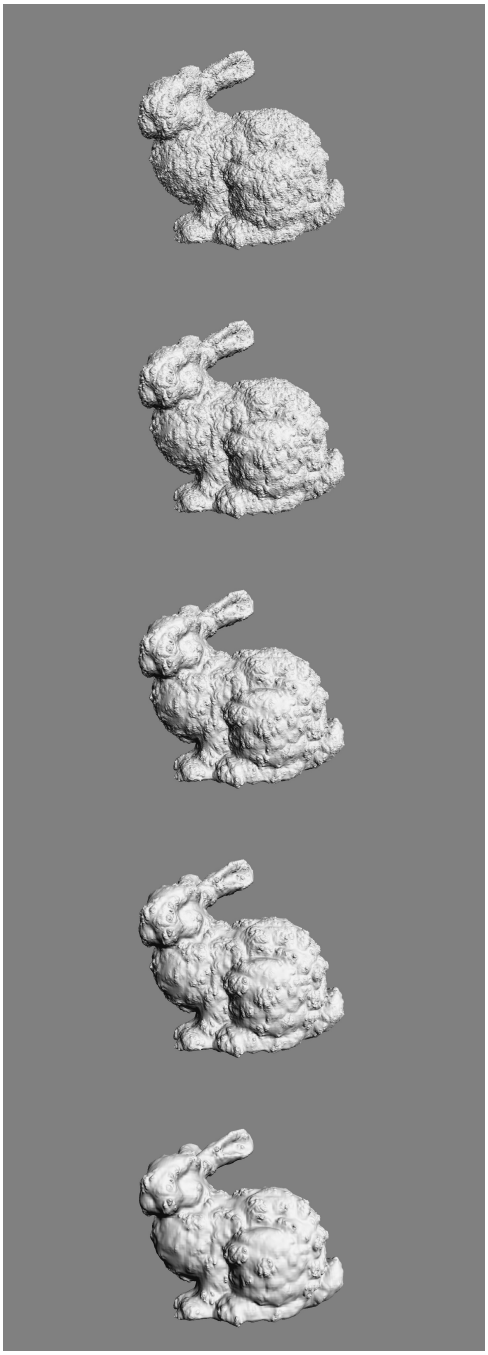


Figure 3.4: Illustration of clearing of displacements in the valleys using the billow parameter. The bottom of figure 3.3 illustrates the three layers of displacement with no billow applied. The noise is FFT-based, and $Q = 1$. From top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

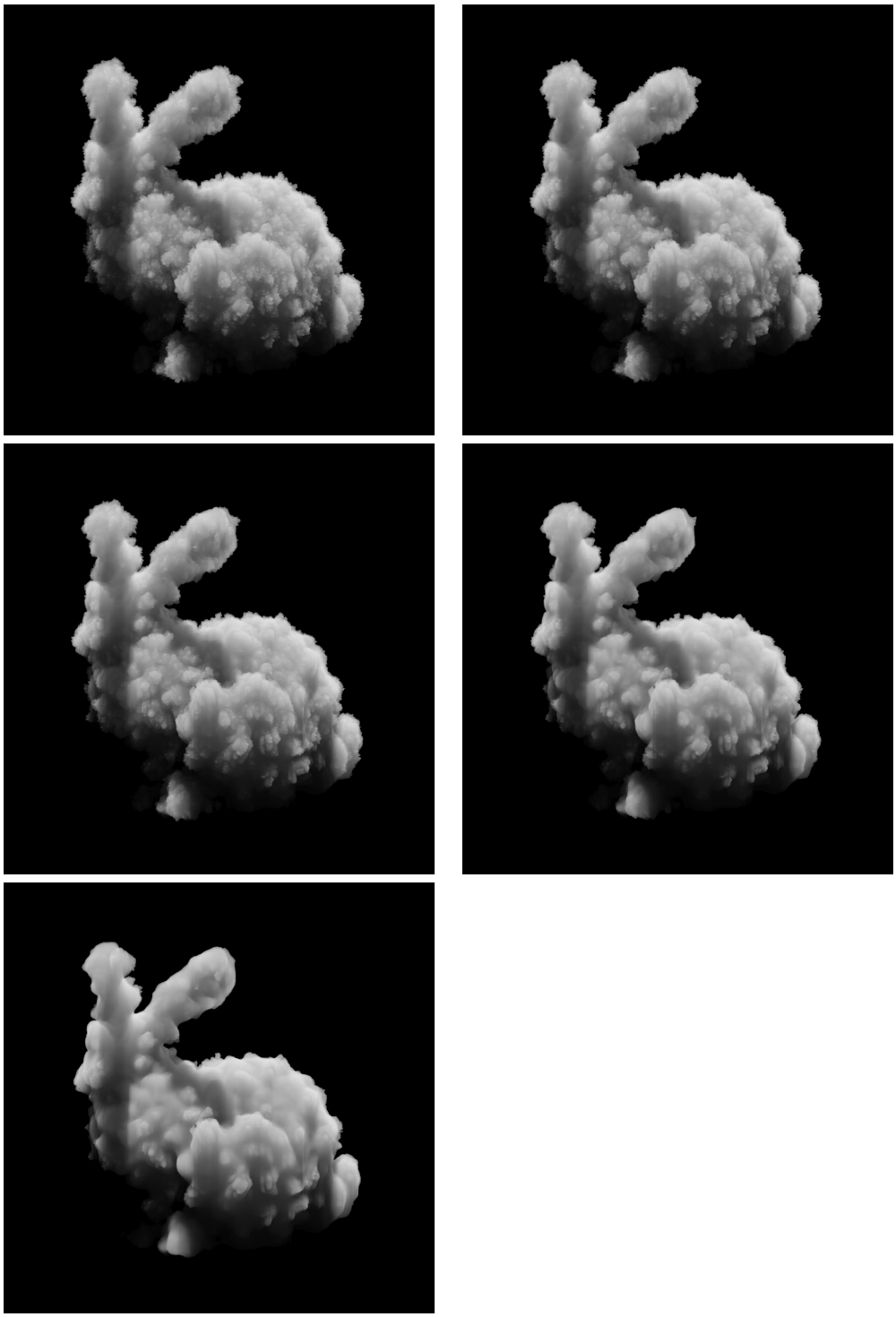


Figure 3.5: Volume renders with various values of billow. Left to right, top to bottom: billow=0.33, 0.5, 0.67, 1, 2.

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.

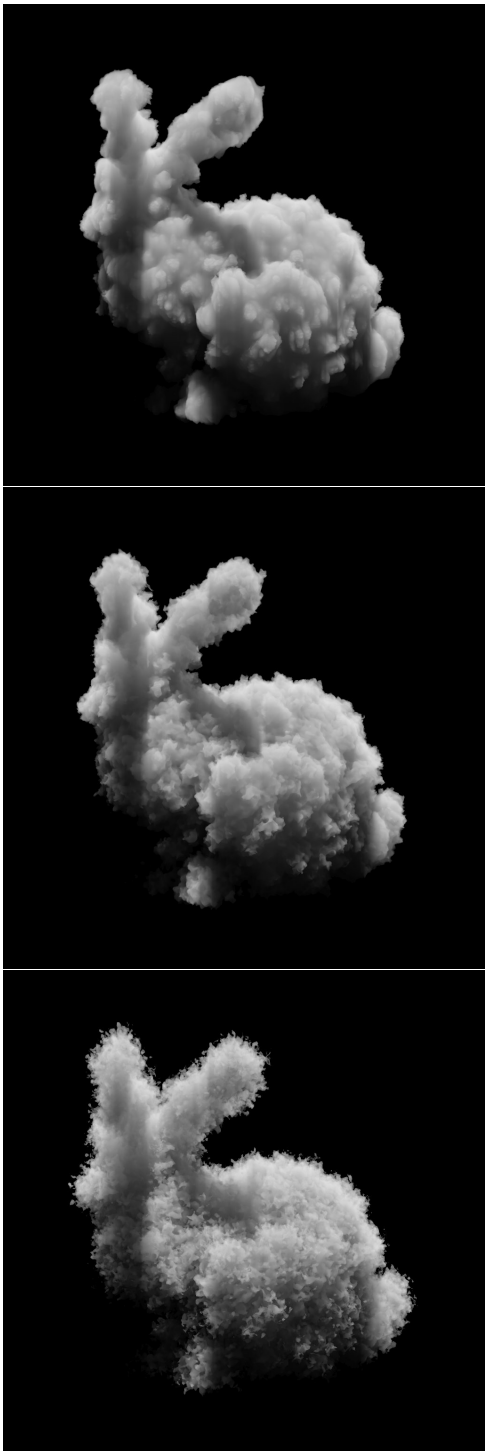


Figure 3.7: Volume renders with various setting of advection, for billow=1. Top to bottom: No advection, medium advection, strong advection.

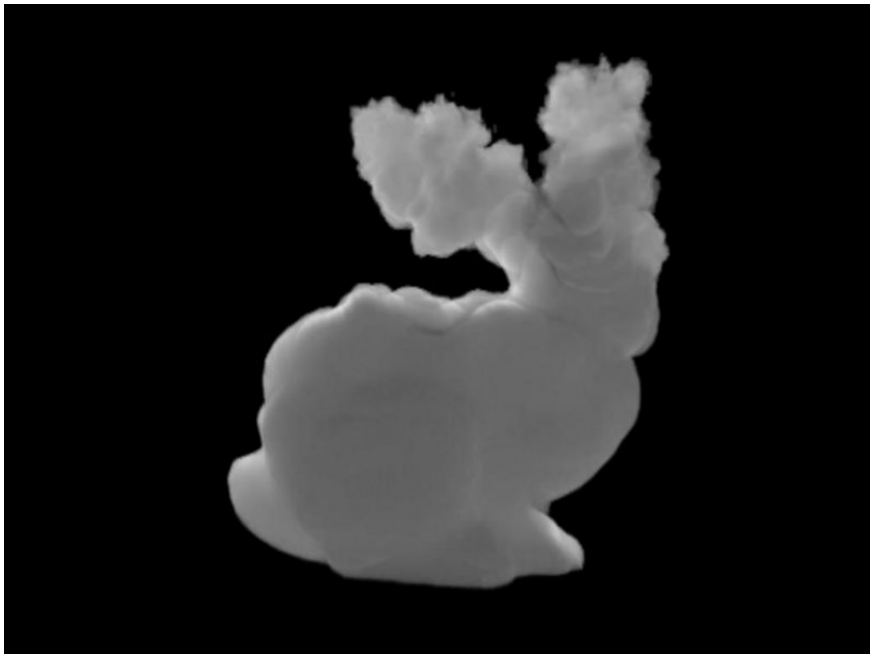


Figure 3.8: Volumetric bunny with spatial control over the pyroclastic displacement.

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.

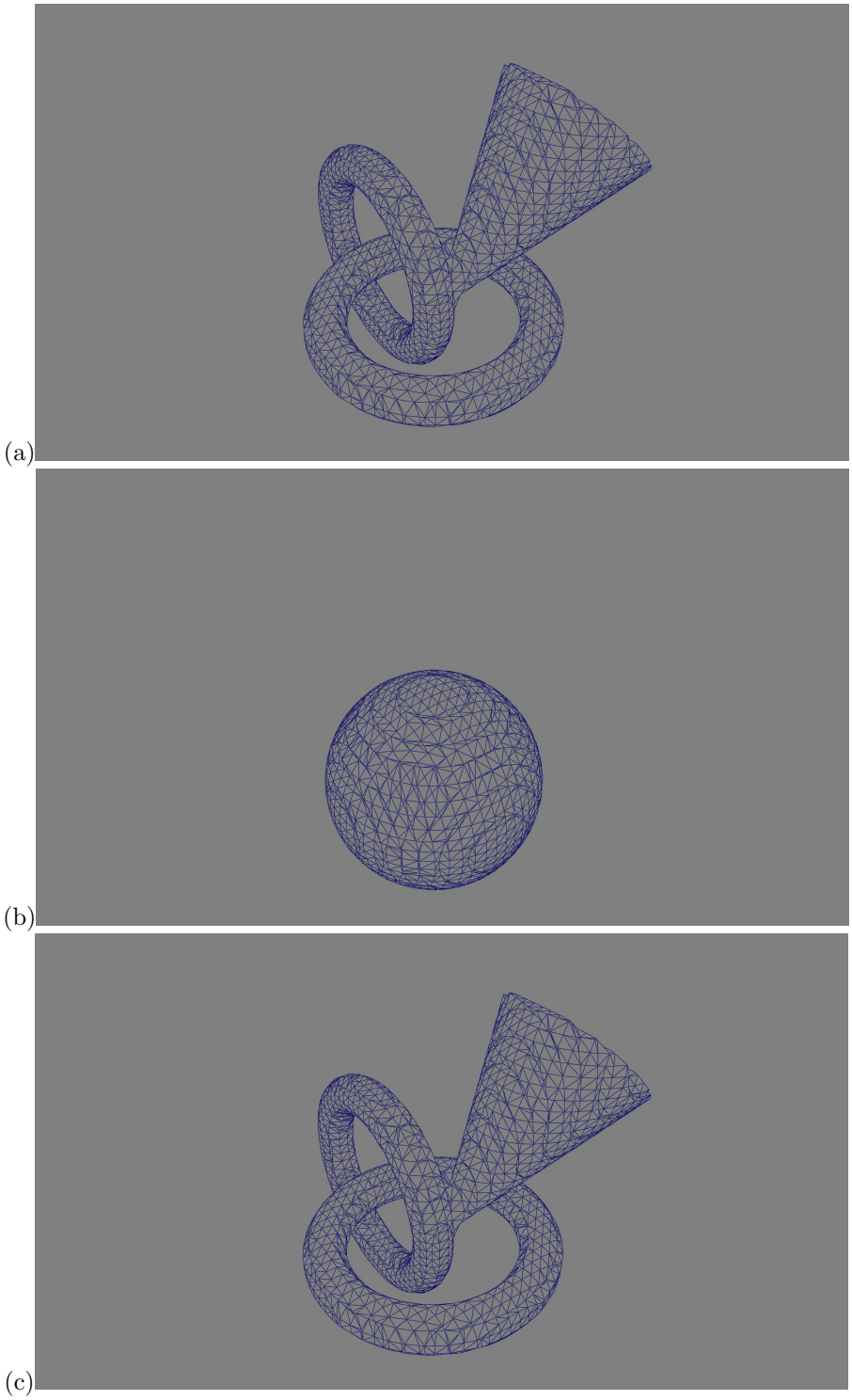


Figure 4.1: Warping of a reference sphere into a complex shape (cone and two torii). (a) Object shape; (b) Reference sphere; (c) Warp shape output from 1 iteration.

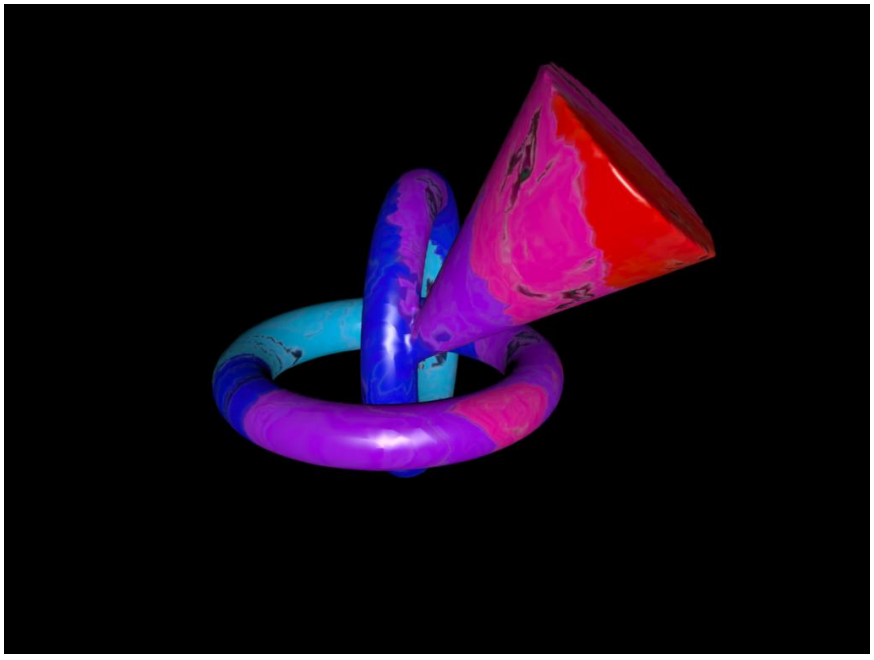


Figure 4.2: Texture mapping of the object shape by transfering texture coordinates from the reference shape.

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.

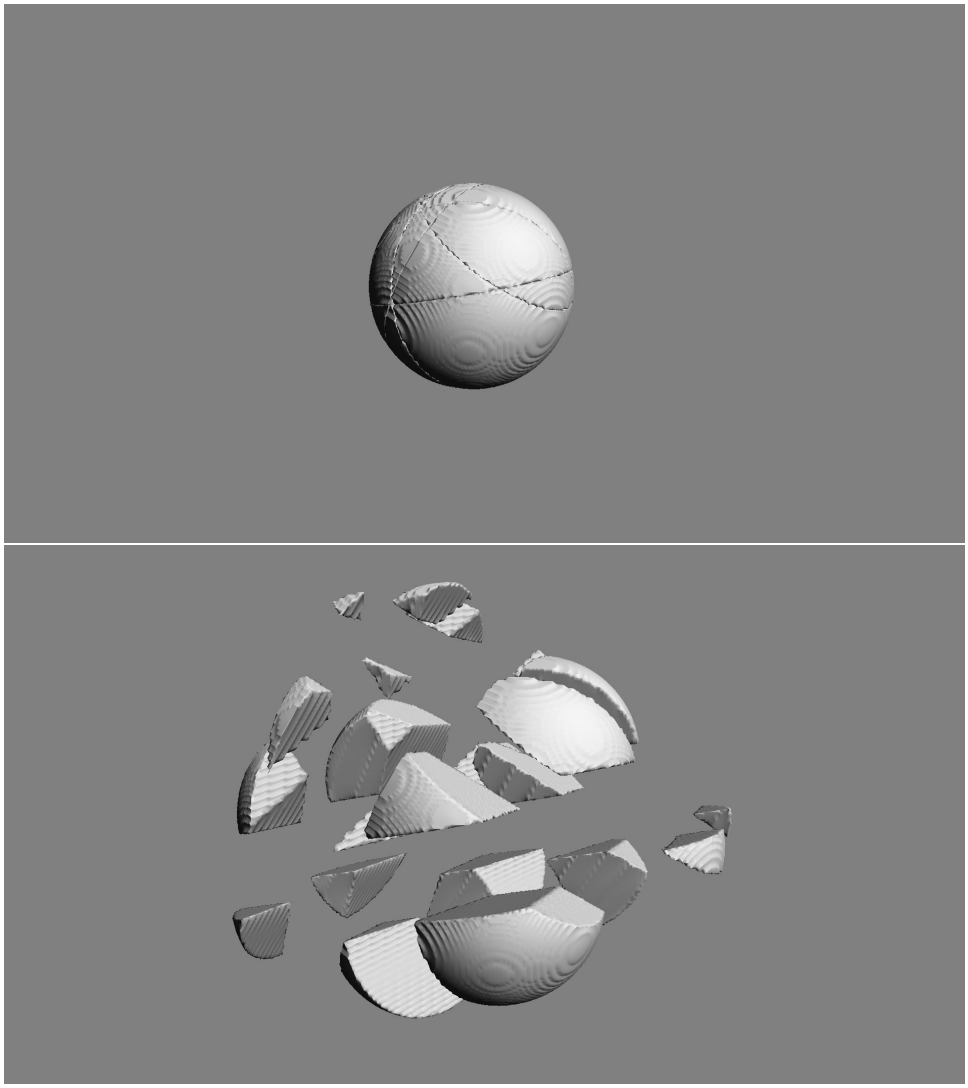


Figure 5.1: A sphere carved into 22 pieces using 5 randomly placed and oriented flat blades. The top shows the sphere with the cuts visible. The bottom is an expanded view of the pieces.

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 );
```

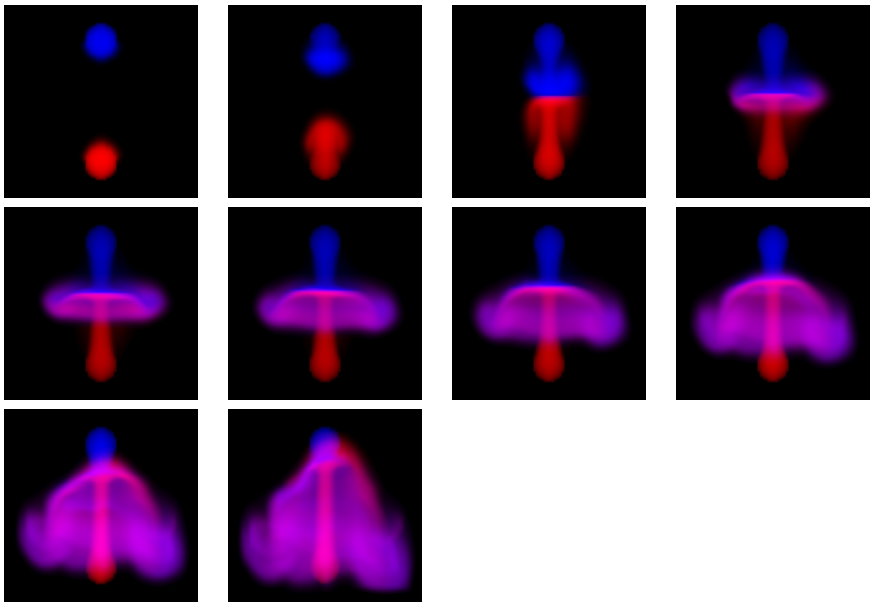


Figure 6.1: Simulation sequence for hot and cold gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

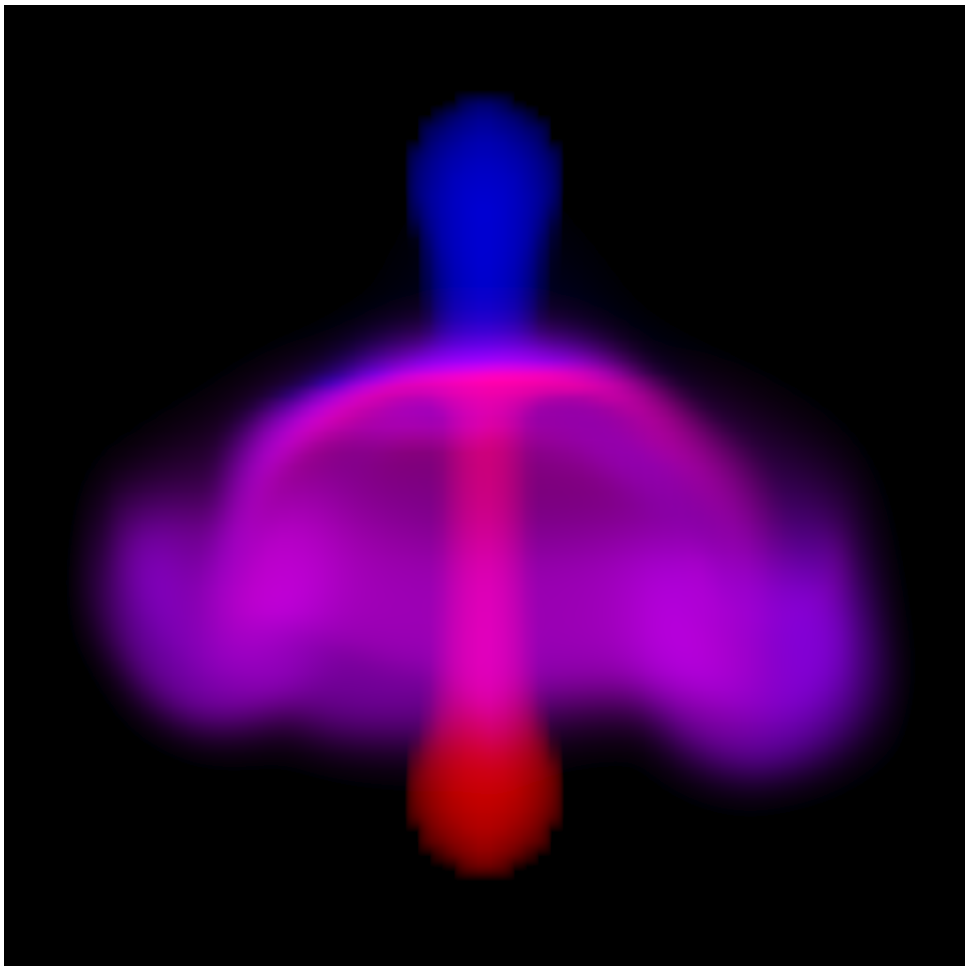


Figure 6.2: Frame of simulation of two gases. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution for all quantities is $50 \times 50 \times 50$.

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

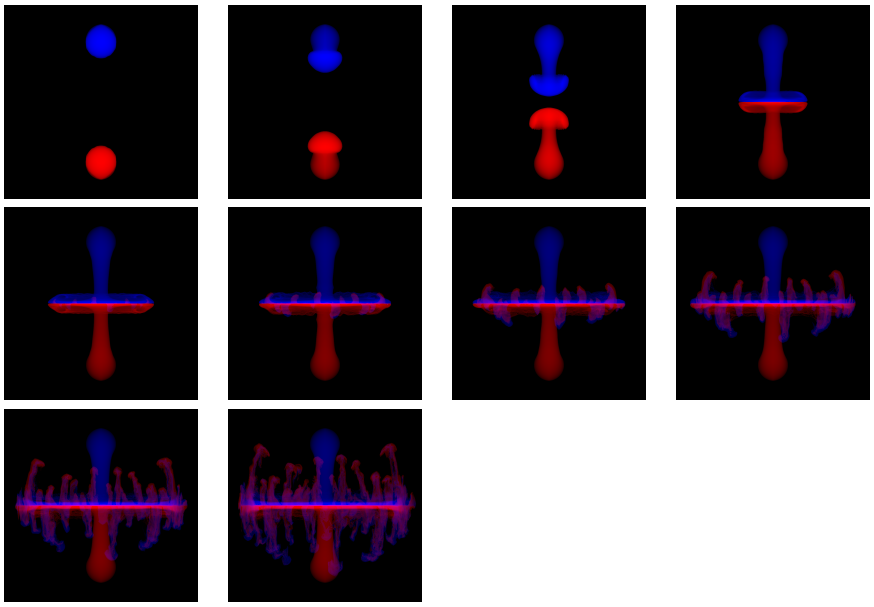


Figure 6.3: Sequence of frames of a simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected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

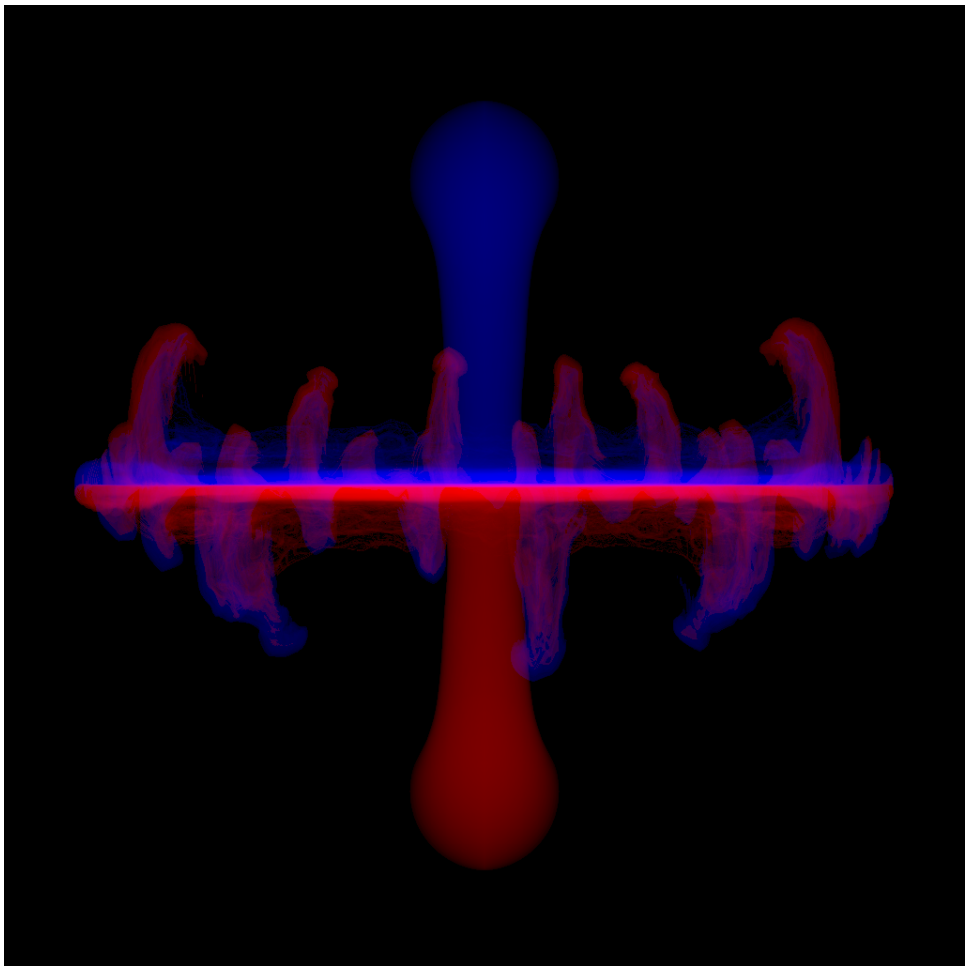


Figure 6.4: Frame of simulation of two gases, in which the densities evolve gridlessly. The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The density is advected but not sampled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

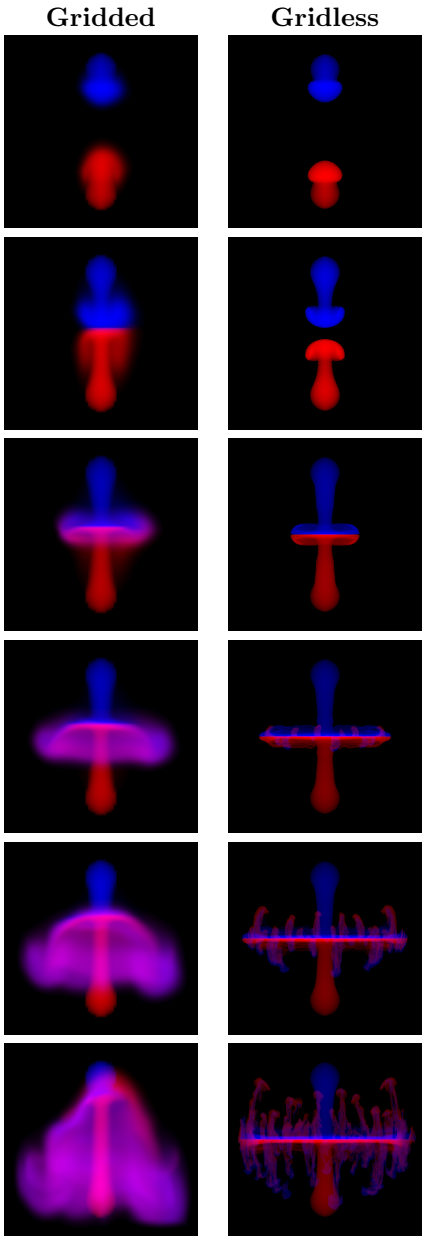


Figure 6.5: Simulation sequences with density gridded (left) and gridless (right). The blue gas is injected at the top and is cold, and so sinks. The red gas is injected at the bottom and is hot, and so rises. The two gases collide and flow around each other. The grid resolution is $50 \times 50 \times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

```
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.

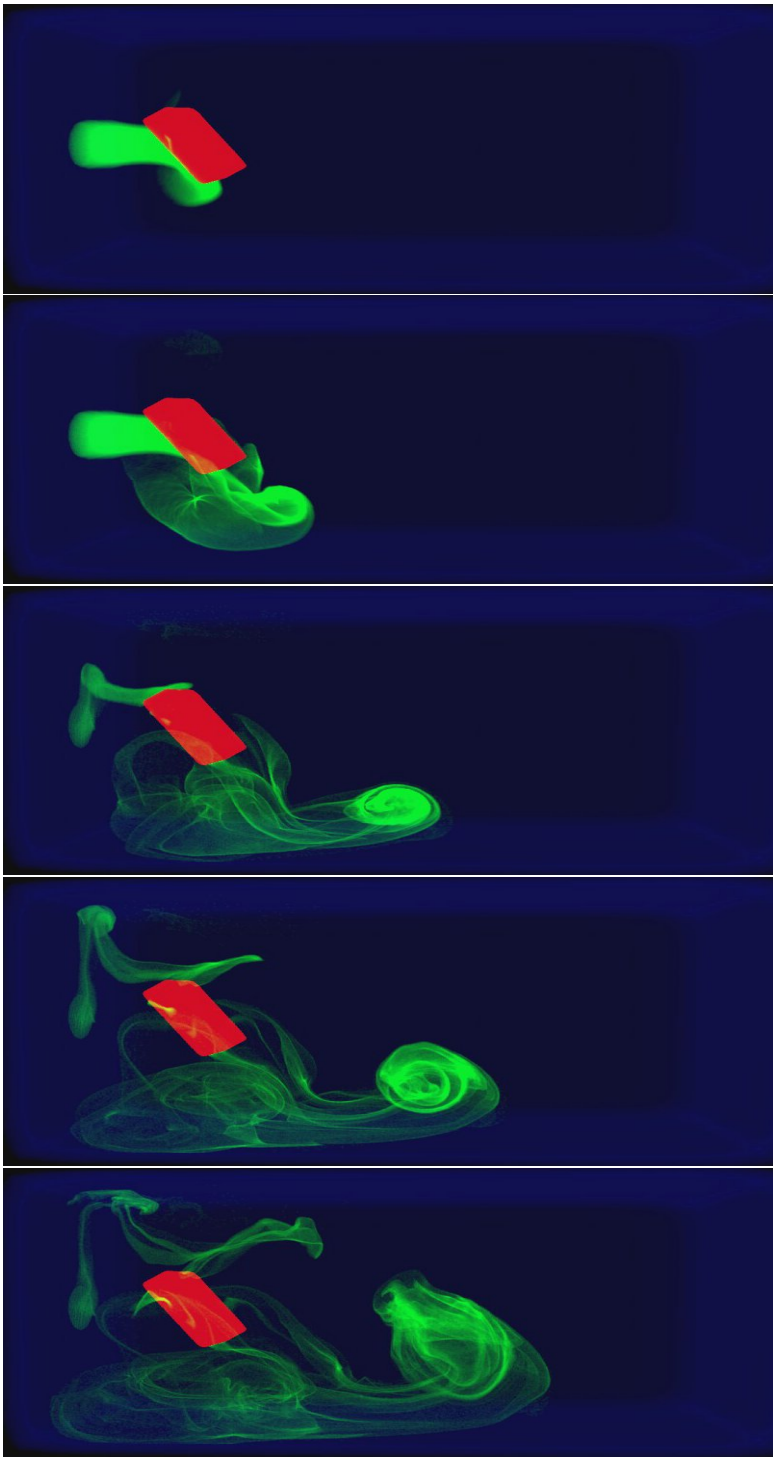


Figure 6.6: Time series of a simulation of bouyant flow (green) confined within a box (blue boundary) and flowing around a slab obstacle (red). Frames 11, 29, 74, 124, 200 from a 200 frame simulation.

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

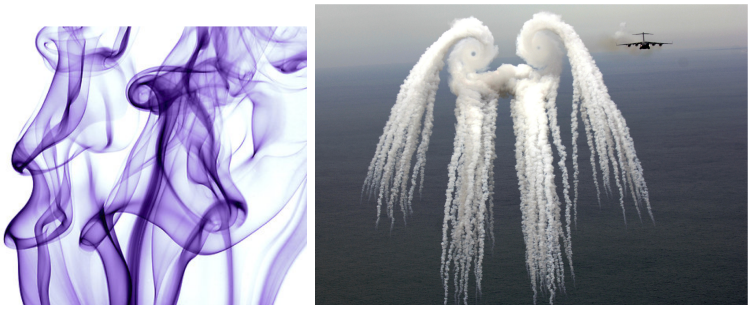


Figure 7.1: Examples of filaments and sheets forming in fluid flow.

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

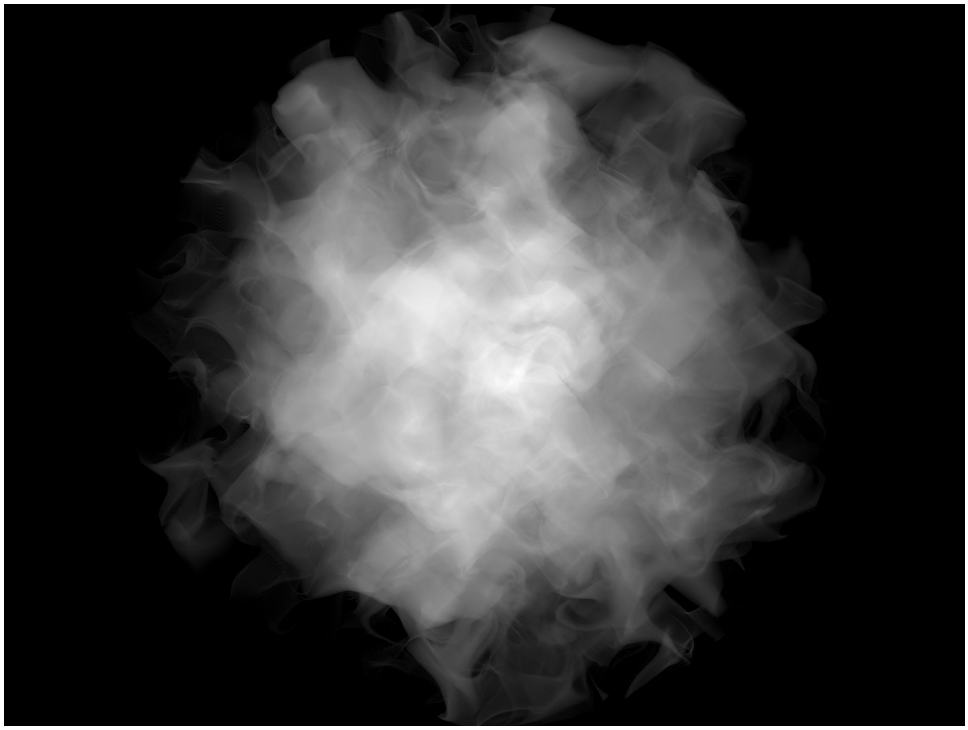


Figure 7.2: Illustration of the effect of a single step of gridless advection. The unadvected density field is a sphere of uniform density.

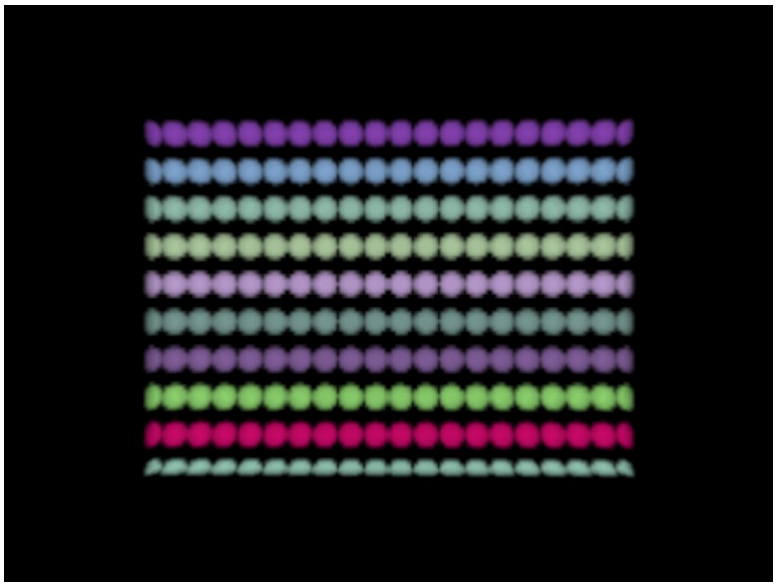


Figure 7.3: Unadvected density distribution arranged from a collection of spherical densities.

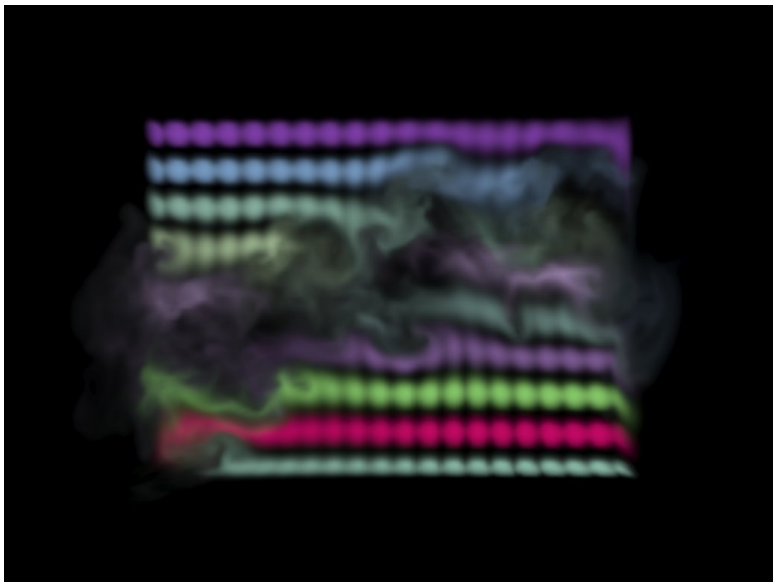


Figure 7.4: Density distribution after 60 frames of advection and sampling to a grid each frame.

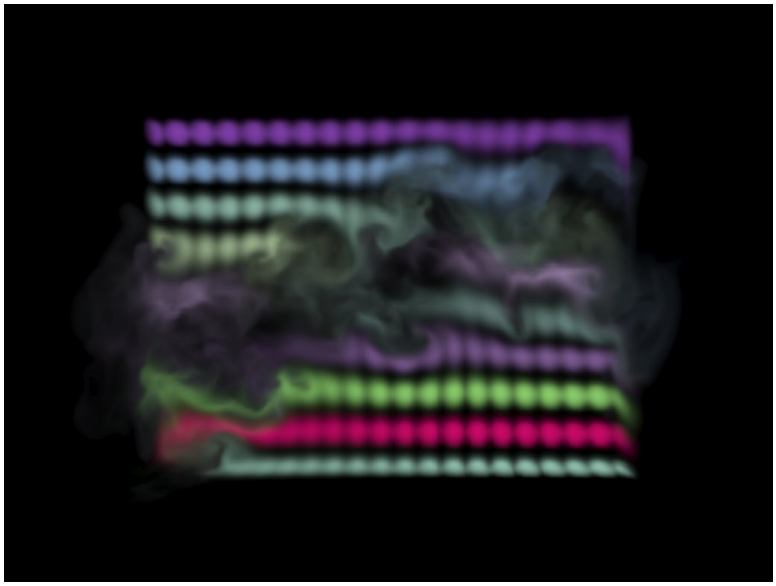


Figure 7.5: Density distribution after 59 frames of advection and sampling to a grid each frame, and one frame of gridless advection. The edges of filaments have been subtley sharpened.

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

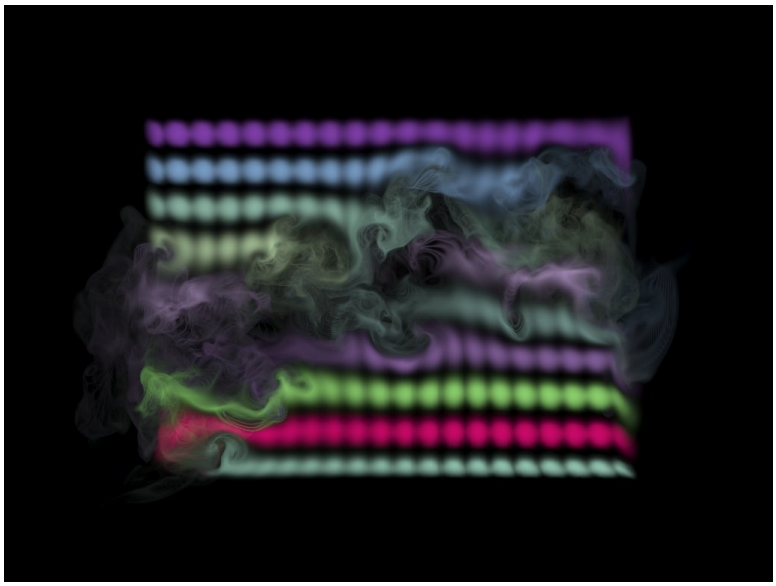


Figure 7.6: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The sharpening of details has increased to the point that the detail is finer than the raymarch stepping, causing significant aliasing in the render.

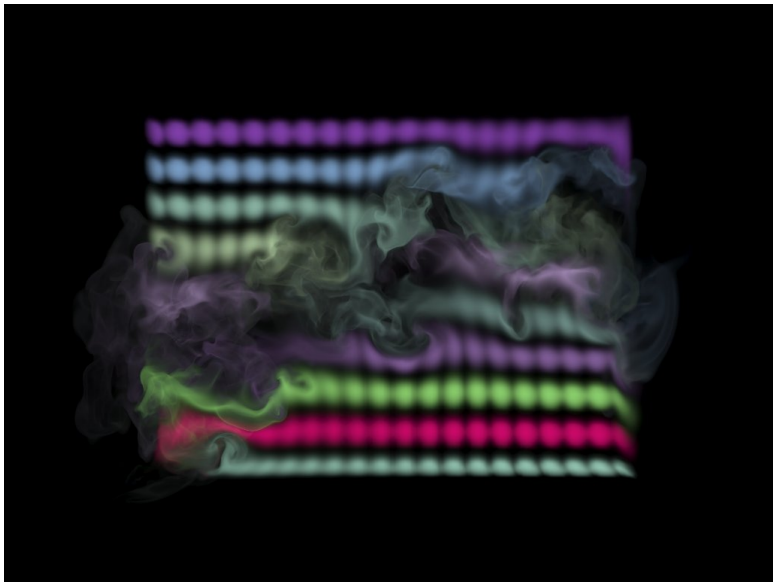


Figure 7.7: Density distribution after 50 frames of advection and sampling to a grid each frame, and ten frames of gridless advection. The fine detail in the density field is now resolved by using a finer raymarching step (1/10-th the grid resolution).

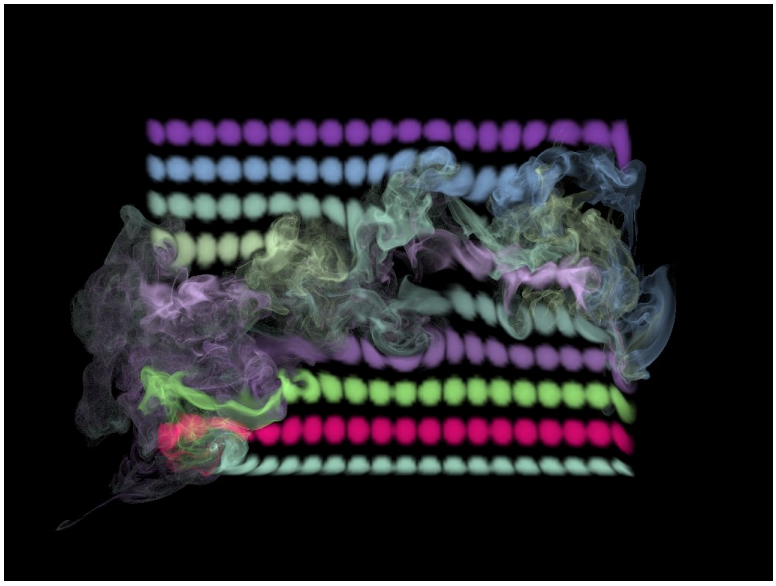


Figure 7.8: Density distribution after 60 frames of gridless advection. The fine detail in the density field is resolved by using a fine raymarching step.

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.

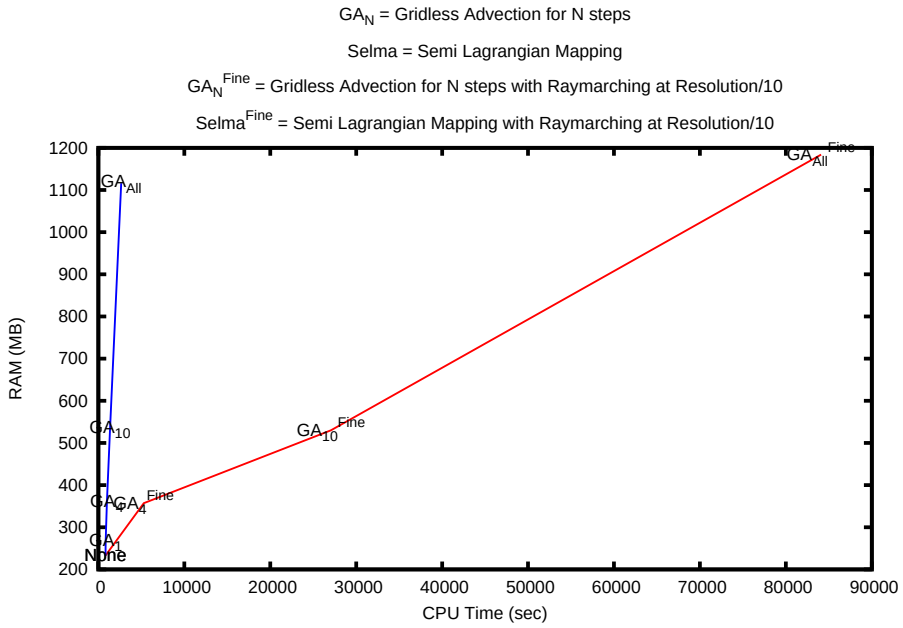


Figure 7.10: Performace of gridless advection as the number of advection frames grows. The steep blue line is gridless advection rendered with the raymarch step equal to the grid resolution. The red line is a raymarch step equal to one-tenth of the grid resolution. These results are not from a production-optimized 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

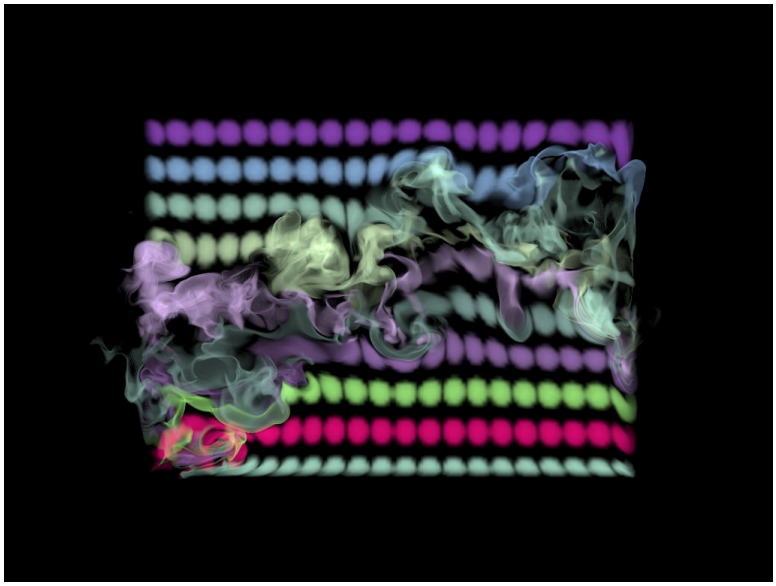


Figure 8.1: Density distribution after 60 frames of SELMA advection. The fine detail in the density field is resolved by using a fine raymarching step.

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.

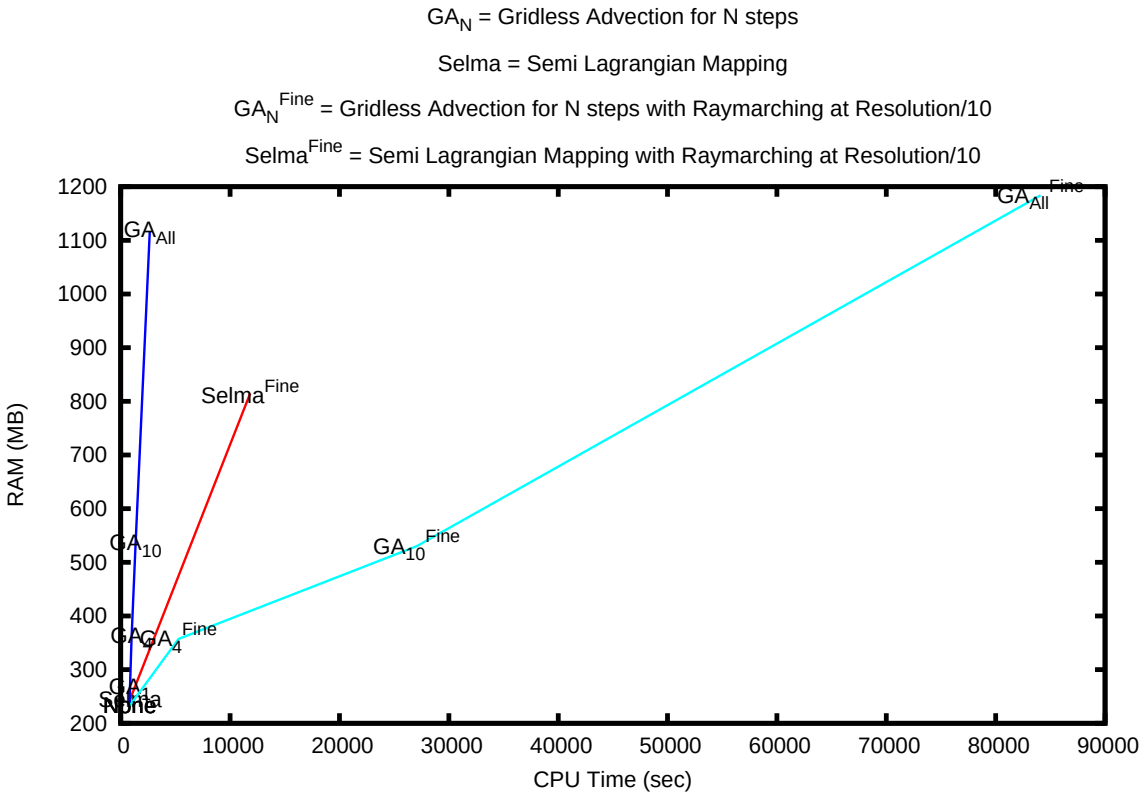


Figure 8.2: Comparison of the performace of Gridless Advection and SELMA.

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:

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

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

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

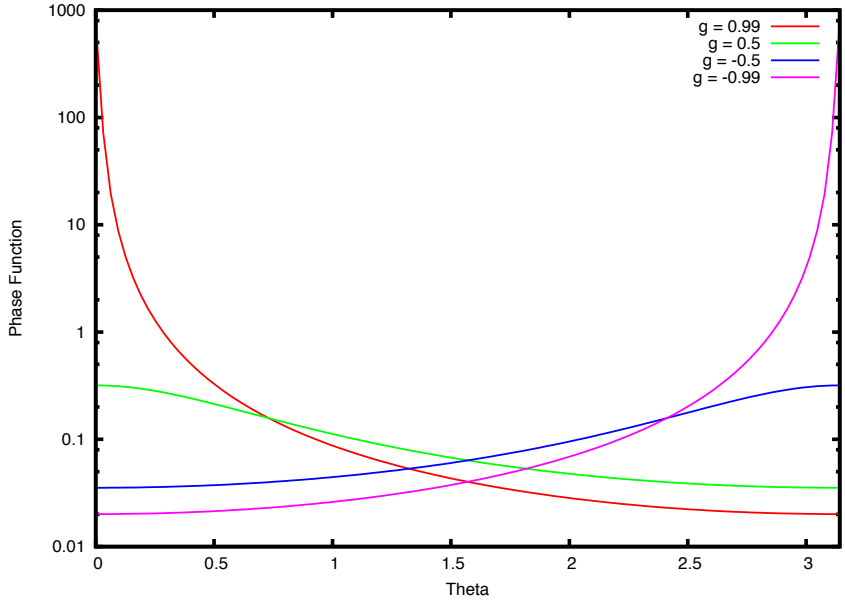


Figure A.1: The Henyey Greenstein phase function for $g = 0.99, 05, -0.5, -0.99$.

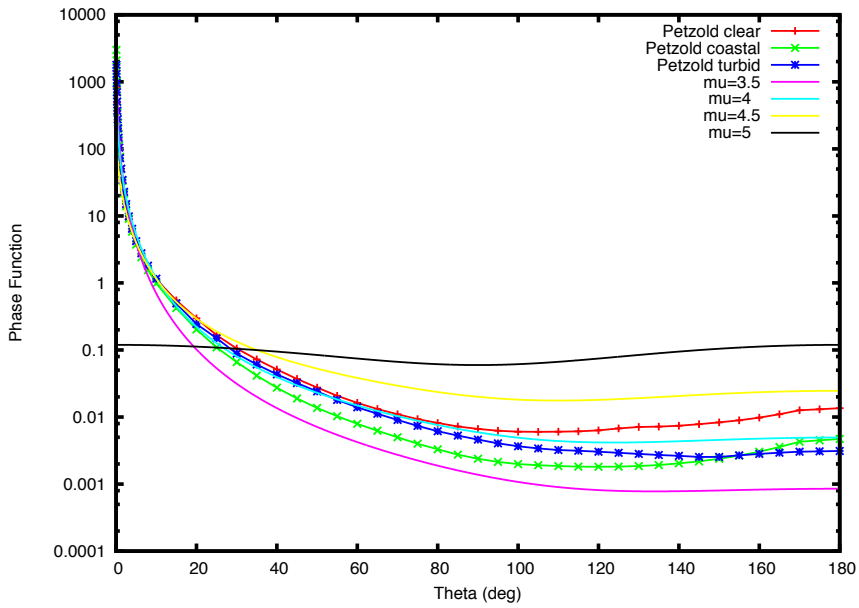


Figure A.2: The Fournier-Forand phase function for $\mu = 0.35, 0.4, 0.45, 0.5$. The parameter n has the value 1.05. Petzold's measured phase functions for clear, coastal, and turbid ocean waters are shown also.

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

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

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

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

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

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

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

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

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)

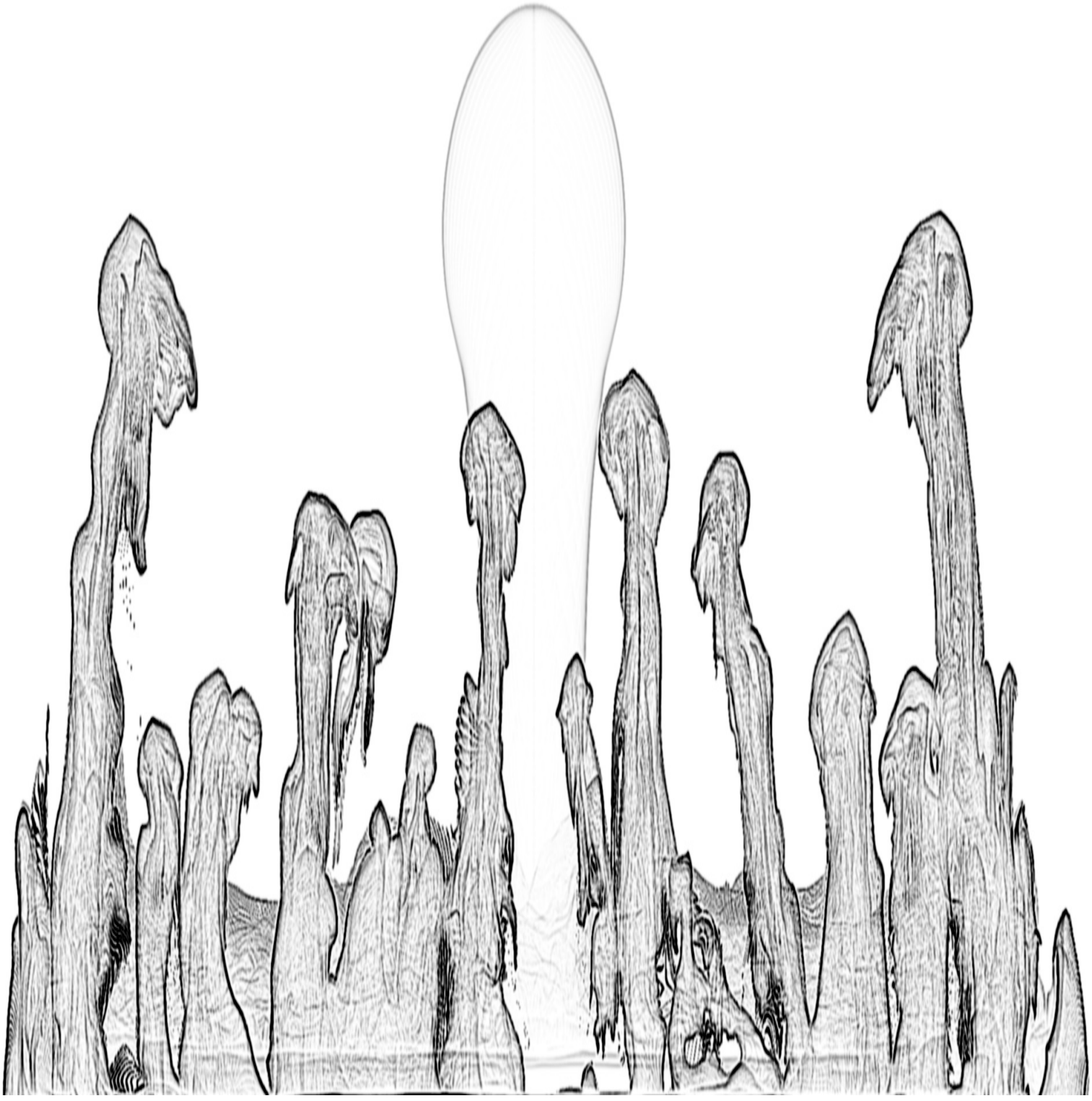
[3] Ken Museth and Michael Clive, *CrackTastic: fast 3D fragmentation in "The Mummy: Tomb of the Dragon Emperor"*, International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH, Los Angeles, California, 2008.

[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
levy = .0.5
numchildren = 1000000000
octaves = 5
rough = 0.5

Appendix E Advection Methodologies

This technical report describes a variety of advection algorithms using a Characteristic Map, including some relatively recent innovations.

Advection Solver Performance with Long Time Steps, and Strategies for Fast and Accurate Numerical Implementation

Jerry Tessendorf*
Digital Production Arts
School of Computing
Clemson University

February 4, 2015

1 Introduction

This note addresses the subject of advection around the theme of applying Characteristic Maps as a method of constructing solvers. Typically the accuracy of solvers is characterized as an asymptotic function of a short time step, short usually defined by restricting the value of the CFL parameter. Within the range of the CFL limit the asymptotic function is expressed as a formula in terms of a power of the time step, i.e. $O(\Delta t^p)$, for some power p.

A key issue is the implementation of solvers that are accurate for "long" time steps. Long time steps might be described as situations that violate CFL restrictions, or at least situations in which it is ambiguous whether the solver can be expected to work based on the CFL restriction. Of course, an advection solver can be used well beyond its CFL limit, with consequent error in the advection. Characterizing the magnitude of that error is of interest. A common approach for computing error is to advect a field over a period of time with some number of steps, then reverse the advection so that, in principle, the field should return to its original form. The error measure is related to the disparity between the original field and the advected one. Defining advection solvers in terms of characteristic maps offers another measure of accuracy. Characteristic maps depend only on the velocity field, and exist independent of the material subject to advection. Given a "reliable" characteristic map, the error of a particular solver could be assessed without invoking any particular material model. This approach is employed in section 7, along with "traditional" evaluation methods.

These issues are addressed here in three stages:

*email: jtessen@clemson.edu

1. Build accurate and fast advection solvers for long time steps using the composition property of characteristic maps. This process is somewhat similar to the Geometric Integration approach for dynamical systems, in that the algorithm starts with a solver that is accurate for short times and builds up one for long times. The product solver has less error than the original. In the case of Geometric Integration, the product solver would have less asymptotic error, i.e. if the original solver has asymptotic error $O(\Delta t^p)$ the product solver would be $O(\Delta t^{p+2})$ or higher. But in this case, the asymptotic error for the product solver is the same as the original, but with a significantly smaller proportionality. However, the construction of the long time advection solver is fast. Typically when a long advection time $T \gg \Delta t$ is desired, the object must be advected over the short time step Δt for $N = T/\Delta t$ times. However, the composition property of the CM generates a solver in $\log(T/\Delta t)$ steps. For example, if $T = 1024\Delta t$, the typical method would require 1024 advections, while the composition approach requires only 10, and has an error 10/1024 times smaller. This is presented in section 3.

2. Build the exact solution of the characteristic map equation. In section 4 the exact solution of the characteristic map equation is derived, apparently for the first time. The solution is explicit, but not analytic. However, a general numerical algorithm, named here "Gradient Stretch," follows in section 6, that is relatively simple but potentially time consuming because accurate exponentiation of 3×3 matrices is required. The appendix includes a discussion of accurate matrix exponentiation. One of the "traditional" test cases for advection solvers employs a velocity field corresponding to a rigid rotation. This test has the fortunate property that the exact characteristic map can be completely evaluated analytically, and as should be anticipated, the map is a rotation transformation in standard form. The general numerical implementation accurately generates this result also.

3. Characterize solver error relative to the exact characteristic map. Typical tests of solver accuracy include two ingredients: a specific velocity field, and an initial object to be advected. The test conducts advection in a way that the exact outcome is known. The measure of solver error follows from the disparity between the advected and expected object. Three of these tests are reproduced in section 7, comparing four different solvers: Semi-Lagrangian, BFECC, Modified MacCormack, and Gradient Stretch. All of these tests involve multiple steps of advection and storing intermediate data on grids. The errors of the solvers cannot be determined directly because the grid storage and interpolation requirement adds errors to the tests. The coupled impact of gridding with multiple advections is demonstrated qualitatively and visually. We also see a set of quantitative error measures by comparing the Gradient Stretch solver directly to other solvers. At short time steps, the classic asymptotic error forms

are reproduced, and at long time steps the solvers transition to a different behavior that has less, but still substantial, error than projected from the asymptotic formulae. This long time error behavior may hint at a "universal" error behavior because the solvers seem to converge on nearly identical behavior.

Before carrying out the outlined stages, the next section reviews the characteristic map formulation of the advection problem. Solvers for Semi-Lagrangian, BFECC, and Modified MacCormack are written in terms of characteristic maps. The approach is phrased for a wide range of velocity fields, including those that are not divergence-free.

2 Characteristic Map Solvers

Advection problems can formally be solved in terms of a vector field called the Characteristic Map (CM) [1], which maps a point in space back to the originating point from which material advected. Generally the CM tracks the time of origination, t', and the current time, t. If the CM is labelled $\mathbf{X}(\mathbf{x}, t, t')$, a field ϕ advected over a time interval Δt updates to the value

$$\phi(\mathbf{x}, t + \Delta t) \;=\; \det\left(\nabla \mathbf{X}(\mathbf{x}, t + \Delta t, t)\right)\; \phi(\mathbf{X}(\mathbf{x}, t + \Delta t, t), t) \tag{1}$$

Scaling by the determinant accounts for changes in field magnitude due to concentration or rarifaction by the underlying velocity field, which may be either compressible or incompressible. This is an exact expression for advection. The time interval Δt is not assumed to be small or large, although it is assumed here to be positive.

The prototypical advection problem in PDE form is

$$\frac{\partial \phi(\mathbf{x}, t)}{\partial t} \;+\; \nabla \cdot (\mathbf{u}(\mathbf{x})\; \phi(\mathbf{x}, t)) \;=\; S(\mathbf{x}, t) \tag{2}$$

where ϕ is the material, and S is an external source/sink of material. The material is assumed to have an initial distribution

$$\phi(\mathbf{x}, t = 0) \;=\; \phi_0(\mathbf{x}) \tag{3}$$

The advection equation has an conservation law for the total amount of the material. Integrating this equation over all 3D space, and assuming there is no material located at spatial infinity, the conservation law is

$$\frac{d}{dt} \int d^3x\; \phi(\mathbf{x}, t) \;=\; \int d^3x\; S(\mathbf{x}, t) \tag{4}$$

which shows that the total amount of the material varies over time due solely to sources and sinks. Advection does not cause net gain or loss of material.

The characteristic map solution of equation 2 is

$$\begin{aligned}\phi(\mathbf{x},t) &= \det\left(\nabla\mathbf{X}(\mathbf{x},t,0)\right)\ \phi_0(\mathbf{X}(\mathbf{x},t,0)) \\ &+ \int_0^t dt'\ \det\left(\nabla\mathbf{X}(\mathbf{x},t,t')\right)\ S(\mathbf{X}(\mathbf{x},t,t'),t') \end{aligned} \tag{5}$$

with the Characteric Map satisfying

$$\frac{\partial\mathbf{X}(\mathbf{x},t,t')}{\partial t} + \mathbf{u}(\mathbf{x})\cdot\nabla\mathbf{X}(\mathbf{x},t,t') = 0 \tag{6}$$

and initial condition $\mathbf{X}(\mathbf{x},t',t') = \mathbf{x}$. This solution also explicitly satisfies the conservation property in equation 4. Integrating this solution over all of 3D space, we have

$$\begin{aligned}\int d^3x\ \phi(\mathbf{x},t) &= \int d^3x\ \det\left(\nabla\mathbf{X}(\mathbf{x},t,0)\right)\ \phi_0(\mathbf{X}(\mathbf{x},t,0)) \\ &+ \int_0^t dt'\ \int d^3x\ \det\left(\nabla\mathbf{X}(\mathbf{x},t,t')\right)\ S(\mathbf{X}(\mathbf{x},t,t'),t') \end{aligned} \tag{7}$$

The combination

$$\int d^3x\ \det\left(\nabla\mathbf{X}(\mathbf{x},t,t')\right) \tag{8}$$

signals a change of integration variable from $\mathbf{x}$ to $\mathbf{X}$. Applying this change,

$$\begin{aligned}\int d^3x\ \phi(\mathbf{x},t) &= \int d^3x\ \phi_0(\mathbf{x}) \\ &+ \int_0^t dt'\ \int d^3x\ S(\mathbf{x},t') \end{aligned} \tag{9}$$

which is equivalent to equation 4.

In a numerical setting in which the material and velocity data may exist on a grid, evaluating advection using the CM means that it is necessary to interpolate the gridded data using whatever interpolation algorithm is of interest. Assuming the interpolation algorithm is bounded, advection via equation 5 is unconditionally stable for the same reason that Semi-Lagrangian advection is unconditionally stable, i.e., the update is bounded by gridded values of the field. But unlike Semi-Lagrangtian advection, we might want our CM advection solver to be valid for higher orders of Δt than linear.

For the remainder of this note we only look at advection over a "single timestep", meaning effectively that the velocity field is constant-in-time during the advection. In this situation, the CM has a time shift symmetry, i.e.

$$\mathbf{X}(\mathbf{x},t,t') = \mathbf{X}(\mathbf{x},t-t',0) \tag{10}$$

This lets us simplify the notation from $\mathbf{X}(\mathbf{x},t,0)$ to $\mathbf{X}(\mathbf{x},t)$.

The CM solver corresponding to Semi-Lagrangian advection is

$$\mathbf{X}_{SL}(\mathbf{x},\Delta t) = \mathbf{x} - \mathbf{u}(\mathbf{x},t)\Delta t \tag{11}$$

Although Semi-Lagrangian error is $O(\Delta t^2)$, other advection schemes have formally smaller error. The BFECC[1] algorithm is constructed from the semi-lagrangian advection as

$$\mathbf{X}_{BFECC}(\mathbf{x}, \Delta t) \;=\; \mathbf{X}_{SL}\left(\mathbf{x} + \frac{1}{2}\left(\mathbf{x} - \mathbf{X}_{SL}\left(\mathbf{X}_{SL}(\mathbf{x}, \Delta t), -\Delta t\right)\right), \Delta t\right) \tag{12}$$

and Modified MacCormack is

$$\mathbf{X}_{MM}(\mathbf{x}, \Delta t) \;=\; \mathbf{X}_{SL}(\mathbf{x}, \Delta t) \;+ \frac{1}{2}\left(\mathbf{x} - \mathbf{X}_{SL}\left(\mathbf{X}_{SL}(\mathbf{x}, \Delta t), -\Delta t\right)\right) \tag{13}$$

Both of these advection schemes have asymptotic error of $O(\Delta t^3)$.

The ideal advection scheme would provide a method to insure as much accuracy as needed for a particular problem. One solution is to employ even higher order advection schemes. But an issue of interest is maintaining accuracy when the time step is large, and it is not known whether high order advection schemes hurt or help.

These issues occur in related dynamical problems in classical mechanics, where the approach of Geometric Integration (GI) fruitfully guides better quality and flexibility in solver construction. GI provides explicit strategies for constructing solvers with high order accuracy from simpler, less-accurate solvers. For example, given a solver $S(\Delta t)$ that updates a dynamical system over time interval Δt with asymptotic error $O(\Delta t^{2p})$, the solver

$$S(\gamma \Delta t)\; S((1-2\gamma)\Delta t)\; S(\gamma \Delta t) \tag{14}$$

where

$$\gamma = \frac{1}{2 - 2^{\frac{1}{2p+1}}} \tag{15}$$

has asymptotic error[2] $O(\Delta t^{2p+2})$. This property is valid universally for Hamiltonian systems of all type, and there are many more similar relationships between solvers. Collectively these universial relationships are valuable tools for selecting a range of accuracy and efficiency criteria for any particular application. No such tools are known for advection.

3 Logarithmic Evaluation of the Characteristic Map

The CM enjoys a composition-in-time property following from the fact that advection conceptually can be deconstructed into a sequence of small advections.

[1] BFECC [3] and Modified MacCormack [5] were not originally constructed as characteristic map solvers. Here they are rebuilt in the language of characteristic maps. Although the error properties and overal structure of these versions follow the original logic, the exact solver is not identical to the originals.

[2] Robert McLachlan and Reinout Quispel, "Six Lectures on the Geometric Integration of ODEs."

Advection solvers that are valid only for small time steps are still useful when long time steps are desired, because a long time advection can be built from iteratively advecting with time steps small enough for the solver to be valid. If the long time step is T, and the solver is valid for time steps $\Delta t < T$, then the material must be advected $N = \text{int}(T/\Delta t)$ steps. For a solver with error $O(\Delta t^p)$, the error for taking a single long step T is $O(T^p)$, whereas taking multiple small steps, the error is $N\ O((T/N)^p)$, or $N^{1-p}\ O(T^p)$. As long as the error exponent p is larger than 1, multiple small advections are more accurate than a single long one. This kind of error analysis does not account for information loss if the data is regridded at each advection step.

The CM has the same composition property for building long-time advections from a sequence of small ones. But for the CM there is an additional fact that the advected CM is also a CM suitable for longer time steps. This is exploitable to reduce the number of advections required from $\text{int}(T/\Delta t)$ to $\text{int}(log_2(T/\Delta t))$. This is a substantial reduction in the number of operations that much be performed, meaning that the advection is faster to execute and there is less regridding loss.

The composition property is the following: given a CM for a time step t_1 and a CM for a time step t_2, the CM for the time step $t_1 + t_2$ follows from the composition of the two:

$$\mathbf{X}(\mathbf{x}, t_1 + t_2) \ = \ \mathbf{X}\left(\mathbf{X}(\mathbf{x}, t_1), t_2\right) \tag{16}$$

In particular, if $t_1 = t_2 = t$,

$$\mathbf{X}(\mathbf{x}, 2t) \ = \ \mathbf{X}\left(\mathbf{X}(\mathbf{x}, t), t\right) \tag{17}$$

This propery sets up the following procedure to construct the CM $\mathbf{X}(\mathbf{x}, T)$ for long time T:

1. Define a short time step $\Delta t \ = \ T/2^M$, for M sufficiently large that Δt is small enough to build an accurate solver.

2. Construct a CM for a chosen short time step Δt, i.e. $\mathbf{X}_0(\mathbf{x}) \ \equiv \ \mathbf{X}(\mathbf{x}, \Delta t)$, using an advection scheme that is accurate for that time step.

3. Construct the following iteration of maps:

$$\mathbf{X}_n(\mathbf{x}) \ = \ \mathbf{X}_{n-1}\left(\mathbf{X}_{n-1}(\mathbf{x})\right), \quad n = 1, \ldots, M \tag{18}$$

The map $\mathbf{X}_M(\mathbf{x})$ is the CM $\mathbf{X}(\mathbf{x}, T)$, and is generated from only M advections of the maps. Normally, a field advected to time $2^M \Delta t$ using an advection scheme accurate for time step Δt requires 2^M advections. This composition rule accelerates long-time advection logarithmically.

How much error is induced by the M advections, called "folds" here, carried out this way? The error for a single short time step is $2^{-pM}\ O(T^p)$, so for the M advections the error accumulates to $M 2^{-pM}\ O(T^p)$. Compared to evaluating all 2^M advections, this error is a factor of $M 2^{-M}$ smaller. In addition, the number

of regridding events is M for this method, as opposed to 2^M previously, so losses due to regridding are reduced as well.

This logarithmic speedup is universally applicable to all advection schemes that are built in terms of characteristic maps.

4 Exact Solution for the Characteristic Map

The exact solution of equation 6 for the CM is an explicit form that depends on the velocity field, the gradient of the velocity field, and the CM at previous times. Derivation starts with constructing the gradient of the CM, $\nabla \mathbf{X}$. From equation 6, it satisfies the evolution equation

$$\frac{\partial \nabla \mathbf{X}(\mathbf{x},t)}{\partial t} \; + \; (\mathbf{u}(\mathbf{x}) \cdot \nabla) \; \nabla \mathbf{X}(\mathbf{x},t) \; + (\nabla \mathbf{u}(\mathbf{x})) \cdot \; \nabla \mathbf{X}(\mathbf{x},t) \; = \; 0 \tag{19}$$

The middle term in this equation induces advection by the velocity field. If this advection term were negligible compared to the other two terms, the gradient has a clear behavior:

$$\nabla \mathbf{X}(\mathbf{x},t) \; \approx \; \exp\left(-t \; \nabla \mathbf{u}(\mathbf{x})\right) \tag{20}$$

Similarly, if the term proportional to the gradient of the velocity were negligible compared to the others, the result is advection of the initial gradient field, which is the identity matrix:

$$\nabla \mathbf{X}(\mathbf{x},t) \; \approx \; \nabla \mathbf{X}(\mathbf{X}(\mathbf{x},t),0) \; \approx \; \mathbf{1} \tag{21}$$

In all other situations, where all three terms balance each other, the solution is:

$$\nabla \mathbf{X}(\mathbf{x},t) \; = \; \exp\left(-\int_0^t dt' \; \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x},t-t')\right)\right)_+ \tag{22}$$

where the notation $(\;)_+$ means that the integral exponential is arranged as an ordered exponential. The definition of an order exponential involves dividing the time interval $(0,t)$ into N segments with time step $\Delta t = t/N$, and arranging them with smallest value of t' on the right most side:

$$\begin{aligned} \exp\left(-\int_0^t dt' \; \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x},t-t')\right)\right)_+ \quad &\approx \quad \exp\left(-\Delta t \; \nabla \mathbf{u}\left(\mathbf{x}\right)\right) \\ &\times \quad \exp\left(-\Delta t \; \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x},\Delta t)\right)\right) \\ &\times \quad \ldots \\ &\times \quad \exp\left(-\Delta t \; \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x},t-\Delta t)\right)\right) \end{aligned}$$

The limit $N \to \infty$ with $N\Delta t = t$ is the exact solution for $\nabla \mathbf{X}$ in terms of the gradient of the velocity field and the advection field $\mathbf{X}$.

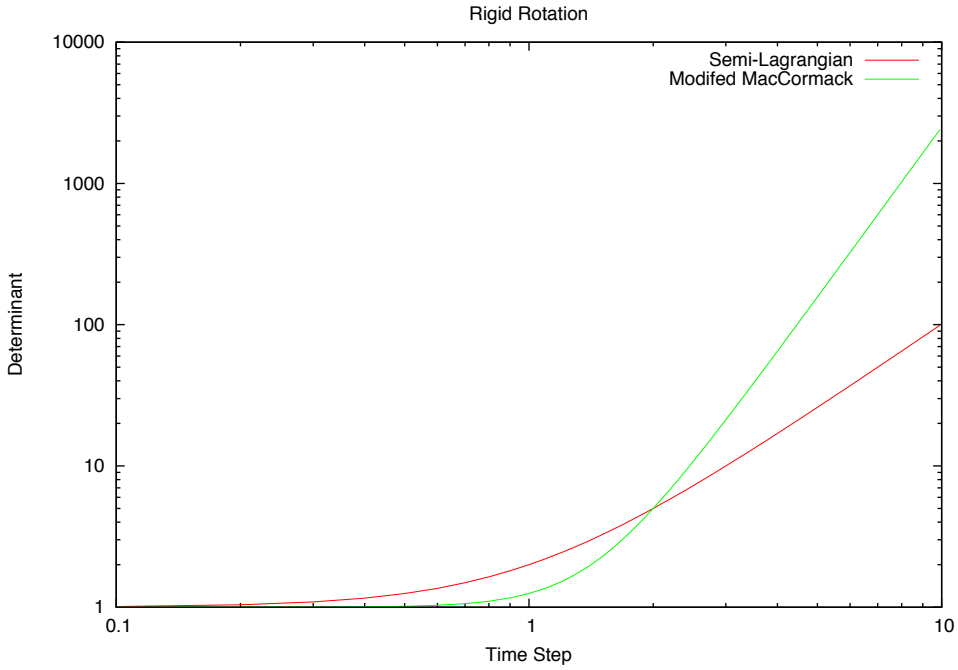


Figure 1: Value of $\det(\nabla \mathbf{X})$ under a rigid rotation, for Semi-Lagrangian and Modified MacCormack solvers. The angular speed is $|\vec{\omega}| = 1$.

This exact expression for the gradient shows several properties. The map gradient is an important quantity for advecting material because its determinant is a factor in equation 5. The determinant for the above expression is

$$\det\left(\nabla \mathbf{X}\right) \;=\; \exp\left(-\int_0^t dt' \;\nabla \cdot \mathbf{u}(\mathbf{X}(\mathbf{x}, t-t'))\right) \tag{23}$$

Two properties come from this expression: (1) For all types of flows, the determinant is positive definite, and the CM is an invertible map; (2) Incompressible flows have a determinant of one.

However, many advection solvers do not enforce this result. An example is shown in figure 1, showing the value of the determinant as a function of the time for the rigid rotation flow. Rigid rotations have an incompressible velocity field $\mathbf{u}(\mathbf{x}) = \mathbf{x} \times \vec{\omega}$, where $\vec{\omega}$ is the angular velocity of the rotation. The deviation from 1 is an error that scales in the same way as the asymptotic error analysis for small time steps, but grows much larger at long times. Although Modified MacCormack has better asymptotic error, at long times it has much larger error than Semi-Lagrangian. In figure 2 the logarithmic evaluation from section 3 has been applied to the solvers with 8 folds, i.e. $M = 8$. The deviation of the

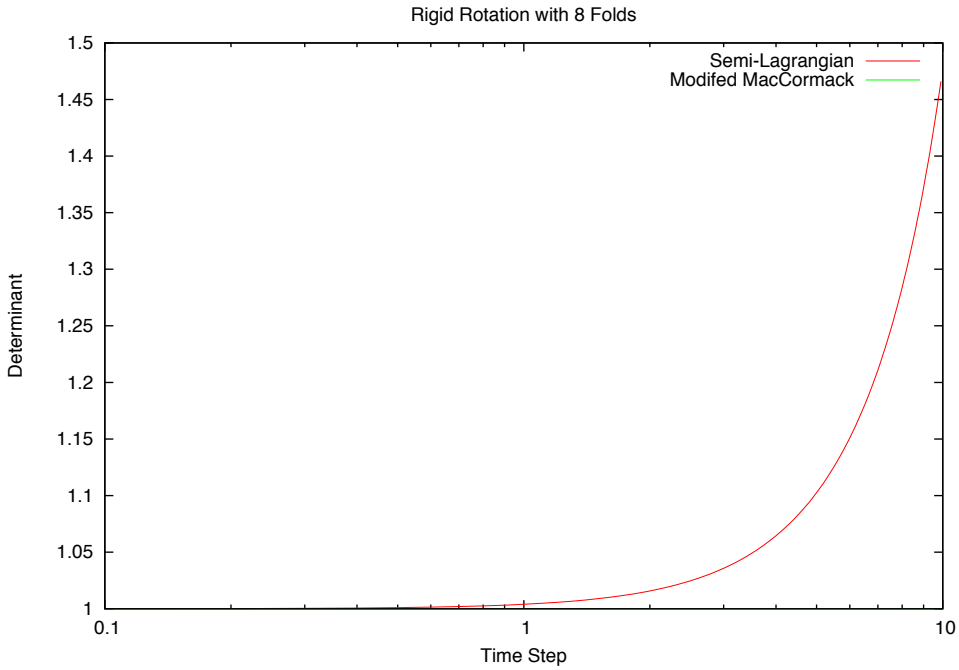


Figure 2: Value of $\det(\nabla \mathbf{X})$ under a rigid rotation, for Semi-Lagrangian and Modified MacCormack solvers and 8 logarithmic folds. Modified MacCormack is not visible in the plot because it deviates from 1 by less than 10^{-5} over this interval. The angular speed is $|\vec{\omega}| = 1$.

determinant from 1 has been reduced by two orders of magnitude for Semi-Lagrangian, and by seven orders of magnitude for Modified MacCormack, even for very long times.

Returning to solving the CM advection equation 6, the exact expression for $\nabla \mathbf{X}$ in equation 22 converts the equation to:

$$\frac{\partial \mathbf{X}(\mathbf{x},t)}{\partial t} = -\mathbf{u}(\mathbf{x}) \cdot \exp\left(-\int_0^t dt' \, \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x}, t-t')\right)\right)_+ \tag{24}$$

This integrates over time to

$$\mathbf{X}(\mathbf{x},t) = \mathbf{x} - \mathbf{u}(\mathbf{x}) \cdot \int_0^t dt' \, \exp\left(-\int_0^{t'} dt'' \, \nabla \mathbf{u}\left(\mathbf{X}(\mathbf{x}, t'-t'')\right)\right)_+ \tag{25}$$

Equation 25 is the exact solution for the CM. It is an explicit solution in that the CM appearing on the right hand side is for earlier times. This is also a starting point for constructing a new numerical advection scheme in section 6 below.

5 Analytic Solution: Constant Gradient, Rigid Rotation

There is one special case in which the CM can be calculated explicity and exactly: a constant gradient of the velocity field. When $\nabla \mathbf{u}$ is a constant matrix, the ordering of the exponential reduced to ordinary exponentiation, and the time integrals in 25 can be evaluated completely:

$$\int_0^t dt' \ \exp\left(-\int_0^{t'} dt'' \ \nabla \mathbf{u}\right) \ = \ \int_0^t dt' \ \exp\left(-t' \ \nabla \mathbf{u}\right) \ \equiv \ t \ \mathrm{sinch}\left(t \nabla \mathbf{u}\right) \quad (26)$$

If the velocity gradient is an invertible matrix, then

$$\mathrm{sinch}(\mathbf{A}) \ = \ (\mathbf{A})^{-1} \ \left(1 - e^{-\mathbf{A}}\right) \quad (27)$$

When it is not invertible, the definition of sinch follows from the Taylor expansion

$$\mathrm{sinch}(\mathbf{A}) \ = \ \sum_{n=0}^{\infty} \ \frac{(-\mathbf{A})^n}{(n+1)!} \quad (28)$$

One important example of a constant gradient is rigid body rotation. The velocity field is

$$\mathbf{u}(\mathbf{x}) \ = \ \mathbf{x} \times \vec{\omega} \quad (29)$$

where $\vec{\omega}$ is the vorticity of the flow. In this case, the gradient matrix is

$$(\nabla \mathbf{u})_{ij} = \sum_k \ \epsilon_{ijk} \ \omega_k \quad (30)$$

and ϵ_{ijk} is the Levi-Civita symbol. With this choice of velocity field, the evaluation of the sinch has an simple analytic form, and the full result is a rotation transformation:

$$\mathbf{X}(\mathbf{x}, t) \ = \ \mathbf{x} \cos(t\omega) \ + \hat{\omega}(\hat{\omega} \cdot \mathbf{x})(1 - \cos(t\omega)) \ - \ (\hat{\omega} \times \mathbf{x}) \sin(t\omega) \quad (31)$$

with $\omega = |\vec{\omega}|$ and $\hat{\omega} = \vec{\omega}/\omega$. This special exact case provides a concrete illustration of the advection solution 25. It is a useful test of solver accuracy, and can also be used as a check of numerical implementations of equation 25 in section 6.

6 Numerical Implementation of the Exact Solution

There are two alternate approaches for implmenting an advection algorithm based on the exact solution 25. One of them is to create a short-time version that

is iterated to arbitrary time using logarithmic acceleration. A good candidate for a short-time version follows from the solution for constant gradient:

$$\mathbf{X}_{GS}(\mathbf{x}, \Delta t) \;=\; \mathbf{x} \;-\; \mathbf{u}(\mathbf{x}) \cdot \Delta t \; \text{sinch}(\Delta t \nabla \mathbf{u}(\mathbf{x})) \tag{32}$$

Note that (a) this short-time advection is exact for any length of time if the gradient is constant, and (2) if the gradient is constant, and $\mathbf{X}_{GS}$ is used in an iteration, the result is still the exact solution. For any time step T, a short time step can be built by selecting a desired number of folds M and setting $\Delta t = T/2^M$. The logarithmic iteration approach of section 3 would build up the full solver.

A second approach is to divide the time interval into N segments to evaluate the integral in 25. The result is algorithm 1. This second approach also enjoys

Algorithm 1 Gradient Stretch Characteristic Map

procedure GRADIENTSTRETCHCHARACTERISTICMAP($\mathbf{u}(\mathbf{x})$, T, N)
  $\Delta t \leftarrow T/N$
  $\mathbf{X}_{GS} \leftarrow \mathbf{x}$
  $\mathbf{M} \leftarrow 0$
  $\mathbf{Q} \leftarrow \Delta t \; \text{sinch}\left(\Delta t \; \nabla \mathbf{u}(\mathbf{x})\right)$
  for $i \leftarrow 0, i < N$ **do**
    $\mathbf{M} \leftarrow \mathbf{M} + \mathbf{Q}$
    $\mathbf{X}_{GS} \leftarrow \mathbf{x} - \mathbf{u}(\mathbf{x}) \cdot \mathbf{M}$
    $\mathbf{Q} \leftarrow \mathbf{Q} \cdot \exp\left(-\Delta t \; \nabla \mathbf{u}(\mathbf{X}_{GS})\right)$
    $i \leftarrow$i + 1
  end for
  return $\mathbf{X}_{GS}$
end procedure

the property that if the gradient is constant, the result is exact, regardless of the choice of N. Note that the choice $N = 1$ reduces algorithm 1 to equation 32.

A third algorithm follows from combining these two. A fold value M and time division N can be used to evaluate algorithm 1 with arguments $(\mathbf{u}, T/2^M, N)$, then iterate that map through M folds.

Despite the assembly of this algorithm from the exact CM, and the robustness of the algorithm for rigid rotations, it has very limited utility for many practical advection scenarios. Accurate computation of the sinch($\mathbf{A}$) function and matrix exponentiation is time consuming. The appendix gives some suggestions based on the effort to produce accurate error analyses in section 7. In comparison, it is much faster to use the CM form of other algorithms, for example Semi-Lagrangian, BFECC, Modified MacCormack, or others, along with the logarithmic iteration process to improve accuracy.

Test	Velocity Field	Domain	$N_i \times N_j \times N_k$	GS M, N
Rigid Rotation	$\frac{\pi}{314}\mathbf{x} \times \vec{\omega}$ $\vec{\omega} = (0, 1, 0)$	$(-100, -100, -100) \times (100, 100, 100)$	$20 \times 20 \times 20$	0, 256
Shear	$(\sin^2(\pi x)\sin(2\pi y),$ $-\sin^2(\pi y)\sin(2\pi x),$ $(1 - r/R)^2)$ $r = ((x - 0.5)^2 + (y - 0.5)^2)^{1/2}$ $R = 0.5$	$(0, 0, 0) \times (1, 1, 1)$	$20 \times 20 \times 20$	11, 20
LeVeque Twist	$(2\sin^2(\pi x)\sin(\pi y)\sin(2\pi z),$ $-\sin^2(\pi y)\sin(\pi z)\sin(2\pi x),$ $-\sin^2(\pi z)\sin(\pi x)\sin(2\pi y))$	$(0, 0, 0) \times (1, 1, 1)$	$100 \times 100 \times 100$	11, 20

Table 1: Velocity fields and calculation domains for the error statistics.

7 Solver Error

One of the standard tests of advection schemes is advection of shapes in a chosen flow, usually reversing the flow after a time and advecting backward for an equal amount of time. The shape is compared before and after advection to assess the accuracy of the advection scheme. These tests, applied to the Semi-Lagrangian, BFECC, Modified MacCormack, and Gradient Stretch algorithms for three different flows listed in table 1, illustrate their relative qualitative performance.

The gradient stretch advection algorithm also provides a reference for estimating the error of other algorithms. Choosing a large value for the fold parameter M for logarithmic iteration, and a large value for the time division parameter N, the value of $\mathbf{X}_{GS}$ can be taken as "ground truth" for comparison with other solvers. An error field defined as

$$\mathcal{E}(\mathbf{x}) \;\equiv\; \mathbf{X}_{solver}(\mathbf{x}) \;-\; \mathbf{X}_{GS}(\mathbf{x}) \tag{33}$$

is the source of spatially-sampled error statistics for the mean error

$$\langle \mathcal{E} \rangle \;\equiv\; \frac{1}{N_i\ N_j\ N_k} \sum_{ijk}\ \mathcal{E}(\mathbf{x}_{ijk}) \tag{34}$$

and the rms error

$$\sigma_{\mathcal{E}} \;\equiv\; \left\{ \frac{1}{N_i\ N_j\ N_k} \sum_{ijk}\ \left(\mathcal{E}(\mathbf{x}_{ijk}) - \langle \mathcal{E} \rangle\right)^2 \right\}^{1/2} \tag{35}$$

using a grid of points $\mathbf{x}_{ijk}$ from a relevant rectangular domain, as listed in table 1 for each test case. The GS solver parameters M and N were chosen for each case to make sure that the statistics are accurate.

7.1 Rigid Rotation

For rigid rotations, the gradient stretch solver is exact and completely preserves the shape without loss, even for $M = 0$ and $N = 1$. The visual test case [2]

consists of a sphere with a notch removed from it rotating around a point outside the sphere, as shown in figure 3.

For the error statistics, the value of N was 256 in order to insure the accuracy of the matrix sinch and exponentiation operations even for large timesteps. Figure 4 shows the rms error $\sigma_{\mathcal{E}}$ for the three solvers as functions of timestep. Through the entire four decades of time step, the error is the power-law predicted by asymptotic analysis: $O(\Delta t^2)$ for Semi-Lagrangian and $O(\Delta t^3)$ for BFECC and Modified MacCormack.

7.2 Shear

Visually, BFECC, Modified MacCormack, and Gradient Stretch produce essentially identical results for the shear test [4], with a small amount of distortion of the sphere after the advections, and Semi-Lagrangian distortion is substantially greater, as seen in figure 5. For the visual test, the simplest form of gradient stretch was used, i.e. $N = 1$, $M = 0$ corresponding to equation 32.

Figure 6 displays the rms error for the solvers. Quantitatively the asymptotic error holds for BFECC and Modified MacCormack over the range of time scales tested. But near $\Delta t \sim 1$ the Semi-Lagangian error transitions from $O(\Delta t^2)$ to $O(\Delta t)$ at large time steps, with overall lower error.

7.3 LeVeque Twist

As with the rigid rotation and shear tests, Semi-Lagrangian advection produces noticeably larger errors than the other solvers for the LeVeque Twist [4] test, visually demonstrated in figure 7. At small time steps the rms errors exhibit the asymptotic behavior. Near $\Delta t \sim 1$ all three solvers deviate to an error $O(\Delta t)$, in figure 8.

8 Conclusion

The Characteristic Map has been employed systematically in this note as a tool for creating practical, stable, efficient, and accurate advection schemes. The use of a logarithmic folding of advections accelerates the creation of long time step schemes from short time step ones, with much less error. A new "gradient stretch" advection scheme has been derived from the exact expression for the Characteristic Map, although this scheme is computationally heavy due to the need to evaluate matrix-valued exponentials and sinch functions. However, the gradient stretch algorithm enjoys accuracy benefits not shared by other advection schemes, such as being the exact solution in the case of rigid rotation, and highly accurate computation of the determinant of the gradient of the map, with consequent elimination of numerically-induced volume loss or gain.

In the shear and LeVeque Twist test cases, the behavior of the rms error underwent a transition at large time steps. In the shear test, the Semi-Lagrangian scheme transitioned from the short time behavior of $O(\Delta t^2)$ to $O(\Delta t)$, and in

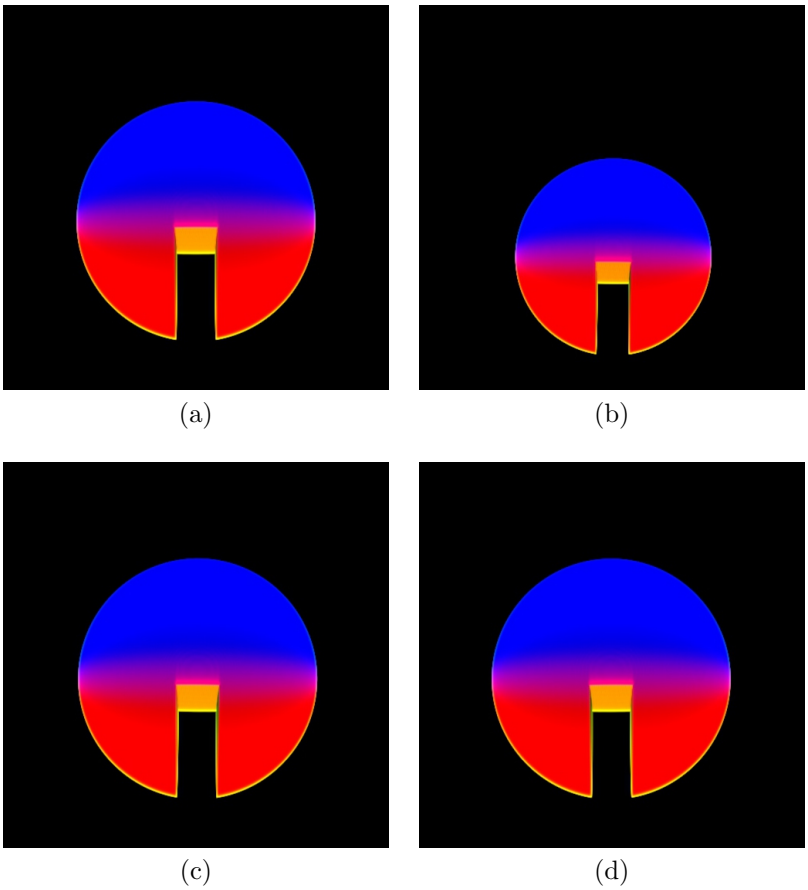


Figure 3: Notched sphere prior to rotation (a); after 360° rotation by Semi-Lagrangian advection (b); BFECC (c); and Modifed MacCormack (d). The time step is 6.28 and there were 100 advections.

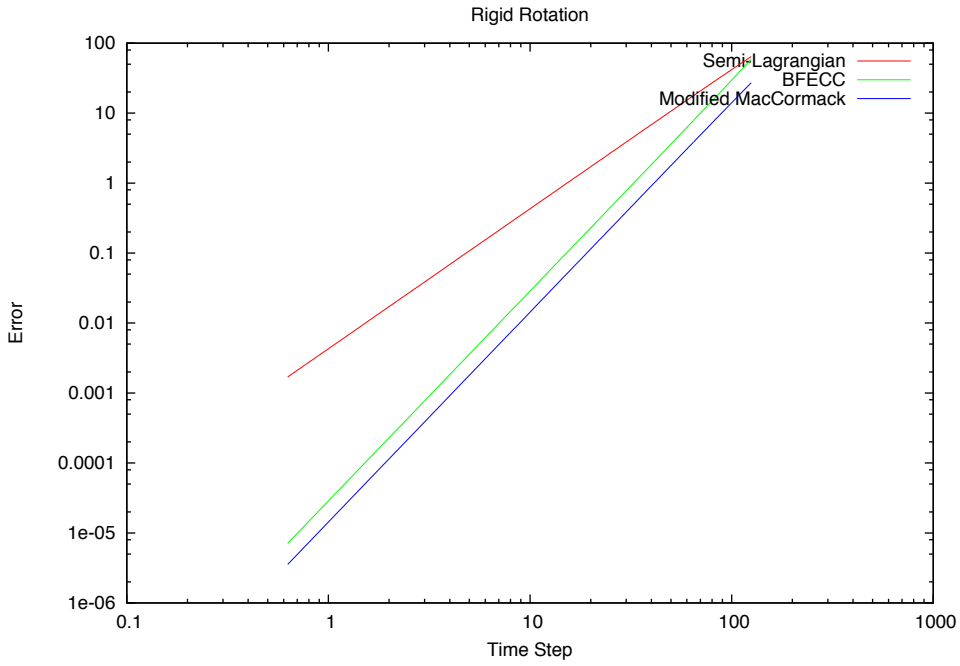


Figure 4: The rms error $\sigma_{\mathcal{E}}$ for the three solvers Semi-Lagrangian, BFECC, and Modified MacCormack, as a function of time step, for the Rigid Rotation case. The CM $\mathbf{X}_{GS}$ was calculated using $M = 0$ and $N = 256$.

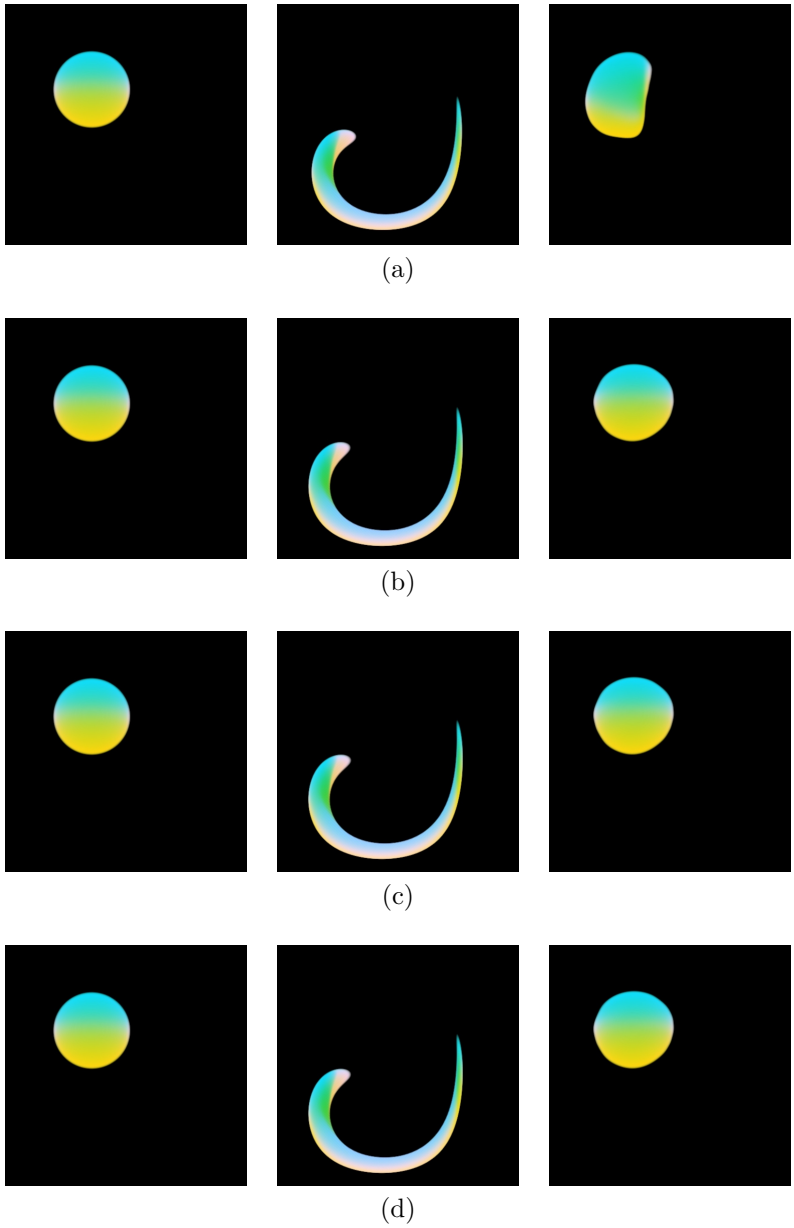


Figure 5: Sphere prior to shear (left) and with maximum shear (center), and returned to original position (right). (a) Semi-Lagrangian; (b) BFECC; (c) Modifed MacCormack; and (d) Gradient Stretch with $N = 1$. The time step was 3/150 and there were 150 advections. The Gradient Stretch case (d) used $N = 1$ and $M = 0$.

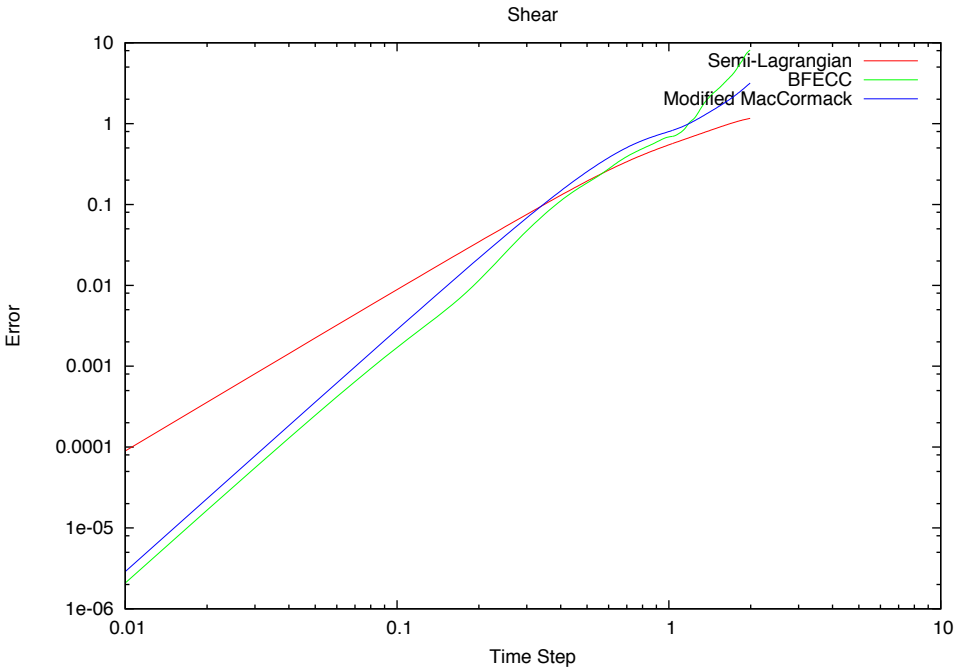


Figure 6: The rms error $\sigma_{\mathcal{E}}$ for the three solvers Semi-Lagrangian, BFECC, and Modified MacCormack, as a function of time step, for the Shear case. The CM $\mathbf{X}_{GS}$ was calculated using $M = 11$ and $N = 20$.

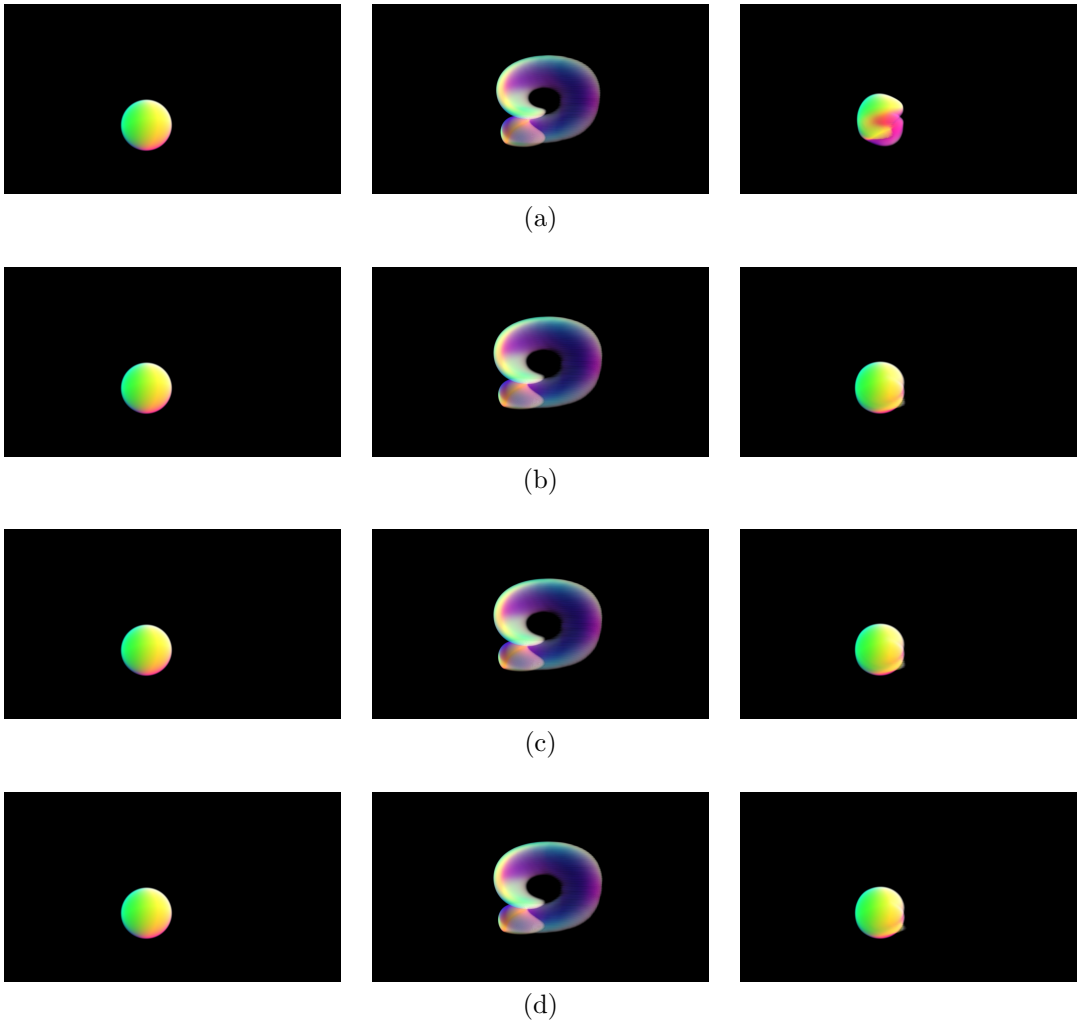


Figure 7: Sphere prior to advection (left) and with maximum advection (center), and returned to original position (right). (a) Semi-Lagrangian; (b) BFECC; (c) Modifed MacCormack; and (d) Gradient Stretch with $N = 6$. The time step was 30/150 and there were 15 advections. Each solver used $M = 4$ folds.

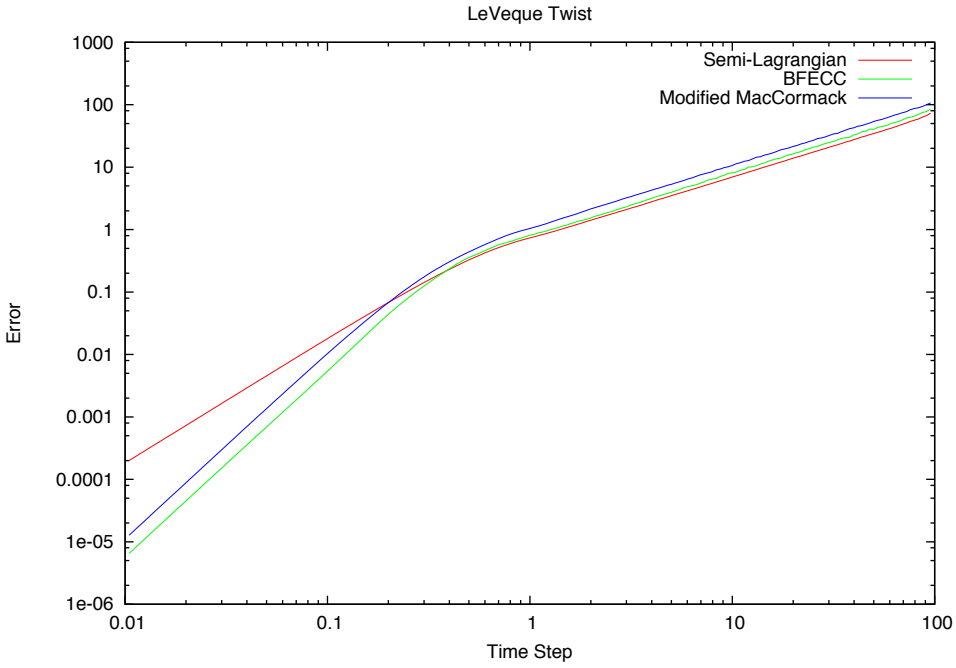


Figure 8: The rms error $\sigma_{\mathcal{E}}$ for the three solvers Semi-Lagrangian, BFECC, and Modified MacCormack, as a function of time step, for the LeVeque Twist case. The CM $\mathbf{X}_{GS}$ was calculated using $M = 11$ and $N = 20$.

the LeVeque Twist test all three schemes made a transition from their respective short time error to nearly identitical $O(\Delta t)$ behavior.

In the interest of motivating future investigation, here is a speculation on the source of this transition. The ordered exponential term can have a wide range of behaviors depending on the magnitude and specifics of its argument. In the case of the rigid rotation, the ordered exponential reduced to a purely oscillatory behavior. When there is a very large argument, i.e. large gradient or large time step or both, strong oscillations in magnitude and phase would be very difficult to reproduce via an advection scheme with an asymptotic error $O(\Delta t^p)$ with p relatively small. In such a situation, the oscillations could dominate an rms error measure such the one used here. Thinking of the ordered exponential as a rapidly fluctuating random variable, the error measure associated with the variance calculation would generate the $O(\Delta t)$ behavior seen in some of the results for the shear and LeVeque Twist cases.

Appendix: Matrix Exponential and Sinch

Matrix exponentiation has a relatively simple implementation in terms of a truncation of the Taylor series

$$e^{\mathbf{A}} \;=\; \sum_{n=0}^{\infty} \frac{\mathbf{A}^n}{n!} \tag{36}$$

to include only the first N powers of $\mathbf{A}$, and is depicted in algorithm 2. For many

Algorithm 2 Matrix exponential via Taylor expansion

```
procedure EXP(A, N)
    E ← 1
    M ← A
    for i ← 1, i ≤ N do
        E ← E + M
        M ← M * A/(i + 1)
        i ←i + 1
    end for
    return E
end procedure
```

choices of $\mathbf{A}$, this approach produces reasonable accuracy for truncations $N \sim 50$. Cases in which the exponential is oscillatory can require many more terms, $N \sim 200$ or more, to insure good reproduction of the phase and amplitude. A good strategy for reducing this load is to take advantage of the multiplicative property of the exponential

$$e^{\mathbf{A}} \;=\; \left(e^{\mathbf{A}/\ell}\right)^{\ell} \tag{37}$$

This relationship is similar to the one for logarithmic acceleration of advection in section 3, and can be exploited similarly in algorithm 3. This algorithm gives

Algorithm 3 Fast matrix exponential via Taylor expansion

```
procedure FastExp(A, N, M)
    E ← Exp(A/2^M, N)
    for i ← 0, i < M do
        E ← E * E
        i ←i + 1
    end for
    return E
end procedure
```

accurate results even for relatively small values of $N \sim 30$, $M \sim 10$ unless $\mathbf{A}$ has elements with very large magnitude.

For the sinch function, there is a simple expression in terms of the exponential when $\mathbf{A}$ is invertible:

$$\text{sinch}(\mathbf{A}) \;=\; \mathbf{A}^{-1} \left(1 \;-\; e^{-\mathbf{A}}\right) \tag{38}$$

When $\mathbf{A}$ is not invertible, there is no choice but to use a relatively time consuming truncation of the Taylor expansion

$$\text{sinch}(\mathbf{A}) \;=\; \sum_{n=0}^{\infty} \frac{(-\mathbf{A})^n}{(n+1)!} \tag{39}$$

as implemented in algorithm 4. In practice, the Taylor expansion truncation

Algorithm 4 Matrix sinch via Taylor expansion

```
procedure Sinch(A, N)
    E ← 1
    M ← −A/2
    for i ← 1, i ≤ N do
        E ← E + M
        M ← M * (−A)/(i + 2)
        i ←i + 1
    end for
    return E
end procedure
```

must be very large, i.e. $N \sim 200$, to produce reasonable accuracy over the range of cases in this note.

References

[1] JohnC. Bowman, MohammadAli Yassaei, and Anup Basu. A fully lagrangian advection scheme. *Journal of Scientific Computing*, pages 1–27, 2014.

[2] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.*, 183(1):83–116, November 2002.

[3] ByungMoon Kim, Yingjie Liu, Ignacio Llamas, and Jarek Rossignac. Flowfixer: Using bfecc for fluid simulation. In Pierre Poulin and Eric Galin, editors, *NPH*, pages 51–56. Eurographics Association, 2005.

[4] Peter Liovic, Murray Rudman, Jong-Leng Liow, Djamel Lakehal, and Doug Kothe. A 3d unsplit-advection volume tracking algorithm with planarity-preserving interface reconstruction. *Computers and Fluids*, 35(10):1011–1032, December 2006.

[5] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35(2-3):350–371, June 2008.