8-2012

A Mathematical Framework for Volume Modeling and Simulation

Brandon Pelfrey
Clemson University, brandonpelfrey@gmail.com
A Mathematical Framework for Volume Modeling and Simulation

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

by
Brandon Pelfrey
August 2012

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Donald House
Dr. Robert Geist
Abstract

Rendering, modeling, and simulating processes in spatially-varying media are complex tasks which typically require tedious mathematical derivations and long development and programming. In this thesis, a framework is presented which allows for fast and casual exploration of ideas for volume-based simulation and modeling. Several applications of the framework are shown, including the simulation of fluid dynamics for computer graphics applications. A procedure for automatically generating OpenCL code from these is given, along with rudimentary examples.
Dedication

I am forever indebted to those which have supported me for the last six years. I am grateful for my advisors Dr. Donald House and Dr. Jerry Tessendorf for their insightfulness, instruction and tutelage, and willingness to let me pursue those subjects which I found exciting.

I have enjoyed the company of excellent and supportive colleagues, including Jay Steele, Zachary Jones, Christopher Corsi, Meng Zhu, Jonathan Cox, Sam Casacio, and Jianwei Liu. I am also forever grateful for Emily Young, who put up with my late night studying and was kind enough to look at endless permutations of smoke bunnies and spheres.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Previous Work</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A New Framework</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Differentiable Fields and Applications</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Language and Functions</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Case Study: Fluid Simulation</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>The Dynamics of Fluids</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Advection</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Viscosity and Turbulence</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Enforcing Incompressibility</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>A Compact Fluid Solver</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Simulating the Rayleigh-Taylor Instability</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Case Study: Blending Fluid Simulations with the Characteristic Map</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Past Work</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>The Characteristic Map</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Displacement Composition for Explosions</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Conclusion</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation Details and Optimizations</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Interactive Scripting</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>OpenCL-Based Acceleration</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>OpenCL as a Parallel Computation Framework</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Generating Functions from the Operator Graph</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>Example execution and results</td>
<td>34</td>
</tr>
<tr>
<td>5.4</td>
<td>Future Direction</td>
<td>35</td>
</tr>
<tr>
<td>Appendices</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>A</td>
<td>Python Script Example for Automatically Generated OpenCL Code</td>
<td>37</td>
</tr>
<tr>
<td>B</td>
<td>Automatically-Generated OpenCL Code</td>
<td>37</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Rendering of running a simple fluid simulation using gridless advection.</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Simulation showing turbulent mixing in the Rayleigh-Taylor instability. The sequence on the left is rendered with density resampled to a grid each frame. The sequence on the right is rendered by gridless advection.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>During volume rendering, the CM, ( \mathbf{X} ), is used to map the initial density distribution ( \rho_0 ) to a new position ( \mathbf{X}(\mathbf{x}) ) due to fluid flow.</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>SELMA-based rendering (left) preserves much more detail when compared to more traditional approaches of rendering explicitly-advected density (right). This ability to capture high spatial gradient enables the rendering of sheets of material, with no virtually no density diffusion.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Frame from the explosion created by compositing rotated copies of the SELMA displacement data (Algorithm 1)</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Sequence of frames from the shape blend described in algorithm 2.</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Expression tree example. This tree, when evaluated at the root, represents the expression ( 1 -</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>UML diagram demonstrating the relationship between the scalar field and it’s underlying representation.</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>OpenCL-accelerated rendering of a procedurally-defined volume. The volume consists of a single large sphere with many smaller spheres “cut out” of the larger one.</td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Volume modeling, rendering, and simulation are complex processes that are becoming more important in many diverse areas, including the visualization of scientific data and visual effects production. While there has been significant progress made in the last decade in rendering and simulating phenomena, the area of procedural modeling has had little investigation. Of those processes which have been studied, many require complex sets of skills. This can make writing and testing even the most basic of ideas a time-consuming and error-prone endeavor. This thesis investigates new tools for expressing and prototyping ideas about volume media.

Simulating volumetric phenomena whose movement occur over space and time is usually a computationally-expensive and complicated process requiring domain-specific knowledge and many trials of iterative refinement, tweaking settings and parameters to achieve a desired appearance or motion. One difficulty barring the investigation of new methods for simulation is the lack of separation of idea from implementation. Often, complex new ideas are difficult to implement, requiring significant effort and programming knowledge on the part of the researcher. This effort makes prototyping difficult and makes editing an existing prototype cumbersome.

In many disciplines of simulating dynamics of volumetric quantities, traditional methods require finding a discrete analog of the continuous dynamics, through methods like Finite Element Methods (FEM) or Finite Difference Methods (FDM). This means that differential operators and other pieces of the simulation require deriving discrete counterparts, which can be difficult to derive and are, perhaps more importantly, only approximations of the continuous case. A variety of discretizations satisfy the technical needs of convergence and consistency, yet produce very different
simulation results. Exploring this variety efficiently requires an adaptable development environment. The more flexible and intuitive this environment is at expressing these mathematical concepts, the faster ideas may be explored and tested.

Modeling volume phenomena is a similarly complicated process which typically involves special tools, experienced user knowledge, and a time-intensive effort of finding the right parameters. There are few tools for procedurally creating or modeling volume content, and even fewer tools for manipulating and exploring new ideas. Often, some of the most interesting ideas and combinations come from free-form and natural investigation.

Rendering volume media is also a complex process in its own right. In production of visual effects for motion pictures, there is an increasing need to render semi-transparent media like smoke, clouds, and dust. Traditional means of rendering these types of media, such as using opacity-mapped billboards, are limited in their fidelity and artist flexibility. The increase in compute capabilities and advances in computer graphics research have produced a great number of algorithms and systems whose outputs require this volume rendering. Manipulating this data in ways that are artist-friendly and intuitive has been a challenge because of the lack of good tools.

In this thesis, a new framework is presented which enables unconstrained, interactive investigation of new algorithms and effects involving volume media in an attempt to address the problems mentioned above. The framework provides a high performance and flexible way of exploring ideas about volume media. Specific examples in fluid simulation and artistic modeling are provided.

1.1 Previous Work

1.1.1 Dataflow Programming

Dataflow Programming is a rich and widely studied area of computer programming languages that focuses on how data is manipulated by a series of operations between inputs and outputs. In contrast, typical imperative programming languages are concerned with the flow of individual instructions and items through the system. In dataflow programming, input data can be passed through a directed graph of operations before becoming output data. Because dataflow programming operates in some sense at a higher level, that of operations on abstract units of data, programing creation and debugging is often relatively simplified.

Additionally, the inherent graph-like nature of dataflow programming allows for more easily
manipulating the program representation when compared to imperative languages. For example, the tree structure of an Expression Tree allows for easily translating and manipulating the representation of the program, for example in cross-compilation to another language.

Dataflow systems have been created for many domain-specific tasks, such as real-time robotic controllers, signal processors, and even the creation of algorithmic music. General systems have also been developed, such as Simulink [8], which provides “multi-domain simulation and Model-Based Design for dynamic and embedded systems”. Many dataflow programming systems use graphical interfaces and allow the user to explicitly add operations and connections between data using a graph system with nodes and edges representing the actual dataflow. These so-called “visual programming” systems are often designed to allow code-less programming by assembling the graph from a pre-defined set of instructions and operations. This paradigm allows for fast design and prototyping of ideas.

1.1.2 Frameworks in Visual Effects and Scientific Visualization

Dataflow programming has been extensively used in visual effects production for many years and for many purposes. Side Effects Software’s Houdini [12], users are given a node-based visual workflow which processes many different types of input, including particle and geometry data. In Autodesk’s Softimage, the ICE framework [1] allows for a similar node-based workflow which can additionally control shading. Rhythm and Hues Studio’s proprietary software, FELT [14] (Field Expression Language Toolkit), is another dataflow-based system, where geometry, dynamics, and other data can be easily processed. FELT is unique in that, in addition to geometry and particles, entire fields of data can be operated on. This abstraction allows for the creation of novel dynamics and artistic compositions with little effort.

Dataflow programming and visual programming are excellent at providing a platform for freeform exploration of data and concepts. Because ideas are simple to implement and “try out”, a data explorer need not spend much time trying an idea and explore its implications on input data. In the case of data mining, it is often not known which relationships to look for or what kinds of trends or behaviors may be found, and so it is important that flexible tools for exploring the data in new ways are available.
1.2 A New Framework

For the purposes of this research, a new framework was developed which addresses these issues. The framework was designed to satisfy the following goals:

- **Maintain, as closely as possible, a mathematical language.**
  Keeping a consistent mathematical form enables users to more quickly input and edit mathematical ideas.

- **Enable effortless, automatic differentiation of all data**
  Spatial gradients are commonly needed objects in both simulation and rendering. By enabling transparent, automatic differentiation, the need to derive discrete differential operators, which can be an error-prone process, is removed. In situations where discrete objects (gridded data) are in use, appropriate discrete differentiation techniques should be used.

- **Preserve mathematical ideas in an exact form whenever possible.**
  When data is given in an exact form (expressions of “primitive” types), this exact form should be preserved in order to avoid errors that may be introduced by discretization. Additionally, when well-defined, the derivatives of exact expressions should also be exact.

The framework expresses mathematics in terms of objects representing three types of tensor fields: scalar fields, vector fields, and matrix fields. An operation on one object carries the notion of performing that operation at every point in space, but involves no evaluation until needed. This form of lazy evaluation enables complex systems to be compactly represented in an exact and resolution-independent way.

In order to facilitate this kind of evaluation, an expression is created symbolically by the user, which may then be “queried” at a particular point in space for its value. At this point, the expression tree encoding the user’s input is evaluated from the leaves towards the root, finally emitting a value.

1.3 Differentiable Fields and Applications

In this framework, the choice to allow for only three types of tensor fields is one of practical importance. In computer graphics and visualization, rank 3 (and higher order) tensors are not
common. The inclusion of arbitrarily high order tensors is possible, but would add an additional layer of unnecessary abstraction. Additional levels of abstraction are detrimental when they take place at low levels, where operations happen with very high frequency. Additionally, because graphics and visualization are overwhelmingly concerned with generating, manipulating, and rendering objects in three-dimensional space, all of the fields in this framework exist in three-dimensional Euclidean space.

It is assumed that scalar and vector-valued fields in this framework are at least once-differentiable. Procedural expressions involving non-discretized data may have exactly-defined gradients, and as expected, the gradient operation on these expressions will return exact results. In cases where data is presented in a discrete format, such as on uniform grids, “exact” differentials are not available and so finite differences are used.

1.4 Language and Functions

The framework developed for this thesis has been implemented with both flexibility and performance in mind. In all cases where these two goals are inconsistent, there was, by design, a preference for flexibility. This flexibility is paramount given that all of the operations a user may want to perform cannot be entirely predicted in advance. With respect to performance, all of the computationally expensive operations are handled in C++. To enable a flexible environment for prototyping, this C++ functionality is exposed to the user in terms of Python bindings. While C++ has relatively low overhead, Python is an interpreted, interactive language and, as such, implementation of numerically-heavy mathematical code is not common. This combination exposes the strengths of both languages and brings flexible, high performance computing to the user.

This language provides a facility for easy prototyping; however, it is recommended that performance-critical code be written in a more “close-to-the-metal” language, such as C++.

To aid the user in prototyping, and to avoid needlessly re-writing common code, there are a number of built in functions and utilities. These facilities allow for function composition, derivatives, translations, rotations, and other transformations. Here several of these operations are explained in context; For a full description of all functions available in the framework, documentation has been provided in the appendix.
1.4.1 Constant Values

The simplest type of expression that can be formed in the framework is a constant value. The `constant` function creates a field whose value everywhere is equal to the argument passed in:

```plaintext
# Create a scalar field whose value is 2.4 everywhere
a = constant(2.4)

# Create a vector field whose value is (1,-3,2) everywhere
b = constant(Vector3(1,-3,2))
```

1.4.2 Arithmetic on Fields

Every arithmetic operation which is appropriate for two given types is valid within the framework. For example, a scalar field object and a vector field object can be multiplied in the usual way to produce a new vector field. Other simple arithmetic operations from linear algebra are also available, such as inner (`dot`), outer (`outerproduct`), and cross products (`cross`).

```plaintext
# Simple arithmetic operations on different type fields
c = constant(Vector3(2,7,-1.2)) + constant(3.2) * constant(Vector3(2,3,-1))

# Familiar operations from linear algebra
d = dot(c, c) * cross(c, b)

# Normalize a vector field
e = d / length(d)
```

Matrix fields also have several elementary operations defined:

```plaintext
# Compute the transpose of a matrix
mt = transpose(m)

# Compute the inverse of a matrix
mi = inverse(mt)

# Compute the trace of a matrix
divergence = trace(grad(velocity))
```

1.4.3 Discrete Data (Grids)

Discrete data is often described by using regular lattice grids, with a value stored, at each node in the grid. In the framework, grids can be created from other (possibly procedural, or not) fields and also saved to, and loaded from, storage.
In order to describe grids, a definition of the enclosing domain is required. Grids in the framework are defined by an axis-aligned bounding box which is defined by an extreme lower corner and extreme upper corner, as well as resolution along each axis:

```python
# Create a 64x128x64 grid enclosing the region from (-1,-2,-1) to (1,2,1)
domain = Domain(64,128,64,Vector3(-1,-2,-1),Vector3(1,2,1))
```

These domains are used in several contexts in the framework whenever a description of discrete space is needed.

`writeToGrid` is a built-in function which takes three arguments: the field which should be written to a grid, a field which should be queried when attempts to access locations outside the grid are made, and a description of the grid itself, in terms of a `Domain` object.

```python
bump = constant(1) / (constant(1) + length(identity()))
outside = constant(0)
gridded = writeToGrid(bump, outside, domain)
```

### 1.4.4 Function Composition

Functions are often used to transform inputs to other functions. This type of operation requires function composition. In the framework, it is possible to explicitly map one function through another so that

\[ e(x) = f(g(x)) \]

can be directly created in code form:

```python
# Form a field which via function composition
e = warp(f, g)
```

Later, it will be shown that this can be used to express the movement of material by a mapping function which describes the movement of material from an “original” distribution.

### 1.4.5 Differentiation

Since scalar and vector fields objects in the framework are assumed to be at least $C^1$, there exist gradient operators which take these rank $k$ tensor fields to rank $(k + 1)$ tensor fields.

Underlying the implementation of each mathematical object is a gradient operator which appropriately handles all multi-variable chain, product, and other rules. For example, consider the
following simple function and its gradient

```plaintext
F = DOT(I, I)
F = GRAD(F)
```

This $g$ object will then evaluate to $\nabla (x \cdot x) = 2x$. In addition to the gradient, the familiar operations $\text{curl}$ and $\text{div}$ are also available.

In the case of discretized data on a grid, the gradient is still defined. The gradient on a gridded object will return a finite centered difference, which is second-order accurate with respect to the size of grid cells.

### 1.4.6 Limits on Differentiation

Because of the weak $C^1$ differentiability assumption on fields, it is not possible to compute second derivatives analytically within the framework. As an example, the following code is not valid and will generate a run-time error:

```plaintext
# Each of these will generate a run-time error!
H = GRAD(GRAD(F))
I = GRAD(LENGTH(CURL(VELOCITY)))
j = DIV(GRAD(RHO))
```

It is still possible to take second derivatives in a certain sense. Because the curl of a vector field is still a vector field, and likewise, the divergence of a vector field is a scalar field, both of which can be expressed in the framework, it is possible to construct higher order derivatives by storing intermediate results within a discretized grid.

Scalar fields and vector fields on grids are differentiable and allow for this kind of technique. For instance, the expression

$$L(\rho) = \nabla \cdot \nabla \rho(x)$$

involves second derivatives, but may be computed through a sequence of finite differences on grids via the code

```plaintext
G = grad(rho)
G = writeToGrid(G, constant(Vector3(0,0,0)), domain)
L = div(G)
```
Chapter 2

Case Study: Fluid Simulation

In visual effects production, fluid simulations are often needed in order to produce natural phenomena such as smoke and water. The framework provides a compact and flexible system for coding fluid simulation systems. In all of the following material, it should be noted that simplifications are made so that visually plausible, rather than physically accurate, fluids can be simulated.

2.1 The Dynamics of Fluids

In computer graphics, simulations of gas dynamics are governed by the Navier-Stokes Equations, which consist of a continuity equation (sometimes called the “momentum” equation) describing the time evolution of a velocity field, and a mass conservation equation which is coupled with the first equation. In the incompressible regime of with ambient density equal to one, these equations take the form

\[
\frac{\partial}{\partial t} \mathbf{u} + \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p - \nabla \cdot \sigma = f \tag{2.1}
\]

\[
\nabla \cdot \mathbf{u} = 0 \tag{2.2}
\]

\[
\frac{\partial}{\partial t} \rho + \mathbf{u} \cdot \nabla \rho = S(x, t) \tag{2.3}
\]

where \( \mathbf{u} \) is the fluid velocity, \( p \) is the pressure, \( \sigma \) is the stress tensor, relating spatial gradients in velocity to stresses present in the medium, \( f \) represents external “body” forcing, such as gravitational forces actings on the fluid, \( \rho \) is the density of a passively advected smoke density, and
$S(x, t)$ is a general source term which encompasses the introduction or deletion of smoke at different points in space/time. It is typical to multiply the first two terms of the momentum equation by a “density” (usually denoted as $\rho$), however we have made the common assumption that this is one, so that it does not participate in the dynamics.

The continuity equation relates the change in velocity to the movement of material through the domain (a process called *advection*), a pressure term which is part of enforcing incompressibility, and viscous forces which act to dampen the motion of the fluid. The mass conservation equation enforces incompressibility of the fluid, which both simplifies the simulation process, and simultaneously defines the pressure field seen in the continuity equation. In many fluid simulations used in computer graphics, it is often assumed that the stress tensor depends linearly on the components of the velocity and has a constant viscosity so that the stress tensor has a form $\sigma = \mu \nabla u$, which simplifies to $\nabla \cdot \sigma = \mu \nabla^2 u$, where $\mu$ is now the effective viscosity of the fluid. This is not always the case: there are other situations where this assumption should not be made, as in elastic media [6].

While these equations have been well studied for hundreds of years, and many careers have been invested in accurately simulating these equations, for the purposes of computer graphics, it is often enough to make approximations to these equations which create results which are visually “plausible”, rather than physically accurate.

The typical approach [13], is to use “operator splitting”, which assumes the individual components of the continuity equation can be accounted for one at a time, rather than simultaneously. This means that the advection, viscous, and incompressibility properties of the equations are handled in separate steps, rather than solving for their combined effects simultaneously.

### 2.2 Advection

One of the typical methods used in graphics for handling the non-linear term, $u \cdot \nabla u$, utilizes the method of characteristics. The method of characteristics integrates a path backwards in time along the velocity field in order to determine what quantity arrives at each particular point. While many other approaches for computing the effects of advection are numerically unstable, this process, known as Semi-Lagrangian Advection (SLA), is unconditionally stable because values determined for a future time are bounded by values at the current time. In computer graphics, this is an important criterion because it is typically not known what types of forces may created, and so it is difficult
to choose particular advection strategies which are not prone to numerical “blow-up”. In scenarios where artists and other systems may interact with fluids and produce unphysical or extremely large forces and velocities, SLA is an attractive method.

Using SLA, the quantity $q$ is advected through the flow $u$ via the update

$$
q(x, t + \Delta t) = q(x - \Delta t \cdot u(x, t), t)
$$

In terms of function composition, this procedure of advecting a field is easily encapsulated via the built-in warp function. By denoting $q_{\text{advected}}(x) \equiv q(x, t + \delta t)$ and $q(x) \equiv q(x, t)$, the advection is written as

```plaintext
# Advect the field q through the velocity field u over a period of dt time
q_advected = warp(q, identity() - dt * u)
```

Since this code is written in a Python context, it is also possible to encapsulate this functionality in a function:

```python
def advect(field, velocity, dt):
    return warp(field, identity() - dt*velocity)
```

## 2.3 Viscosity and Turbulence

Interestingly, many fluids which are simulated in computer graphics require viscosity in order to appear natural, and yet are simulated without any effects of viscosity. A Taylor series analysis of the SLA technique shows that second order terms in the advection procedure actually introduce a velocity-dependent viscosity-like diffusion term, which causes “artificial viscosity”. There are both benefits and problems with this effect. On one hand, fluids appear unnatural when simulated without some amount of viscosity. On the other hand, the fact that this false viscosity scales with the magnitude of the velocity is problematic, since this is not the original behavior dictated by the Navier-Stokes equations.

Turbulent flow is visually interesting and is a large part of what makes fluid motion distinctly different from other phenomena. In this respect, many methods have been created which attempt to inject turbulent motion or mitigate the loss of turbulent flow by using stochastic noise injection, turbulence-injecting particles, and other methods, most of which attempt to re-inject lost energy in
a realistic way. Other methods mitigate the dissipation of energy by using higher-order schemes, such as the high-order Adams-Bashford methods in the QUICK scheme [10].

Perhaps the most popular method of mitigating artificial viscosity though, is the so-called vorticity confinement method. This method attempts to re-inject rotational motion wherever it is lost in the fluid simulation by measuring location rotation by means of the curl, $\omega = \nabla \times u$. The gradient of the magnitude of this vorticity, $\eta = \nabla |\omega|$, will point in the direction of the center of a local vortex. By applying a force orthogonal to this line, the method introduces a rotational motion back into the simulation. It has been shown that linearization of this foring term yields $-\tau \nabla^2 u$, a kind of anti-dissipation term. Because there has not been any construction to date which can determine what the magnitude of this force should be (at least in the context of computer graphics, the strength of this term is often controlled by user or artistic input. Because vorticity confinement is expressed in terms of differential operators on fields, the framework provides an especially concise form for expressing it in terms of code.

```python
def vconfinement(velocity, epsilon):
    # Compute the vorticity field, $\omega = \nabla \times u$ from the velocity field
    omega = curl(velocity)
    # Compute the vector pointing towards the center of the local vortex
    eta = grad(length(omega))
    # Compute the vorticity confinement force and scale based on user input
    N = normalize(eta)
    return constant(epsilon) * cross(N, omega)
```

## 2.4 Enforcing Incompressibility

For simplicity, fluids in computer graphics are often assumed to be incompressible, signified in part by the mass conservation equation

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

This incompressibility condition states that the net amount of material flowing in and out of a infinitessimal small region of space must be zero. The inability of material to arbitrarily compress into or flow out of a region of space is why rotational motion occurs in fluids. Since turbulence (again, one of the main draws to simulating fluids for graphics in the first place) is highly dependent
upon this kind of behavior, it is paramount that this condition be enforced.

There have been only a few different methods developed for satisfying this condition. By far, the most popular methods used in graphics utilizes the vector field decomposition theorem due to Hodge. A vector field $\mathbf{u}$ can be decomposed into a divergence-free part, and a curl-free part

$$
\mathbf{u} = \mathbf{w} + \nabla \phi
$$

$$
\nabla \cdot \mathbf{w} = 0
$$

$$
\nabla \times \nabla \phi = 0
$$

This decomposition suggests a simple method for enforcing incompressibility in velocity fields. In the spirit of operator splitting, after velocity advection and viscosity terms have been handled, the simulator has an intermediate velocity field $\mathbf{u}^*$ which is not divergence-free. By using this Hodge decomposition, it is possible to project the vector field into the space of divergence-free vector fields, and thus satisfy the incompressibility constraint. The projection process is given as follows. First, taking the divergence of both sides

$$
\nabla \cdot \mathbf{u} = \nabla \cdot \mathbf{w} + \nabla \cdot \nabla \phi
$$

$$
\nabla \cdot \mathbf{u} = \nabla^2 \phi
$$

where we have used the property that $\nabla \cdot \mathbf{w} = 0$. This last equation, with appropriate boundary conditions, is a Poisson equation for the scalar field $\phi$. In the discrete setting, this problem describes a linear system of equation in terms of the $3n$ velocity degrees of freedom of the discrete system. After the values of $\phi$ have been computed, the last step of the project is to compute $\mathbf{w} = \mathbf{u} - \nabla \phi$, which is now divergence-free.

This system of equations is typically solved in terms of solving sparse linear algebraic systems. Because the Laplacian is a symmetric positive definite operator when defined with the appropriate boundary conditions, methods like Conjugate Gradient are popular methods of solution. In the framework presented here, a spectral method is used instead. The Fourier Transform has the remarkable property of converting linear differential operators and linear equations into algebraic operators and equations in the “frequency” domain. Assuming that the velocity $\mathbf{u}$ goes to zero at infinity, and assuming no boundaries, the Fourier Transform of the divergence operator on $\mathbf{u}$ has the
\[ F(\nabla \cdot \mathbf{u}) = i(\mathbf{k} \cdot \mathbf{\hat{u}}) \]

which gives a convenient method for projection. Because the divergence-free condition is satisfied when \( \mathbf{\hat{u}} \) is orthogonal to \( \mathbf{k} \), the divergence-free portion of \( \mathbf{u} \) is given by projecting the Fourier representation \( \mathbf{\hat{u}} \) to the plane orthogonal to \( \mathbf{k} \)

\[
\mathbf{\hat{u}} \leftarrow \mathbf{\hat{u}} - \frac{(\mathbf{\hat{u}} \cdot \mathbf{k})}{|\mathbf{k}|^2} \mathbf{k}
\]

In the framework, this process of projection via Fourier Transforms is given by the function \texttt{divfree}, which takes a vector field as an argument, and returns the closest divergence-free vector field (in the \( L^2 \) sense) as a result. Because of the nature of these discrete solutions, the returned vector field is always a discretized grid. In addition to the vector field which is to be projected, the user must also provide a \texttt{Domain} object for specifying the location and resolution of the grid on which the projection will be computed.

## 2.5 A Compact Fluid Solver

Putting all of these components together, we can write a short program for simulating the Navier-Stokes equations. We will simulate the evolution of a bouyant cloud of smoke which is constantly injected near the bottom of the domain. First, defining the domain that our simulation will live in (required for vector field projection), the velocity field \( \mathbf{u} \), the density field, and our time step,

```python
domain = Domain(64,64,64,Vector3(-1,-1,-1), Vector3(1,1,1))
velocity = constant(Vector3(0,0,0))
source = mask(.25 - length(identity()) - constant(Vector3(0,-.7,0))))
density = source
dt = constant(.1)
```

After these entities have been defined, we can step forward in time, advecting density through the velocity field and evolving the velocity field by the continuity equation

```python
for frame in range(400):
    # Compute the vorticity confinement force
    vcforce = vconfinement(velocity, .04)
```
# Adveqt the density field through the velocity field

density = advect(density, velocity, dt)
density = writeToGrid(density, constant(0), domain)

# Evolve the velocity field via the momentum equation

force = rho * constant(Vector3(0,1,0)) + vforce
velocity = advect(velocity, velocity, dt) + force * dt
velocity = divfree(velocity, domain)

It is worth noting that density should be written to a grid after each step of advection; otherwise, the distribution of density on the $k$-th frame (needed for advancing to the next time step and for rendering) will involve evaluating the advection for the preceding $k - 1$ frames up to that point in the simulation. This process of advecting density procedurally without resampling to a grid is called Gridless Advection [14]. This method avoids the diffusion of density due to artificial viscosity but can take an extraordinary amount of time to both simulate and render.

## 2.6 Simulating the Rayleigh-Taylor Instability

The Rayleigh-Taylor instability is an often-studied instability in fluid dynamics caused by non-zero density gradient across an interface embedded in the fluid. While Rayleigh-Taylor instabilities have been well studied and solved to high accuracy in a number of scenarios, it is possible to mimic this behavior, albeit with low accuracy, using very little code in this framework:

```python
# Generate a signed distance function for plane

def plane(point, normal):
    return dot(identity() - constant(point), constant(normal))
```
# Initialize the simulation by creating two thin extruded slabs near the origin
red = mask(plane(Vector3(0,0,0), Vector3(0,1,0)))
red *= mask(plane(Vector3(0,.1,0), Vector3(0,-1,0)))
blue = mask(plane(Vector3(0,-.1,0), Vector3(0,1,0)))
blue *= mask(plane(Vector3(0,0,0), Vector3(0,-1,0)))

Here, the same cubic domain as the previous example has been used, but the density has been separated into two components, “red” and “blue”. All of the points in the range $0 \leq y \leq 0.1$ plane are assigned as red, and all of the points in the range $-0.1 \leq y < 0$ are marked as blue. Applying a force upward for the blue density and a force downward for the red density causes a zero net force at the interface between them until the symmetry is broken.

```
for frame in range(400):
    # Compute the vorticity confinement force
    vc_force = vconfinement(velocity, .08)

    # Compute forces
    force = (blue - red) * constant(Vector3(0,1,0)) + vc_force

    # Advect quantities through the flow
    red = advect(red, velocity, dt)
    blue = advect(blue, velocity, dt)

    # Evolve velocity
    velocity = advect(velocity, velocity, dt) + dt * force
    velocity = divfree(velocity, domain)

    # Write densities to grids
    red = writeToGrid(red, constant(0), domain)
    blue = writeToGrid(blue, constant(0), domain)

    render(red, blue)
```

Utilizing the gridless advection approach [14], where density is not resampled to a grid, much more detail can be obtained.
Figure 2.2: Simulation showing turbulent mixing in the Rayleigh-Taylor instability. The sequence on the left is rendered with density resampled to a grid each frame. The sequence on the right is rendered by gridless advection.
Using these techniques, many different types of fluid phenomena can be modeled. The preceding examples demonstrate how different effects can be quickly created using a minimum amount of code. Focusing efforts on experimentation rather than tedious implementation allows for one to explore new ideas quickly.
Chapter 3

Case Study: Blending Fluid Simulations with the Characteristic Map

Fluid simulations in production are often difficult to direct, take a considerable amount of time to compute, and cannot be easily reused for other uses or effects without redoing the simulation. A method is presented, based on the Characteristic Mapping (CM), which reuses simulation data and achieve new effects using a previously-generated simulation. By using a collection of operations on the CM, such as transformations, blends, and compositing, it is possible to mix simulations into novel situations, such as explosions and targeted motion.

Fluid simulation is an important aspect of modern graphics and is used extensively in many scenarios. It is, however, often difficult to control or direct the appearance because of the chaotic physics underlying fluids. In this paper a new methodology for manipulating fluid and volume data as well as techniques for creating novel motions from a single simulation is presented.

A vector field construction, the Characteristic Map (CM), is used to create art-directable fluid-like motion which may have completely different appearance and style from the original underlying simulation data. The framework presented in this thesis is used to manipulate this map and demonstrates several ways to combine and augment fluid motion in order to produce new effects without conducting additional fluid simulations.
3.1 Past Work

Volume rendering of fluids such as smoke is typically done via directly ray marching density fields on grids. Shading participating media such as smoke is a well-researched topic which has seen several competitive methods. Volume photon mapping has been used to compute shading of dense smoke by simulating individual scattering events. While this method is attractive, realistic results require a large number of samples, and corresponding a large amount of memory. Deep Shadow Maps (DSM) [9] have also been used to precompute and store light attenuation in a grid for fast render-time queries. DSMs are effective because they enable constant time queries for lighting information.

The Characteristic Map [15] was introduced as a method to preserve thin details and areas of high spatial gradients of advected material. The method of characteristics used in the CM is perhaps best known from [13] where the transport of material is phrased in terms of tangent curves of the underlying flow field. This method has also seen extensive use in the solution to other PDEs and forms the analytical basis for several solutions to non-linear systems where shocks and high gradients are difficult to capture in other approaches. [4] The method of characteristics is attractive here since its geometric approach to the solution of a system is well-understood and allows for us to construct notions of advection and displacement in an intuitive way.

As was shown in [2], one of the principle issues in Semi-Lagrangian advection is that density is diffused over space as the simulation progresses. This loss of density has been partially mitigated by using more costly but accurate integration schemes, such as QUICK [10], WENO [11], and BFECC [5]. In computer graphics, one major focus of advection schemes is the notion of unconditional stability. Schemes with this property are not prone to numerical “blow up” because they compute values which are bounded by those in the domain. Numerical schemes with unconditional stability often suffer from relatively low accuracy. This loss of accuracy is typically manifested through a form of artificial energy loss not found in the original system. FLIP [2], a method which advects field quantities by using a hybrid particle/mesh representation, has been advocated as a high-accuracy, low-dissipation method, but it can be difficult to uniformly distribute particles in turbulent flow. Other, completely grid-based schemes, such as the high order Adams-Bashford-based integration techniques [10] allow for very turbulent fluids, but are also expensive and lack unconditional stability.

In terms of the CM, this density diffusion is effectively supressed since the density is not
explicitly advected at any time, and numerical diffusion is transferred to the CM. The diffusion erodes fine spatial structure in the CM, but not the overall shape of the mapping field. Because of this, the CM has been shown to preserve the fine features and large gradients that are desirable in fluid phenomena without any appreciable sacrifice in simulation or rendering speed.

[3] resolves fine interface details of a body deforming under fluid-like flow by using a surface mesh advected by flow primitives. While this method is attractive for its ease of implementation, it does not couple the velocity to the density during simulation, which limits possible applications of the technique.

Recently, [16] presented a method for visualizing flow features based on characteristic lines. They use the method of characteristics to identify different geometric properties within the flow and to track these features over time. While not focusing on volume rendering, their work echoes the concept that characteristic lines can extract a wealth of geometric information from fluid flow for visualization.

### 3.2 The Characteristic Map

The CM is a time-varying vector field which tracks how the flow transports material to a point in space. The CM evolution follows from the advection of spatial coordinates and is computed using the velocity field of a fluid simulation. The evolution of the characteristic map is governed by the continuity equation

\[
\frac{\partial X(x, t, t')}{\partial t} + u(x, t) \cdot \nabla X(x, t, t') = 0
\]  

(3.1)

where \( X : \mathbb{R}^5 \rightarrow \mathbb{R}^3 \) is the mapping function and \( u \) is the incompressible velocity field at time \( t \). The mapping function has three arguments which define the flow which begins at \( x \) at time \( t' \) and takes place during the time interval \([t', t]\). Intuitively, the map explains where material “came from” after some period of flow. The density at a particular time can be phrased in terms of it’s initial distribution and source \( S(x, t) \):

\[
\rho(x, t) = \rho_0(X(x, t, 0)) + \int_0^t ds \ S(X(x, t, s), s)
\]  

(3.2)

The additional integral term accounts for density that is injected over time. This particular way of phrasing the solution is handy because it easily lends itself to numerical implementation.
When it is necessary to query for density, \( \rho_0(\mathbf{x}(\mathbf{x}, t, 0)) \) maps the point \( \mathbf{x} \) to where it was originally located at the beginning of the flow. In the case of a non-zero source (e.g., injecting density over time), additional mapping functions are created for each frame in which density is injected. A graphical description of this process is presented in figure 3.1.

From now on, when the notation \( \mathbf{X}(\mathbf{x}, t) \) is used, it is implied that \( t' = 0 \) so that the map accounts for flow from the “beginning” of the flow.

Figure 3.1: During volume rendering, the CM, \( \mathbf{X} \), is used to map the initial density distribution \( \rho_0 \) to a new position \( \mathbf{X}(\mathbf{x}) \) due to fluid flow.

Because of this representation, the density distribution can be changed at render time with no additional render-time cost. An examples of this change in density is given in figure ??.

### 3.2.1 Semi-Lagrangian Mapping

SELMA (SEmi-Lagrangian MApping) is a particular method of implementing the characteristic mapping which uses the unconditionally-stable semi-Lagrangian method of advection to evolve the characteristic mapping function and sample the new CM to a grid.

One very attractive feature of SELMA is its ease of implementation [15].

In practice, SELMA is simply a set of gridded vector fields, one defined at each time step. The SELMA grid is initialized so that it is the identity mapping, i.e.

\[
\mathbf{X}^0(\mathbf{x}) = \mathbf{x}
\]
By applying semi-Lagrangian advection, we can update the grid using the velocity at the current time:

\[ X^k(x) = X^{k-1}(x - u(x, t^k) \Delta t) \]

Under the viewpoint of entire fields, all that is needed for calculating the SELMA is a primitive for computing semi-Lagrangian advection, which most fluid simulators already make use of.

In the framework, this means that at time \( k \), an advected density field is presented to the renderer by making use of the warp function. Following the definition in equation 3.2, the advected density can be described succinctly in terms of the initial density \( \rho_0 \) and the mapping function at time \( k \) by \( \text{warp}(\rho_0, X_k) \).

Because the computation of the CM only requires a general method of advecting material through a velocity field, it is also possible to use lower dissipation schemes such as BFECC or FLIP. These schemes provide better accuracy at the expense of stability or ease of implementation. Even though these higher-order schemes exist, in this thesis, it has been found that semi-Lagrangian advection of the CM dramatically improves the visual quality of the fluid structure.

There is a marked difference between the visual style of SELMA-based rendering and the traditional method of ray marching density. Thin features and interfaces are preserved by SELMA, but lost in traditional density volume rendering. In figure 3.2, it is clear that the surface of the advected material is poorly resolved due to diffusion, but in SELMA the same areas are well resolved and have no apparent diffusion.

If the advected CM is not written to a grid, then this mapping instead takes the form of gridless advection. In gridless advection, there may be increased visual detail because of the lack of dependence on grid resolution, however, this will cause increased computational overhead due to multiple evaluations of velocity fields.

### 3.3 Displacement Composition for Explosions

A spherical explosion is created by compositing multiple copies of the original CM data. Though the original “shape” of the buoyant smoke is an upward blooming motion, a sum of randomly rotated displacement fields effectively copies this behavior over a broader volume. For rotations, a uniformly random point on the unit sphere is chosen as an axis and then coupled with a uniformly random angle by which to rotate. The displacement field is rotated by this angle-axis pair and
Figure 3.2: SELMA-based rendering (left) preserves much more detail when compared to more traditional approaches of rendering explicitly-advected density (right). This ability to capture high spatial gradient enables the rendering of sheets of material, with no virtually no density diffusion.

accumulated into the final displacement field. For this example, a sum of 20 copies of the displacement field is used. The pseudocode to this effect is presented in algorithm 1. A sample from an animation demonstrating this effect is given in figure 3.3. The effect is visually plausible because visually isolating a single motion from the numerous copies of the turbulent displacement field is difficult.

The final map used in creating the explosion, $X_E$, is thus created by a uniformly-weighted sum of rotated mapping functions

$$X_E(x, t) = x + \sum_{i=0}^{20} (R_i^{-1} X(R_i x, t) - x)$$

where the $R_i$ encodes matrices of uniformly random rotation. The inverse of these matrices is needed in order to bring the displacements into the world reference frame after the original rotation.

<table>
<thead>
<tr>
<th>Algorithm 1: Pseudo-code used in compositing SELMA data for a spherical explosion.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> SELMA Data $X^k$ for the current frame of animation, and the initial density distribution $\rho_0$.</td>
</tr>
<tr>
<td><strong>Output:</strong> Displaced density $\rho_{final}$</td>
</tr>
<tr>
<td>$D = X^k - x$;</td>
</tr>
<tr>
<td>$S = 0$;</td>
</tr>
<tr>
<td>for $i = 1$ to 20 do</td>
</tr>
<tr>
<td>$\text{dir} = \text{randomDirection}()$;</td>
</tr>
<tr>
<td>$\phi = \text{randomAngle}()$;</td>
</tr>
<tr>
<td>$S += \text{rotate}(D, \text{dir}, \phi)$;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>$\rho_{final} = \text{warp}(\rho_0, S + x)$;</td>
</tr>
</tbody>
</table>
3.3.1 Morphing Between Two Shapes

Another application of the CM exploits the representation of density and allows one to do fluid morphing between two density distributions. Previous physically-based methods for blending between two shapes have been presented, however, they require introducing forces during simulation. In this new method, two shapes can be blended without the need for any particular forces or flow patterns. In this example, a scene has been set up using two objects, the Stanford bunny and the Armadillo. The original CM data describes a flow which travels upwards, and so copying and rotating the CM data will generate another mapping which flows downward. The first half of the animation flows upward using displacements from the original data. During the middle of the animation, smooth blend takes place from the upward displacement field to the downward displacement field, and the density being used to render is transitioned from one object to the other during the turbulent phase. To insure that the flow is smooth, the fact that the bouyant smoke simulation begins with zero displacement is utilized. This means that both the beginning and ending of the new composited mapping will have zero velocity, giving a smooth appearance. The blending between the two maps is designed to happen only in a particular range of time which encompasses the turbulent phase of flow, which diguises the transition in densities.

Mathematically, the rendered density is defined in terms of the original and rotated mapping
functions which span the time range $t \in [0, T]$ as

$$
\rho(x, t) = \alpha(t)\rho_1(X(x, t)) \\
+ (1 - \alpha(t))\rho_2(x + X^*(x, T - t))
$$

(3.4)

where $X^*$ is a rotation of the displacement encoded by the mapping function:

$$
X^*(x, t) = R^T X(Rx, t) - x
$$

In this work, a linear ramp is used for interpolation, i.e. $\alpha(t) = 1 - t$. This interpolation profile can be varied artistically to control the rate of blending and the window of time in which the density blending occurs.

---

**Algorithm 2**: Pseudo-code used for blending two shapes via the equation 3.4

| Input: SELMA Data $X(x, t)$ generated from a fluid simulation, and the intial density distributions for two shapes $\rho_1$ and $\rho_2$. |
| Output: Blended density $\rho_{blended}(x, t)$ |
| $\alpha = 1 - t/T$; |
| $X_1 = X(x, t)$; |
| $X_2 = X^*(x, t) = R^T X(Rx, T - t)$; |
| $\rho_{blended}(x, t) = a\text{warp}(\rho_1, X_1) + (1 - \alpha)\text{warp}(-\rho_2, X_2)$; |
3.4 Conclusion

The Characteristic Map is an attractive method for assembling new effects from existing fluid simulations. It gives new kinds of control over chaotic flow and enables the addition of flow features without executing an expensive fluid simulation.

These new ideas could be integrated with complementary production tools for creating volumes and presenting artists with controls for the parameters and functions defined here. The only “extra” feature that must be added to volume rendering software in order to accommodate rendering density with the CM is a mechanism for procedurally evaluating the mapping function.
Chapter 4

Implementation

The framework defined in this thesis is implemented using a combination of the C++ and Python programming languages, in order to provide a combination of fast computation and flexible and interactive interface. In this chapter, the representation and implementation details of tensor field arithmetic and differential calculus within the framework are discussed.

4.1 Implementation Details and Optimizations

Within the framework, the fundamental type is the tensor field, which permits evaluation at points in space. These fields may generally depend on other fields via a composition of operations, and as a result, define expressions. These expressions are exact in the sense that they symbolically represent the underlying concept rather than discretized data. This resolution-independent encoding enables the user/programmer to represent items at various levels of detail and with a custom memory/speed trade-off.

In order to represent these expressions, the notion of an expression tree is employed. In an expression tree, each node which is not a leaf represents some operation (possibly unary or binary in this framework) acting on its children. At the leaves of the tree are nodes which can be evaluated independently of all other fields, such as constant values, discretized grid data, or noise functions.

The definition of expressions in terms of an expression tree is both convenient and powerful. Tree-based expressions are simple to parse and transform, can be used to generate code in other target languages, and perhaps most importantly allow for delayed evaluation. This so-called "lazy
evaluation” means that the construction of the tree does not require actually evaluating anything and as such, the tree itself is lightweight in the sense of compute and memory requirements. Values are only computed when they are needed, which allows for the framework to have a notion of resolution independence, and eliminates discretization error until discretization becomes necessary.

Because the framework is centered on exploring and manipulating 3D data in ways typical of computer graphics, there are three available types of tensor fields: scalar, vector, and matrix fields. These three different field types share a common interface in the sense that they may be evaluated (and possibly differentiated) at points in space, and only differ by what values are returned.

From an Object Oriented Programming (OOP) viewpoint, nodes within the expression tree may return values based on various mappings of their input arguments. For example, an addition node will have two children and when evaluated will evaluate its children and return the addition of these result to its parent. Because the nodes share this common evaluation requirement, all nodes within the tree are termed “Field Nodes” and must provide functionality for evaluating and taking a gradient. In order to facilitate operator overloading and to avoid an explosion in the required number of operator definitions arising from many node types, expression trees are stored inside a separate template class “Field”, which has three relevant specializations: ScalarField, VectorField, and MatrixField.

This Field class stores a shared pointer to the underlying expression tree and may be evaluated at a point in space. Calling eval(...) on a Field object causes the underlying expression tree to be evaluated recursively, beginning at the root of the tree. The choice to use shared pointers
comes from the goal of making expression tree memory management simple and robust to various situations. Vanilla pointers would allow for multiple Field objects to reference an expression tree, but would make determining the appropriate time to destroy that tree difficult. Since Field objects use shared pointers, when all Field objects that are accessing an expression tree go out of scope, the underlying tree and any allocated memory is automatically freed.

### 4.2 Interactive Scripting

Because the framework is meant to be interactive and free-form, bindings were created for Python via SWIG. SWIG is an open source software tool for creating bindings from compiled languages to scripting languages, as in this case from C++ to Python. This exposes any desired functionality in C++ to the user by allowing the Python interpreter to make calls to compiled code. The process of calling native code from Python does incur performance penalties due to the fact that Python must resolve variable types and functions at runtime and save and restore FPU environment state before and after calling native code. However, because the bulk of the processing and evaluation of expression trees (and rendering of this data) takes place inside of these native functions, these conversion penalties are relatively few. In this sense, Python is essentially allowing the user to interactively (or via a script) create C++ objects which have the advantage of optimized evaluation.

Typically many of the FieldNode operations could be templatized to reduce code duplication, as in the case of addition of like tensor fields which only differ in their type. Because Python does not have a notion of templated object types, SWIG is unable to translate this kind of code in a clean way that is invisible to the user. This results in a fair amount of code duplication in the internal parts of the framework. This problem is partially mitigated by the use of macros and multiple expansions with different arguments. Unfortunately this means that every combination of arithmetic operators must be created by hand, whereas a templated version of the code could infer types at compile time.
and reduce code duplication. While this is inconvenient to implement and maintain, there is no effect whatsoever on the end user.

Python was chosen as the target scripting language because of its high availability, extensive built-in library, ease of use, and popularity in graphics. Many other projects exposing numeric computing have also been ported to Python, such as NumPy, an open source discrete data processing and computation system which mimics the functionality of Matlab. These systems allow for fast prototyping and evaluation of ideas, while utilizing file and data manipulation constructs available to Python.
Chapter 5

OpenCL-Based Acceleration

The operator tree representation employed in this framework lends itself to code generation in other languages. In this chapter, it is shown how procedural fields are translated directly from the framework into working OpenCL code for fast, parallelizable volume rendering.

5.1 OpenCL as a Parallel Computation Framework

OpenCL is a open framework and API for parallel computation on heterogenous compute systems [7]. It allows programmers to develop parallel processing code in a single language and deploy to different architectures, such as CPU, GPU, and FPGAs. Each hardware vendor must provide a compiler which will compiler OpenCL code to the target architecture.

Computation takes place in user-written kernels which are a set of functions which may run on the Device, which is the actual parallel-capable hardware. On the Host, an API allows for the allocation/deallocation of memory, task scheduling, and data transfer between the Host and Device.

In this thesis, preliminary tests of automatically converting operator trees to working OpenCL code were performed and given to a generic volume rendering system.

5.2 Generating Functions from the Operator Graph

In the operator tree, the order to compute the underlying expression is well-defined. Because of inner nodes of the tree depend only on the values of children, it is sufficient to compute the values
of children and then the current node based on these values. Because the operator tree representation is inherently acyclic, it is impossible for any of the evaluations of node in the tree to feature a cyclic dependency. This means that a postorder traversal of the operator tree will be the proper order to evaluate nodes in order to assure that information needed for any particular evaluation has already been computed. As a reminder, an inorder traversal of this tree will produce the usual infix notation of the expression that the tree encodes.

Second, actual code for each node/function is generated. The functions themselves take only one argument (where they should be evaluated in space), and have the form

```c
ReturnType func_1234(const vec3 x) {
    CType1 P1 = func_xxxx(x);
    CType2 P2 = func_yyyy(x);
    ...
    return P1 + P2;
}
```

where ReturnType may be `float`, `vec3`, or `mat3`, depending on the tensor rank associated with the FieldNode that the function encodes. The first section of each function defines variables which are given values based on evaluation of the child nodes. These function names are known because every node has a simple name based only on the node number, which has already been computed. The number of variables in this section will be equal to the number of children under this node in the operator tree. It follows that nodes which generate data but have no child nodes (the leaves of the operator tree) will have no variables declared.

The name of these variables follows a predictable pattern of increasing numbers so that template code forming the substance of the function for the node can reference its arguments in a predictable way.

The second section is defined by a code template. In a function, this template defines the actual functionality of the node. The template for addition node may be something like

```c
return P1 + P2;
```

This kind of template code is defined per Field Node so that each type can be used in generating code for OpenCL.
5.3 Example execution and results

As an example, consider the following script, which produces a signed distance function for a unit sphere centered at the origin:

\begin{verbatim}
sphere = constant(1) - length(identity())
print generateOpenCLCode(sphere.emitTree(), 'evalDensity')
\end{verbatim}

The method `emitTree()` is available to all nodes in the framework which support code generation. Upon calling this function, the framework will output a language-agnostic form of the operator tree which is suitable for translation into another language.

This intermediate form is then translated by the function `generateOpenCLCode()`, which performs the node numbering and code generation steps explained above and outputs a string of OpenCL code which can be integrated with a volume rendering system for rendering. The second argument to `generateOpenCLCode()` is the name of the function which should be callable in order to evaluate the root of the tree. It is necessary for the user to supply this name since the tree itself has no implicit function name.

In this thesis, the output of the `generateOpenCL()` function is combined with a simple OpenCL direct volume rendering kernel. Despite performing no optimization, several examples can be rendered at realtime/interactive rates.
In figure 5.3, a screenshot of a very simple scripted volume is shown. The python script and the automatically generated OpenCL code counterpart are given in the appendix.

5.4 Future Direction

While the examples here demonstrate the possibility for creating OpenCL code automatically from the operator tree, they do not allow for arbitrary expressions. Future work is needed in order to automate the handling of allocation, management, and synchronization of objects between the host and device.

Discretized grids are needed for many models and procedures, however, synchronizing their state with OpenCL is nontrivial. The destruction of a node on the host requires destruction of the associated OpenCL buffer and its representation. Coordinating these allocations and deallocation in an automatic way is an interesting future direction, especially since it is one of the only major hurdles in allowing fully general scripts to run on the device.
Appendices
Appendix A  Python Script Example for Automatically Generated OpenCL Code

```python
from tracer import *
from OpenCLCodeGen import *
import os, random

random.seed(13)

def randomVector3(a,b):
    x = a + (b-a)*random.random()
    y = a + (b-a)*random.random()
    z = a + (b-a)*random.random()
    return Vector3(x,y,z)

def sphere(radius, center):
    return Constant(radius) - length(identity()) - constant(center))

s = mask(sphere(1., Vector3(0,0,0)))
for i in xrange(10):
    s = s - mask(sphere(random.random(), randomVector3(-1,1)))

code = generateOpenCLCode(s.emitTree(), 'evalDensity')
print code
clf = open('_oclrender.cl', 'w')
clf.write( code + '\n' + open('opencl-utilities/volrender.cl').read() )
clf.close()

os.system('./opencl-utilities/volrender _oclrender.cl')
```

Appendix B  Automatically-Generated OpenCL Code

```c
inline float func_86(float4 x);
inline float func_85(float4 x);
inline float func_84(float4 x);
inline float func_83(float4 x);
inline float func_82(float4 x);
inline float func_81(float4 x);
inline float func_80(float4 x);
inline float func_79(float4 x);
```
inline float func_78(float4 x);
inline float func_77(float4 x);
inline float func_76(float4 x);
inline float func_75(float4 x);
inline float4 func_74(float4 x);
inline float4 func_73(float4 x);
inline float4 func_72(float4 x);
inline float func_71(float4 x);
inline float func_70(float4 x);
inline float func_69(float4 x);
inline float func_68(float4 x);
inline float func_67(float4 x);
inline float4 func_66(float4 x);
inline float4 func_65(float4 x);
inline float4 func_64(float4 x);
inline float func_63(float4 x);
inline float func_62(float4 x);
inline float func_61(float4 x);
inline float func_60(float4 x);
inline float func_59(float4 x);
inline float4 func_58(float4 x);
inline float4 func_57(float4 x);
inline float4 func_56(float4 x);
inline float func_55(float4 x);
inline float func_54(float4 x);
inline float func_53(float4 x);
inline float func_52(float4 x);
inline float func_51(float4 x);
inline float4 func_50(float4 x);
inline float4 func_49(float4 x);
inline float4 func_48(float4 x);
inline float func_47(float4 x);
inline float func_46(float4 x);
inline float func_45(float4 x);
inline float func_44(float4 x);
inline float func_43(float4 x);
inline float4 func_42(float4 x);
inline float4 func_41(float4 x);
inline float4 func_40(float4 x);
inline float func_39(float4 x);
inline float func_38(float4 x);
inline float func_37(float4 x);
inline float func_36(float4 x);
inline float func_35(float4 x);
inline float4 func_34(float4 x);
inline float4 func_33(float4 x);
inline float4 func_32(float4 x);
inline float func_31(float4 x);
inline float func_30(float4 x);
inline float func_29(float4 x);
inline float func_28(float4 x);
inline float func_27(float4 x);
inline float4 func_26(float4 x);
inline float4 func_25(float4 x);
inline float4 func_24(float4 x);
inline float func_23(float4 x);
inline float func_22(float4 x);
inline float func_21(float4 x);
inline float func_20(float4 x);
inline float func_19(float4 x);
inline float4 func_18(float4 x);
inline float4 func_17(float4 x);
inline float4 func_16(float4 x);
inline float func_15(float4 x);
inline float func_14(float4 x);
inline float func_13(float4 x);
inline float func_12(float4 x);
inline float func_11(float4 x);
inline float4 func_10(float4 x);
inline float4 func_9(float4 x);
inline float4 func_8(float4 x);
inline float func_7(float4 x);
inline float func_6(float4 x);
inline float func_5(float4 x);
inline float func_4(float4 x);
inline float4 func_3(float4 x);
inline float4 func_2(float4 x);
inline float4 func_1(float4 x);
inline float4 func_0(float4 x);

// Auto-generated function for node type: SFSubNode
inline float func_86(float4 x) {
    float P0 = func_78(x);
    float P1 = func_85(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_85(float4 x) {
    float P0 = func_84(x);
    return P0 > 0 ? 1.0 : 0.0;
}
// Auto-generated function for node type: SFSSubNode
inline float func_84(float4 x) {
    float P0 = func_79(x);
    float P1 = func_83(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_83(float4 x) {
    float4 P0 = func_82(x);
    return length(P0);
}

// Auto-generated function for node type: VFSSubNode
inline float4 func_82(float4 x) {
    float4 P0 = func_80(x);
    float4 P1 = func_81(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_81(float4 x) {
    return (float4)(0.596517, 0.971362, 0.381845, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_80(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_79(float4 x) {
    return (float)(0.819644);
}

// Auto-generated function for node type: SFSSubNode
inline float func_78(float4 x) {
    float P0 = func_70(x);
    float P1 = func_77(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_77(float4 x) {
    float P0 = func_76(x);
\[ \text{return } P_0 > 0 \ ? 1.0 : 0.0; \]

// Auto-generated function for node type: SFSSubNode
inline float func_76(float4 x) {
    float P0 = func_71(x);
    float P1 = func_75(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float4 func_75(float4 x) {
    float4 P0 = func_74(x);
    return length(P0);
}

// Auto-generated function for node type: VFSSubNode
inline float4 func_74(float4 x) {
    float4 P0 = func_72(x);
    float4 P1 = func_73(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_73(float4 x) {
    return (float4)(-0.140355, -0.372477, -0.765280, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_72(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_71(float4 x) {
    return (float)(0.364203);
}

// Auto-generated function for node type: SFSSubNode
inline float func_70(float4 x) {
    float P0 = func_62(x);
    float P1 = func_69(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
```c
inline float func_69(float4 x) {
    float P0 = func_68(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSNode
inline float func_68(float4 x) {
    float P0 = func_63(x);
    float P1 = func_67(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_67(float4 x) {
    float4 P0 = func_66(x);
    return length(P0);
}

// Auto-generated function for node type: VFSNode
inline float4 func_66(float4 x) {
    float4 P0 = func_64(x);
    float4 P1 = func_65(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_65(float4 x) {
    return (float4)(0.510008, 0.128277, -0.140845, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_64(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_63(float4 x) {
    return (float)(0.486786);
}

// Auto-generated function for node type: SFSNode
inline float func_62(float4 x) {
    float P0 = func_54(x);
    float P1 = func_61(x);
    return P0 - P1;
}
```

42
// Auto-generated function for node type: ScalarMaskField
inline float func_61(float4 x) {
    float P0 = func_60(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_60(float4 x) {
    float P0 = func_55(x);
    float P1 = func_59(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_59(float4 x) {
    float4 P0 = func_58(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_58(float4 x) {
    float4 P0 = func_56(x);
    float4 P1 = func_57(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_57(float4 x) {
    return (float4)(0.587541, -0.486584, 0.699878, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_56(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_55(float4 x) {
    return (float)(0.949323);
}

// Auto-generated function for node type: SFSubNode
inline float func_54(float4 x) {
    float P0 = func_46(x);
    float P1 = func_53(x);
return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_53(float4 x) {
    float P0 = func_52(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SSubNode
inline float func_52(float4 x) {
    float P0 = func_47(x);
    float P1 = func_51(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_51(float4 x) {
    float4 P0 = func_50(x);
    return length(P0);
}

// Auto-generated function for node type: VFSSubNode
inline float4 func_50(float4 x) {
    float4 P0 = func_48(x);
    float4 P1 = func_49(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_49(float4 x) {
    return (float4)(0.612094, 0.652731, 0.489500, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_48(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_47(float4 x) {
    return (float)(0.809740);
}

// Auto-generated function for node type: SSubNode
inline float func_46(float4 x) {

```c
float P0 = func_38(x);
float P1 = func_45(x);
return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_45(float4 x) {  
  float P0 = func_44(x);
  return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSNode
inline float func_44(float4 x) {  
  float P0 = func_39(x);
  float P1 = func_43(x);
  return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_43(float4 x) {  
  float4 P0 = func_42(x);
  return length(P0);
}

// Auto-generated function for node type: AFSubNode
inline float4 func_42(float4 x) {  
  float4 P0 = func_40(x);
  float4 P1 = func_41(x);
  return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_41(float4 x) {  
  return (float4)(-0.448326, -0.706579, 0.742569, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_40(float4 x) {  
  return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_39(float4 x) {  
  return (float)(0.014432);
}
```
// Auto-generated function for node type: SFSubNode
inline float func_38(float4 x) {
    float P0 = func_30(x);
    float P1 = func_37(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_37(float4 x) {
    float P0 = func_36(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_36(float4 x) {
    float P0 = func_31(x);
    float P1 = func_35(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_35(float4 x) {
    float4 P0 = func_34(x);
    return length(P0);
}

// Auto-generated function for node type: VSubNode
inline float4 func_34(float4 x) {
    float4 P0 = func_32(x);
    float4 P1 = func_33(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_33(float4 x) {
    return (float4)(-0.136839, 0.675313, 0.216804, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_32(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_31(float4 x) {
    return (float)(0.294657);
// Auto-generated function for node type: SSubNode
inline float func_30(float4 x) {
    float P0 = func_22(x);
    float P1 = func_29(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_29(float4 x) {
    float P0 = func_28(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SSubNode
inline float func_28(float4 x) {
    float P0 = func_23(x);
    float P1 = func_27(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_27(float4 x) {
    float4 P0 = func_26(x);
    return length(P0);
}

// Auto-generated function for node type: VSubNode
inline float4 func_26(float4 x) {
    float4 P0 = func_24(x);
    float4 P1 = func_25(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_25(float4 x) {
    return (float4)(-0.739574, 0.062630, -0.572185, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_24(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
```c
inline float func_23(float4 x) {
    return (float)(0.734024);
}

// Auto-generated function for node type: SFSubNode
inline float func_22(float4 x) {
    float P0 = func_14(x);
    float P1 = func_21(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_21(float4 x) {
    float P0 = func_20(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_20(float4 x) {
    float P0 = func_15(x);
    float P1 = func_19(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_19(float4 x) {
    float4 P0 = func_18(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_18(float4 x) {
    float4 P0 = func_16(x);
    float4 P1 = func_17(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_17(float4 x) {
    return (float4)(-0.538883, -0.705680, -0.549674, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_16(float4 x) {
    return x;
}
```
// Auto-generated function for node type: ConstantNode
inline float func_15(float4 x) {
    return (float)(0.185724);
}

// Auto-generated function for node type: SFSNode
inline float func_14(float4 x) {
    float P0 = func_6(x);
    float P1 = func_13(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_13(float4 x) {
    float P0 = func_12(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSNode
inline float func_12(float4 x) {
    float P0 = func_7(x);
    float P1 = func_11(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_11(float4 x) {
    float4 P0 = func_10(x);
    return length(P0);
}

// Auto-generated function for node type: VFSNode
inline float4 func_10(float4 x) {
    float4 P0 = func_8(x);
    float4 P1 = func_9(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_9(float4 x) {
    return (float4)(0.370516, 0.368164, 0.698672, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_8(float4 x) {
// Auto-generated function for node type: ConstantNode
inline float func_7(float4 x) {
    return (float)(0.259008);
}

// Auto-generated function for node type: ScalarMaskField
inline float func_6(float4 x) {
    float P0 = func_5(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_5(float4 x) {
    float P0 = func_0(x);
    float P1 = func_4(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_4(float4 x) {
    float4 P0 = func_3(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_3(float4 x) {
    float4 P0 = func_1(x);
    float4 P1 = func_2(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_2(float4 x) {
    return (float4)(0.000000, 0.000000, 0.000000, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_1(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_0(float4 x) {
    return x;
}


```c

return (float)(1.000000);
}

inline float evalDensity(float4 x) {
    return func_86(x);
}

bool hit_box(
    float4 ray_start,
    float4 ray_dir,
    float4 bmin,
    float4 bmax,
    float *tnear,
    float *tfar
) {
    float4 invr = (float4)(1.0f, 1.0f, 1.0f, 1.0f) / ray_dir;
    float4 ttop = invr * (bmax - ray_start);
    float4 tbot = invr * (bmin - ray_start);
    float4 tmin = min(ttop, tbot);
    float4 tmax = max(ttop, tbot);
    *tnear = max(max(tmin.x, tmin.y), tmin.z);
    *tfar = min(min(tmax.x, tmax.y), tmax.z);
    return *tfar > *tnear;
}

__kernel void render
{
    __global float* rgb,
    int time

    int WIDTH = get_global_size(0), HEIGHT = get_global_size(1);
    int4 dim = {get_global_size(0), get_global_size(1), 0, 0};
    int i = get_global_id(0); int j = get_global_id(1);

    if(i==WIDTH) return;
    if(j==HEIGHT) return;
    int idx = (WIDTH-1-j)*WIDTH+i;
    float4 col = (float4)(0,0,0,0);

    float camx = cos(time * .1f)*1.5f;
    float camz = sin(time * .1f)*1.5f;
    float4 cam = (float4)(camx,.7f,camz,0);
```
float4 look = (float4)(0,0,0,0);
float4 dir = normalize(look-cam);

// Compute camera coordinate frame and ray direction for this pixel.
float4 right = cross(dir,(float4)(0,1,0,0));
float4 up = cross(right, dir);
float u = -1.f + 2.f*(float)I/(float)WIDTH;
u *= (float)WIDTH / (float)HEIGHT;
float v = 1.f - 2.f*(float)j/(float)HEIGHT;
float screen_distance = (float)WIDTH/(float)HEIGHT / (2.f*sin(75.f/2.f * 3.141592659/180.f));
float4 D = screen_distance*dir + up*v + right*u;
D = normalize(D);

// Some constants for the volume rendering
const float ds = .01f;
const float kappa = 1.f;
const float _kappa = -kappa * ds;
const float _kappa_inv = 1.f / kappa;
float T = 1.f;

// Quick BBox Check
float tnear = 0;
float tfar = 6;
float4 bmin = (float4)(-1,-1,-1,0);
float4 bmax = (float4)(1,1,1,0);
if(hit_box(cam,D,bmin, bmax,&tnear, &tfar))
  for(float s=tnear; s<tfar; s+=ds) {
    float4 x = cam + D*s;
    float rho = evalDensity(x);
    if(rho<=0) continue;

  // Color from the velocity field
    float4 color = (float4)(1,1,1,1);
    const float q = 1.f;
    const float dT = exp(rho*_kappa);
    T *= dT;
    col = col + (1.f-dT)*_kappa_inv * color * q;
  }

  //col *= kappa;

float alpha = 1.f-T;
col = alpha;
col = min(max(col,(float4)(0,0,0,0)),(float4)(1,1,1,1));
rgb[3*idx+0] = col.x;
rgb[3*idx+1] = col.y;
rgb[3*idx+2] = col.z;
Appendix C  Automatically-Generated OpenCL Code

```c
inline float func_86(float4 x);
inline float func_85(float4 x);
inline float func_84(float4 x);
inline float func_83(float4 x);
inline float4 func_82(float4 x);
inline float4 func_81(float4 x);
inline float4 func_80(float4 x);
inline float func_79(float4 x);
inline float func_78(float4 x);
inline float func_77(float4 x);
inline float func_76(float4 x);
inline float func_75(float4 x);
inline float4 func_74(float4 x);
inline float4 func_73(float4 x);
inline float4 func_72(float4 x);
inline float func_71(float4 x);
inline float func_70(float4 x);
inline float func_69(float4 x);
inline float func_68(float4 x);
inline float func_67(float4 x);
inline float4 func_66(float4 x);
inline float4 func_65(float4 x);
inline float4 func_64(float4 x);
inline float func_63(float4 x);
inline float func_62(float4 x);
inline float func_61(float4 x);
inline float func_60(float4 x);
inline float func_59(float4 x);
inline float4 func_58(float4 x);
inline float4 func_57(float4 x);
inline float4 func_56(float4 x);
inline float func_55(float4 x);
inline float func_54(float4 x);
inline float func_53(float4 x);
inline float func_52(float4 x);
inline float func_51(float4 x);
inline float4 func_50(float4 x);
inline float4 func_49(float4 x);
inline float4 func_48(float4 x);
inline float func_47(float4 x);
inline float func_46(float4 x);
inline float func_45(float4 x);
inline float func_44(float4 x);
inline float func_43(float4 x);
```

54
inline float4 func_42(float4 x);
inline float4 func_41(float4 x);
inline float4 func_40(float4 x);
inline float func_39(float4 x);
inline float func_38(float4 x);
inline float func_37(float4 x);
inline float func_36(float4 x);
inline float func_35(float4 x);
inline float4 func_34(float4 x);
inline float4 func_33(float4 x);
inline float4 func_32(float4 x);
inline float func_31(float4 x);
inline float func_30(float4 x);
inline float func_29(float4 x);
inline float func_28(float4 x);
inline float func_27(float4 x);
inline float4 func_26(float4 x);
inline float4 func_25(float4 x);
inline float4 func_24(float4 x);
inline float func_23(float4 x);
inline float func_22(float4 x);
inline float func_21(float4 x);
inline float func_20(float4 x);
inline float func_19(float4 x);
inline float4 func_18(float4 x);
inline float4 func_17(float4 x);
inline float4 func_16(float4 x);
inline float func_15(float4 x);
inline float func_14(float4 x);
inline float func_13(float4 x);
inline float func_12(float4 x);
inline float func_11(float4 x);
inline float4 func_10(float4 x);
inline float4 func_9(float4 x);
inline float4 func_8(float4 x);
inline float func_7(float4 x);
inline float func_6(float4 x);
inline float func_5(float4 x);
inline float func_4(float4 x);
inline float4 func_3(float4 x);
inline float4 func_2(float4 x);
inline float4 func_1(float4 x);
inline float func_0(float4 x);

// Auto-generated function for node type: SFSUBNODE
inline float func_86(float4 x) {

float P0 = func_78(x);
float P1 = func_85(x);
return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_85(float4 x) {
    float P0 = func_84(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubeNode
inline float func_84(float4 x) {
    float P0 = func_79(x);
    float P1 = func_83(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_83(float4 x) {
    float4 P0 = func_82(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubeNode
inline float4 func_82(float4 x) {
    float4 P0 = func_80(x);
    float4 P1 = func_81(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_81(float4 x) {
    return (float4)(0.596517, 0.971362, 0.381845, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_80(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_79(float4 x) {
    return (float)(0.819644);
}
// Auto-generated function for node type: SFSNode
inline float func_78(float4 x) {
    float P0 = func_70(x);
    float P1 = func_71(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_77(float4 x) {
    float P0 = func_76(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: VFSNode
inline float func_76(float4 x) {
    float P0 = func_71(x);
    float P1 = func_75(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_75(float4 x) {
    float4 P0 = func_74(x);
    return length(P0);
}

// Auto-generated function for node type: ConstantNode
inline float4 func_74(float4 x) {
    float4 P0 = func_72(x);
    float4 P1 = func_73(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float func_73(float4 x) {
    return (float)(-0.140355, -0.372477, -0.765280, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_72(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_71(float4 x) {
    return (float)(0.364203);


// Auto-generated function for node type: SFSNode
inline float func_70(float4 x) {
    float P0 = func_62(x);
    float P1 = func_69(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_69(float4 x) {
    float P0 = func_68(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSNode
inline float func_68(float4 x) {
    float P0 = func_63(x);
    float P1 = func_67(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_67(float4 x) {
    float4 P0 = func_66(x);
    return length(P0);
}

// Auto-generated function for node type: VFSNode
inline float4 func_66(float4 x) {
    float4 P0 = func_64(x);
    float4 P1 = func_65(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_65(float4 x) {
    return (float4)(0.510008, 0.128277, -0.140845, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_64(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_63(float4 x) {
    return (float)(0.486786);
}

// Auto-generated function for node type: SFSubNode
inline float func_62(float4 x) {
    float P0 = func_54(x);
    float P1 = func_61(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_61(float4 x) {
    float P0 = func_60(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_60(float4 x) {
    float P0 = func_55(x);
    float P1 = func_59(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_59(float4 x) {
    float4 P0 = func_58(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_58(float4 x) {
    float4 P0 = func_56(x);
    float4 P1 = func_57(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_57(float4 x) {
    return (float4)(0.587541, -0.486584, 0.699878, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_56(float4 x) {
    return x;
}
// Auto-generated function for node type: ConstantNode
inline float func_55(float4 x) {
    return (float)(0.949323);
}

// Auto-generated function for node type: SFSUBNode
inline float func_54(float4 x) {
    float P0 = func_46(x);
    float P1 = func_53(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_53(float4 x) {
    float P0 = func_52(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSUBNode
inline float func_52(float4 x) {
    float P0 = func_47(x);
    float P1 = func_51(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_51(float4 x) {
    float4 P0 = func_50(x);
    return length(P0);
}

// Auto-generated function for node type: VFSUBNode
inline float4 func_50(float4 x) {
    float4 P0 = func_48(x);
    float4 P1 = func_49(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_49(float4 x) {
    return (float4)(0.612094, 0.652731, 0.489500, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_48(float4 x) {
return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_47(float4 x) {
    return (float)(0.809740);
}

// Auto-generated function for node type: SFSUBNode
inline float func_46(float4 x) {
    float P0 = func_38(x);
    float P1 = func_45(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_45(float4 x) {
    float P0 = func_44(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSUBNode
inline float func_44(float4 x) {
    float P0 = func_39(x);
    float P1 = func_43(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_43(float4 x) {
    float4 PO = func_42(x);
    return length(PO);
}

// Auto-generated function for node type: VFSUBNode
inline float4 func_42(float4 x) {
    float4 P0 = func_40(x);
    float4 P1 = func_41(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_41(float4 x) {
    return (float4)(-0.448326, -0.706579, 0.742569, 0);
}
// Auto-generated function for node type: IdentityNode
inline float func_40(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_39(float4 x) {
    return (float)(0.014432);
}

// Auto-generated function for node type: SFSUBnode
inline float func_38(float4 x) {
    float P0 = func_30(x);
    float P1 = func_37(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_37(float4 x) {
    float P0 = func_36(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSUBnode
inline float func_36(float4 x) {
    float P0 = func_31(x);
    float P1 = func_35(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_35(float4 x) {
    float4 P0 = func_34(x);
    return length(P0);
}

// Auto-generated function for node type: VFSUBnode
inline float4 func_34(float4 x) {
    float4 P0 = func_32(x);
    float4 P1 = func_33(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float func_33(float4 x) {
    return (float4)(-0.136839, 0.675313, 0.216804, 0);
// Auto-generated function for node type: IdentityNode
inline float4 func_32(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_31(float4 x) {
    return (float)(0.294657);
}

// Auto-generated function for node type: SFSubNode
inline float func_30(float4 x) {
    float P0 = func_22(x);
    float P1 = func_29(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_29(float4 x) {
    float P0 = func_28(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_28(float4 x) {
    float P0 = func_23(x);
    float P1 = func_27(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float4 func_27(float4 x) {
    float4 P0 = func_26(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_26(float4 x) {
    float4 P0 = func_24(x);
    float4 P1 = func_25(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_25(float4 x) {
    return (float4)(-0.739574, 0.062630, -0.572185, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_24(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_23(float4 x) {
    return (float)(0.734024);
}

// Auto-generated function for node type: SFSubNode
inline float func_22(float4 x) {
    float P0 = func_14(x);
    float P1 = func_21(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_21(float4 x) {
    float P0 = func_20(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_20(float4 x) {
    float P0 = func_15(x);
    float P1 = func_19(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float4 func_19(float4 x) {
    float4 P0 = func_18(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_18(float4 x) {
    float4 P0 = func_16(x);
    float4 P1 = func_17(x);
    return P0 - P1;
}
// Auto-generated function for node type: ConstantNode
inline float func_17(float4 x) {
    return (float4)(-0.538883, -0.705680, -0.549674, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_16(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_15(float4 x) {
    return (float)(0.185724);
}

// Auto-generated function for node type: SFSubNode
inline float func_14(float4 x) {
    float P0 = func_6(x);
    float P1 = func_13(x);
    return P0 - P1;
}

// Auto-generated function for node type: ScalarMaskField
inline float func_13(float4 x) {
    float P0 = func_12(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_12(float4 x) {
    float P0 = func_7(x);
    float P1 = func_11(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_11(float4 x) {
    float4 P0 = func_10(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_10(float4 x) {
    float4 P0 = func_8(x);
    float4 P1 = func_9(x);
```c
return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_9(float4 x) {
    return (float4)(0.370516, 0.368164, 0.698672, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_8(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_7(float4 x) {
    return (float)(0.259008);
}

// Auto-generated function for node type: ScalarMaskField
inline float func_6(float4 x) {
    float P0 = func_5(x);
    return P0 > 0 ? 1.0 : 0.0;
}

// Auto-generated function for node type: SFSubNode
inline float func_5(float4 x) {
    float P0 = func_0(x);
    float P1 = func_4(x);
    return P0 - P1;
}

// Auto-generated function for node type: LengthNode
inline float func_4(float4 x) {
    float4 P0 = func_3(x);
    return length(P0);
}

// Auto-generated function for node type: VFSubNode
inline float4 func_3(float4 x) {
    float4 P0 = func_1(x);
    float4 P1 = func_2(x);
    return P0 - P1;
}

// Auto-generated function for node type: ConstantNode
inline float4 func_2(float4 x) {
```
return (float4)(0.000000, 0.000000, 0.000000, 0);
}

// Auto-generated function for node type: IdentityNode
inline float4 func_1(float4 x) {
    return x;
}

// Auto-generated function for node type: ConstantNode
inline float func_0(float4 x) {
    return (float)(1.000000);
}

inline float evalDensity(float4 x) {
    return func_86(x);
}

bool hit_box(
    float4 ray_start,
    float4 ray_dir,
    float4 bmin,
    float4 bmax,
    float *tnear,
    float *tfar
) {
    float4 invr = (float4)(1.f,1.f,1.f,1.f) / ray_dir;
    float4 ttop = invr*(bmax-ray_start);
    float4 tbot = invr*(bmin-ray_start);
    float4 tmin = min(ttop,tbot);
    float4 tmax = max(ttop,tbot);
    *tnear = max(max(tmin.x,tmin.y),tmin.z);
    *tfar = min(min(tmax.x,tmax.y), tmax.z);
    return *tfar > *tnear;
}

__kernel void render
(
    __global float* rgb,
    int time
)
{
    int WIDTH = get_global_size(0), HEIGHT=get_global_size(1);
    int4 dim = (get_global_size(0), get_global_size(1), 0, 0);
int i = get_global_id(0); int j = get_global_id(1);

if(i>=WIDTH) return;
if(j>=HEIGHT) return;
int idx = (WIDTH-1-j)*WIDTH+i;
float4 col = (float4)(0,0,0,0);

float camx = cos( time * .1f)*1.5f;
float camz = sin( time * .1f)*1.5f;
float4 cam = (float4)(camx,.7f,camz,0);

float4 look = (float4)(0,0,0,0);
float4 dir = normalize(look-cam);

// Compute camera coordinate frame and ray direction for this pixel.
float4 right = cross(dir,(float4)(0,1,0,0));
float4 up = cross(right, dir);
float u = -1.f + 2.f*(float)i/(float)WIDTH;

// Some constants for the volume rendering
const float ds = .01f;
const float kappa = 1.f;
const float _kappa = -kappa * ds;
const float _kappa_inv = 1.f / kappa;
float T = 1.f;

// Quick BBox Check
float tnear = 0;
float tfar = 6;
float4 bmin = (float4)(-1,-1,-1,0);
float4 bmax = (float4)(1,1,1,0);

if(hit_box(cam,D,bmin, bmax, &tnear, &tfar))
  for(float s=tnear; s<tfar; s+=ds) {
    float4 x = cam + D*s;

    float rho = evalDensity(x);
    if(rho<=0) continue;

// Color from the velocity field
    float4 color = (float4)(1,1,1,1);
const float q = 1.f;
    const float dT = exp(rho*kappa);
    T *= dT;
    col = col + (1.f-dT)*T*kappa_inv * color * q;
}

//col *= kappa:

float alpha = 1.f-T;
    col *= alpha;
    col = min(max(col, (float4)(0,0,0,0)), (float4)(1,1,1,1));
    rgb[3*idx+0] = col.x;
    rgb[3*idx+1] = col.y;
    rgb[3*idx+2] = col.z;
}
Bibliography


