Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.
Multiresolution Liquid Simulation and Rendering

by

Richard Lee, B.A.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Doctor of Philosophy

University of Dublin, Trinity College

October 2007
Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Richard Lee

March 25, 2008
Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Richard Lee

March 25, 2008
Acknowledgments

First and foremost, my thanks go to Carol O’Sullivan for all her help, advice, encouragement and patience. I am especially grateful to Carol for giving me the freedom to explore the research topics I find most interesting. Secondly, my gracious thanks go to Steve Collins for helping to organize my internship at Intel’s Applications Research Lab in Santa Clara, California. At Intel, thanks to everybody who provided help, especially Mikhail Smelyanskiy and Jim Hurley. Finally, thanks to all GV2 members, past and present.

Richard Lee

University of Dublin, Trinity College
October 2007
Multiresolution Liquid Simulation and Rendering

Publication No. ____________________

Richard Lee, Ph.D.
University of Dublin, Trinity College, 2007

Supervisor: Carol O’Sullivan
Multiresolution Liquid Simulation and Rendering
Richard Lee
Supervised by Carol O'Sullivan

Abstract Current approaches to real-time liquid simulation are based on solutions to the two-dimensional wave equation. In this thesis, we introduce a new, explicit solution method for the shallow water equations that achieves more realistic results. We obtain real-time performance by collocating variables to increase advection efficiency and we treat the fluid as weakly compressible to avoid solving a Poisson equation for pressure. Our solver is easily adapted to work on multiresolution grids with view dependent refinement criteria and we can handle accurate interaction with static or dynamic obstacles, including thin shells, immersed in the flow.

The results are rendered in real-time using fast approximations for light scattering within liquid volumes. If higher quality results are desired, a fast offline rendering approach based on Monte Carlo path tracing can also be used. This is accelerated by sampling from sets of precomputed probability distributions to rapidly evaluate light scattering paths within the homogeneous liquid medium. Finally, we introduce a framework for efficient rendering of small scale liquid particles that computes volumetric illumination by treating the particles as an inhomogeneous participating medium sampled on a grid.
Related Publications


Contents

Acknowledgments iv

Abstract v

List of Tables xii

List of Figures xiii

Chapter 1 Introduction 1
1.1 Liquid Simulation .................................................. 1
1.2 Liquid Rendering .................................................. 2
1.3 Summary of Chapters ............................................. 4

Chapter 2 Related Work 6
2.1 Liquid Simulation .................................................. 6
  2.1.1 Heightfield Methods ......................................... 7
  2.1.2 Grid-based Methods ......................................... 11
  2.1.3 Particle-based Methods ...................................... 15
  2.1.4 Spray and Foam .............................................. 17
2.2 Liquid Rendering .................................................. 18
  2.2.1 Liquid Representations ...................................... 19
  2.2.2 Light Scattering .............................................. 22
  2.2.3 Spray and Foam .............................................. 25

Chapter 3 Fluid Simulation Concepts 28
3.1 Navier-Stokes Equations ....................................... 29
3.2 Marker-and-Cell Method ................................................................. 30
  3.2.1 Incompressible Euler Equations .................................................... 30
  3.2.2 Projection Method ........................................................................ 32
  3.2.3 MAC Grid Discretization ............................................................... 33
  3.2.4 Boundary Conditions .................................................................... 35
  3.2.5 Semi-Lagrangian Advection ............................................................ 36

3.3 Smoothed Particle Hydrodynamics ......................................................... 38
  3.3.1 Overview ...................................................................................... 38
  3.3.2 Smoothing Kernels ....................................................................... 40
  3.3.3 Artificial Compressibility .............................................................. 41
  3.3.4 Lagrangian Discretization .............................................................. 42
  3.3.5 Simulation Algorithm .................................................................... 43
  3.3.6 Optimizations ................................................................................ 44

Chapter 4 Weakly Compressible, Heightfield Liquid Simulation ............... 46
  4.1 Introduction ...................................................................................... 46
  4.2 Governing Equations ....................................................................... 47
  4.3 Solver Overview ................................................................................ 49
  4.4 Grid Configuration ............................................................................ 51
  4.5 Advection Step .................................................................................. 52
  4.6 Pressure Step .................................................................................... 55
  4.7 Discussion ......................................................................................... 56
  4.8 Performance .................................................................................... 59
  4.9 Adaptive Simulation ......................................................................... 61
    4.9.1 Multiresolution Grids ................................................................. 62
    4.9.2 Pressure Gradient Discretization ............................................... 64
    4.9.3 View Dependent Dynamic Adaptivity ....................................... 67
  4.10 Thin Boundaries ............................................................................. 68
  4.11 Visual Enhancements ...................................................................... 70
    4.11.1 Choppy Wave Modification .................................................... 70
    4.11.2 Procedural High Frequency Waves ....................................... 71
    4.11.3 Spray and Foam ..................................................................... 72
Chapter 5  Liquid Optics

5.1 Introduction ................................................................. 74
5.2 Radiometry of Liquids ...................................................... 75
  5.2.1 Surface Scattering .................................................... 75
  5.2.2 Volume Scattering .................................................... 77
5.3 Ray Tracing Liquids ......................................................... 79
  5.3.1 Whitted Ray Tracing ................................................ 79
  5.3.2 Absorption .............................................................. 80
  5.3.3 Scattering ............................................................... 81
5.4 Path Tracing Liquids ......................................................... 84
  5.4.1 Monte Carlo Rendering ........................................... 84
  5.4.2 Probabilistic Intersections ....................................... 85
5.5 Accelerated Light Propagation ........................................... 86
  5.5.1 Motivation .............................................................. 86
  5.5.2 Overview .............................................................. 87
  5.5.3 Precomputation ...................................................... 88
  5.5.4 Sampling .............................................................. 91
  5.5.5 Rendering ............................................................ 92
  5.5.6 Results and Discussion .......................................... 94
5.6 GPU Rendering ............................................................. 97
  5.6.1 Exploiting Graphics Hardware .................................. 97
  5.6.2 Multisampling ....................................................... 98
  5.6.3 Liquid Shaders ...................................................... 100
5.7 Rendering Whitewater ................................................... 105
  5.7.1 Overview ............................................................ 105
  5.7.2 Particle Rendering ................................................ 106
  5.7.3 Particle Shading .................................................... 109
  5.7.4 Specular Effects ................................................... 120

Chapter 6  Conclusions and Future Work ................................ 129

6.1 Summary of Contributions ............................................. 129
6.2 Future Work ................................................................ 131
# List of Tables

4.1 Performance comparison of different solvers for different grid resolutions.  
   All values are in updates per second. ................................................................. 59

4.2 Memory requirements of different solvers for different grid resolutions.  
   All values are in kilobytes. ..................................................................................... 59

5.1 Effect of grid resolution on performance. All times are in seconds. ... 119

5.2 Effect of particle count on performance when using a 200x400x200 grid.  
   All times are in seconds. ..................................................................................... 119
# List of Figures

2.1 In Eulerian simulators (left), the fluid attributes such as velocities are sampled at fixed grid points. In Lagrangian simulators (right), they are sampled at mobile points that move with the fluid. ................................................ 7

2.2 Procedural ocean waves on a heightfield, based on our implementation of the Fourier synthesis technique described in [Tes04]. ................................................................. 9

2.3 Vorticity visualization of a grid-based simulation of swirling fluid. The image on the left was computed using semi-Lagrangian advection with bilinear interpolation, while the right image used the monotonic bicubic interpolation scheme of [FSJ01]. The higher order interpolant uses additional data points to perform a sharper reconstruction of the advected velocity field which results in greater detail. .................................................. 11

2.4 Reproduction of the Zalesak sphere test from [EFFM02]. Even with a 5th order accurate advection scheme on a 100³ grid, the object on the left still loses volume due to numerical dissipation after a single revolution. ................................................................. 13

2.5 The left image shows our implementation of a level set polygonized using marching cubes [LC87] and rendered in OpenGL. The right image shows the same level set raytraced using sphere tracing [HSK89]. ................................................................. 20

2.6 The discrete signed distance function given by the union of a group of spherical particles is smoothed and raytraced using sphere tracing. The left image shows our implementation result using trilinear interpolation to reconstruct the continuous function and the right image shows the result using the tricubic Catmull-Rom interpolant of [ML94]. ................................................................. 21

3.1 Frames from our animation of falling drops of liquid filling a container. This was simulated on a 700x200 MAC grid. ................................................................. 31
3.2 The left image shows the locations of horizontal velocity samples, vertical velocity samples, and pressure samples. The right image visualizes the same velocity field in a more natural way.

3.3 A field $q$ is sampled at the centre points of grid cells. To compute $q^{(n+1)}(\vec{x})$, we trace backwards through the velocity field and interpolate $q^{(n)}$ at the point at which we end up, $\vec{x} - \Delta t \vec{u}(\vec{x})$.

3.4 Frames from our animation of an oscillating wall inducing breaking waves. This was simulated using 100,000 smoothed particles.

4.1 Shallow water simulation on a $100 \times 100$ grid. On the left, the user interactively applies forces to the fluid to form a whirlpool. The corresponding density and velocity fields are shown on the right.

4.2 Location of horizontal velocities (green), vertical velocities (red) and densities (blue) for a staggered grid configuration (left), and a collocated configuration (right).

4.3 Using central differences on collocated grids can lead to checkerboard instabilities, examples of which can be seen in the density field on the left, and the heightfield on the right.

4.4 All cell variables are advected together by forward-tracing $N \times N$ packets, where $N = 2$ in this example (left). After advection, the quantities carried by the packets are accumulated on a new grid (right).

4.5 Variation in wave detail on grids of different resolution (from left to right: $25 \times 25$, $50 \times 50$, $100 \times 100$, $200 \times 200$, $400 \times 400$). An identical forcing function was applied to the fluid in all cases.

4.6 Interactive, quadtree solver using vorticity-based adaptation to focus computational effort in regions of high detail.

4.7 Values required to evaluate the horizontal pressure gradients at $p_C$ and $p_D$. The blue points are pressure samples and the orange points are obtained by linear interpolation.

4.8 1D adaptive shallow water simulation. Notice that the height changes smoothly over the boundary between cells of different size.
4.9 Adaptive version of our 2D solver showing tiled uniform grids of different resolutions (chosen arbitrarily in this example). The green tiles are $8 \times 8$ cells, the red tiles are $16 \times 16$ cells, the blue tiles are $32 \times 32$ cells and the yellow tiles are $64 \times 64$ cells.

4.10 To support thin boundaries, extra density variables are placed in each cell that intersects the boundary.

4.11 Waves bouncing off of an infinitely thin boundary. Even though the two wave fronts seen in the first frame enter the same cells (those intersected by the red wall), they do not interact with one another in any way.

4.12 From left to right: normal in-plane grid vertices; in-plane grid vertices deformed by pressure gradient; corresponding pressure field; 3D height-field with (top) and without (bottom) choppy wave modification.

4.13 The left frame shows the wake produced by a scripted ball moving through water. The right frame shows the exact same scene with procedural wave displacements added.

4.14 An example foam texture generated using procedural wave synthesis (left), the result of converting a collection of foam particles into a foam density map (middle), and the final texture obtained after masking the first texture with the second (right).

5.1 Various ways in which photons can interact with the surface and volume of a liquid.

5.2 Whitted's recursive ray tracing algorithm accurately accounts for perfect specular reflection and refraction which makes it appropriate for rendering liquids.

5.3 White light turns blue when passing through an absorbing medium (left). For scattering media, additional light bounces into the path based on the angle it makes with the light source (right).

5.4 Ray-traced liquid featuring absorption.

5.5 Correct evaluation of single scattering requires ray marching to account for attenuation of in-scattered light.

5.6 Ray-traced liquid featuring absorption, out-scattering and a fast in-scattering approximation that avoids ray marching.
5.7 Even for highly scattering media such as molten wax, our in-scattering approximation gives plausible results.

5.8 Examples of participating media rendered using Monte Carlo path tracing (left, centre). For comparison, the image on the right was computed by ignoring subsurface scattering and only simulating absorption events.

5.9 On the left, a photon enters a participating medium and undergoes a series of scatters before exiting; on the right, when the photon reaches a certain distance within the medium, it is "teleported" between spheres of different radii until it is back within reach of the surface. Less computation is required to compute the path of the photon on the right.

5.10 The PDFs are defined in a coordinate system where the photon is located at the origin, directed upwards (left). A fraction of the photons that exit the sphere when precomputing a PDF are shown, along with their outgoing directions, which are visualized by mapping x, y and z coordinates to red, green and blue colour components respectively (right).

5.11 The left image was rendered in 484 minutes using pure Monte Carlo path tracing. The right image was rendered in 257 minutes using accelerated photon propagation. There is no visible difference between the images indicating that our acceleration technique does not introduce perceptible error into the calculation.

5.12 The image on the left shows a participating medium rendered in 135 minutes using pure Monte Carlo path tracing. The image on the right shows the same scene rendered in 71 minutes using accelerated photon propagation. Again, there is no perceptible difference between the images.

5.13 Single frame from an animation of spinning cubes rendered using (a) no effects, (b) anti-aliasing only (64 samples per pixel), (c) depth of field only (64 spp), (d) box-filtered motion blur only (64 spp), (e) tent-filtered motion blur only (64 spp), and (f) all effects (1024 spp).

5.14 Values required to compute the distance $s$ that the refracted ray travels between the water surface and the ground.
5.15 Comparison of in-scattered light when only accounting for attenuation between the eye and refracted ray (left), and when accounting for attenuation between both eye and refracted ray, and light source and refracted ray (right).

5.16 Values required to determine how in-scattered light is attenuated between the refracted ray and the light source.

5.17 Two frames from an animation of a ball bobbing in an ocean demonstrating depth of field and local raytraced refractions (left) and reflections (right).

5.18 To render a particle as a disk it is necessary to both orient it so that it is pointing towards the eye position, and scale it to account for its position relative to the eye.

5.19 From left to right: slice-based volume rendering; particle rendering with self-shadowing; particle rendering without self-shadowing; particle rendering with scattering parameters chosen to resemble liquid foam.

5.20 The optical depth at each grid point is calculated in a single sweep through the grid. To account for an arbitrary lighting direction, densities need to be accumulated in four different directions. The first four images show the lighting results when each of these directions is considered by itself and the last image shows the result when all directions are considered together.

5.21 Since geometry and lighting are decoupled, grid resolution does not have a large effect on lighting quality, although shadows become increasingly blurry. The resolutions from left to right are: 20x40x20, 50x100x50, 100x200x100 and 200x400x200.

5.22 Attenuation through an inhomogeneous liquid medium is equivalently computed by considering attenuation through its constituent, homogeneous parts.

5.23 Fake multiple scattering can be achieved by “blurring” the lighting information stored at each grid cell. From left to right, the number of smoothing passes using a 3x3x3 box filter is 0, 1, 5 and 20.
5.24 Frames from an animation of a particulate Buddha being entrained into a simulated tornado. Achieving a similar level of detail with purely volume-based or mesh-based approaches would be very difficult.

5.25 The photograph on the left shows a primary and secondary rainbow. Notice that the brightness of the sky also changes as a function of the scattering angle (source: http://www.atoptics.co.uk/rainbows). The second photograph shows a broken rainbow formed out of ocean mist (source: http://en.wikipedia.org/wiki/Rainbow).

5.26 The amount of light that exits a water droplet varies depending on the scattering angles. This data was computed by simulating 9 million photons for each wavelength and recording the results in a table of 1,800 bins.

5.27 Photon paths which give rise to (a) specular highlights (no refraction), (b) glowing (no internal reflection), (c) the primary rainbow (one internal reflection), and (d) the secondary rainbow (two internal reflections).

5.28 Simulated rainbow computed by scattering light through 50 million water droplets.

5.29 Frames from an animation of a particulate Buddha dispersing in the wind demonstrating broken primary and secondary rainbows and motion-blurred specular highlights.
Chapter 1

Introduction

1.1 Liquid Simulation

The field of computational fluid dynamics (CFD) has been investigating ways to solve the Navier-Stokes equations for over 50 years. These complex equations describe how fluids move over time and their numerical solution is of practical interest to just about every scientific and engineering discipline. It wasn't until the mid '90s that computer graphics researchers began adapting simulation techniques from CFD and applying them to the problem of animating fully three-dimensional liquids. Prior to that time, water animation was generally limited to the simple case of water wave propagation and was either achieved by purely procedural means or by solving the two-dimensional wave equation.

In the last decade, the field of fluid animation has seen much progress. Literally hundreds of numerical methods have been developed by the CFD community to solve the Navier-Stokes equations and most of these are tailored for specialized applications. This makes it very difficult to find numerical methods which are versatile and robust enough to cope with the limitless variety of scenarios one may wish to animate. Of course, animators and engineers have vastly different goals in terms of accuracy and efficiency and it makes sense therefore to instead tailor numerical methods to the specific requirements of animators by, for example, trading off accuracy for visual fidelity, or stability for speed.

Motivated by this reasoning, two approaches have emerged from CFD as standard
tools for simulating fluid flows in computer graphics: grid-based methods where fluid properties are sampled at fixed grid points, and particle-based methods where fluid properties are sampled at the location of mobile points. However, while both of these techniques have been very active areas of research in recent years, there has been next to no progress made in the development of techniques for real-time fluid simulation. In fact, real-time techniques today are basically the same as they were in the '80s. Since computational power has not yet reached the point where computing solutions to the 3D Navier-Stokes equations is practical, modern interactive applications still resort to simulating liquids by solving the 2D wave equation and using the results to perturb the heights of vertices distributed over a plane.

In this thesis, we will consider solution techniques for a more complex set of equations called the shallow water equations which have mostly been ignored by the graphics community. These equations describe a better approximation to the Navier-Stokes equations than the 2D wave equation and therefore their solution results in more realistic liquid motion. Although 3D effects such as splashing or breaking waves cannot be modelled using these equations, they do bring us one step closer to developing a real-time solver for the full Navier-Stokes equations.

With regard to improved interactive simulation of liquids, the main contributions of this thesis are as follows:

• We introduce a novel solution method for the shallow water equations that is easy to understand and implement, and performs competitively with solvers based on the 2D wave equation.

• We present an adaptive version of our solver that uses view dependent refinement criteria to focus computational effort where it is most important.

• We present a new technique for handling interaction between fluids and infinitely thin objects.

1.2 Liquid Rendering

While the state of the art in real-time liquid simulation is no match for the results currently possible with offline animation techniques, approaches for rendering liquid
surfaces in real-time can often achieve results comparable in quality to offline renderers. The optical properties of liquids are well understood and computing important effects such as reflection, refraction or attenuation is relatively easy on modern graphics hardware. Not surprisingly, most of the work on rendering liquids in real-time has been based on a heightfield representation of the liquid surface. This configuration greatly simplifies the computation of ray paths through the liquid and provides opportunities to introduce simplifying assumptions without adversely affecting quality. However, one effect that has received little attention is the scattering of light within the liquid volume. This is particularly important for rendering large bodies of water, such as oceans or lakes, where organic matter and other impurities suspended in the volume cause significant amounts of scattering.

An accurate model of light scattering within participating media such as liquids requires solving the volume rendering equation and is one of the most difficult problems in rendering. As such, real-time techniques for liquid rendering generally ignore this problem and assume that no scattering occurs. In this thesis, we consider new techniques to accurately compute multiple scattering of light in homogeneous participating media, and also fast single-scattering approximations which are appropriate for real-time rendering. We also consider light scattering within variable density volumes. This will be used to compute a fast lighting approximation for splashing whitewater, which will be represented and simulated using an uncoupled particle system.

Our contributions in rendering liquids will be presented in the second part of this thesis and can be summarized as follows:

- We discuss using raytracing to realistically render liquids and derive new in-scattering approximations for the general case, and tailored specifically for height-fields.

- We introduce a new technique for accelerating path tracing of homogeneous participating media based on “teleporting” photons between points in the medium.

- We present a multisampling framework for rendering high quality animations using OpenGL.

- We present a fast, OpenGL-based particle rendering solution that uses coarse grids to rapidly compute volumetric illumination.
• We extend the particle shading model with effects that arise from specular reflection and transmission through water droplets. Our method can accurately simulate the formation of rainbows and related phenomena at negligible additional cost.

1.3 Summary of Chapters

The remainder of this thesis is divided into the following chapters:

Chapter 2 — Related Work: The first half of this chapter provides a review of state of the art simulation techniques that use heightfield, grid and particle representations of the fluid volume. It also discusses techniques that have been used to simulate ocean spray and foam effects. The second half of the chapter discusses both real-time and offline techniques that have been developed to render liquids. In particular, we discuss the various representations that are commonly used to represent liquid volumes and how these can facilitate efficient rendering, and we provide a brief survey of the important work on modelling light scattering within participating media. The chapter closes with a review of prior techniques that have been used to render particle-based spray and foam.

Chapter 3 — Fluid Simulation Concepts: In this chapter, we introduce the Navier-Stokes equations and describe the two most popular algorithms used to solve them — the marker-and-cell method and smoothed particle hydrodynamics. Many of the basic principles introduced here will later be applied to the problem of solving the shallow water equations.

Chapter 4 — Weakly Compressible, Heightfield Liquid Simulation: This chapter presents a fast and compact solver for the shallow water equations that runs at real-time speeds. We also discuss how to extend this method to multiresolution grids and how to achieve accurate interaction between the fluid and infinitely thin objects that are immersed in the flow. Finally, we present various approaches that can be used to further increase the visual quality of the simulation.

Chapter 5 — Liquid Optics: This chapter begins with a review of the fundamental principles governing the interaction between light and liquids. It then describes
techniques for rendering light scattering within liquids using raytracing and path-tracing. This is followed by a description of how OpenGL can be used to render high quality animations with motion blur and depth of field, real-time oceans with in-scattered light, and large particle systems. Finally, we discuss how to realistically light particles using a grid-based solution and how to model the phenomena that occurs when the particles are treated as spherical water droplets.

Chapter 6 — Conclusions and Future Work: In these closing remarks, the contributions of the thesis are summarized and suggested directions for future work are presented.
Chapter 2

Related Work

2.1 Liquid Simulation

The starting point for nearly all liquid animation techniques used in computer graphics is a set of time-dependent, partial differential equations called the Navier-Stokes equations. These equations describe how a fluid's velocity evolves over time due to self-advection, viscosity, pressure and gravity, while simultaneously being constrained to conserve mass at all times. Most methods for simulating fluids are based on a discretization of these equations, which involves selecting a finite number of points in space and time at which a numerical solution to the equations is computed. In reality, the equations are continuous, which means that they hold at all points in space and time. However, by choosing sample points that are appropriately spaced, the desired amount of detail can be captured while keeping computational expenditure to a minimum.

There are two common approaches used in computer graphics to simulate fluids: grid-based methods and particle-based methods (see Figure 2.1). The main difference between them is that grid-based methods use the Eulerian point of view where the locations of sample points are fixed, whereas particle-based methods use a Lagrangian formulation where the sample points are free to move about. Of course, the choice of discretization scheme is greatly complicated by the need to simultaneously satisfy the goals of stability, accuracy\(^1\) and efficiency, so all of the methods we discuss in this

\(^1\)For the purposes of animation, achieving physical accuracy is not usually a requirement, provided that visually plausible results are obtained.
chapter make various tradeoffs in satisfying these goals.

In addition to approaches that solve the full Navier-Stokes equations, there are also many methods that solve simplified versions of the equations such as Bernoulli’s equation or the shallow water equations. In fact, all of the liquid animation techniques cited in this chapter are physically-based to some degree of approximation. This means that all of the approaches can be arrived at by simplifying the Navier-Stokes equations under various assumptions to give either a reduced set of equations that can be solved numerically, or a set of closed-form, analytical expressions that provide specific solutions to the Navier-Stokes equations. Also note that, while this chapter deals primarily with techniques that are applicable to the simulation of liquids, solutions to the incompressible Navier-Stokes equations can be used to simulate a wide range of natural phenomena, including smoke [FSJ01, FM97], fire [NFJ02, HSF07], explosions [FOA03, YOH00, RNGF03], clouds [HBSL03], chemically reactive fluids [KJI07, LSSF06], viscoelastic fluids [GOB04, LSSF06], viscoplastic fluids [BWHT07], and sand [ZB05].

2.1.1 Heightfield Methods

Solving the 3D Navier-Stokes equations is very computationally expensive and is not yet practical for interactive applications. As a result, most real-time simulations of liquids are based on a 2D heightfield approximation. A heightfield is simply a uniform grid of
vertices defined on a horizontal plane. The vertical coordinates are free to change, but the in-plane coordinates remain fixed. This surface approximation is often suitable for animating large bodies of water such as an ocean or swimming pool. However, because the heightfield representation only defines a single height value for each point on the plane, it cannot be used to animate breaking waves or splashing water.

**Procedural Techniques**

Much of the earlier work on animating water used ad hoc, parametric functions to animate heightfields. Max [Max81] modelled the elevation of a water surface using a sum of sinusoids that represented waves of various amplitudes, wave lengths and frequencies. Fourier and Reeves [FR86] improved the realism of ocean wave motion by using the Gerstner model that accounts for the circular orbits of points on the water surface. They also modelled wave refraction and detected the onset of breaking waves due to the elevation of the ocean floor. Peachey [Pea86] blended between more complex wave profiles based on wave steepness and water depth to model a wider variety of wave shapes. Ts’o and Barsky [TB87] also modelled water as a superposition of sine waves but used ray-tracing techniques to compute how waves change direction due to refraction in shallow water. More recently, Hinsinger et al. [HNC02] modelled deep water using summed Gerstner waves to perturb an adaptive, view-dependent heightfield mesh. Similar approaches were used in [BC06] and [Bre07], with the addition of controls to allow artist-driven shaping of waves and the ability to define regions of influence for waves.

Tessendorf [Tes04] popularized an approach for ocean wave synthesis using Fast Fourier Transforms to greatly speed up the superposition of sinusoidal waves (see Figure 2.2). This approach makes it practical to compute the propagation of millions of waves on grids with resolutions as high as 2048x2048 and is routinely used for ocean visual effects in feature film production (see for example [LZD06]). Tessendorf samples wave properties from statistical distributions such as the Phillips spectrum and achieves realistic wave propagation by modelling the dispersive relationship between wave speed and wavelength. Furthermore, similar to the results achieved with Gerstner waves, Tessendorf presents a technique to displace vertices in the horizontal plane to make the water appear choppy instead of calm.
Lastly, Yuksel et al. [YHK07] recently introduced a new approach for procedural wave animation based on the superposition of “dents” and “bumps” defined by non-interacting particles. Collectively, these particles represent moving wavefronts. When wavefronts diverge, the particles are subdivided to maintain a smooth wave shape. While this approach is unconditionally stable and achieves real-time performance, it is not able to handle waves of different frequency or other effects such as wave refraction and diffraction.

Simulation Techniques

There are two very important advantages of procedural methods. Firstly, they provide arbitrarily high spatial and temporal resolution; and secondly, since waves are defined as continuous functions, no energy loss occurs over time (unless explicit damping is built into the wave model). It is generally not possible to achieve the same level of detail using numerical simulation techniques. For one thing, extremely high resolution grids would be required to capture the shape of high frequency waves. But more importantly, the numerical dissipation that results from trying to advect a continuous signal on a discrete grid causes wave motion to dampen artificially.

However, procedural techniques do suffer from one major disadvantage. They do not allow surface disturbances to affect the behavior of the water. For example, it is not possible for objects to interact with the water and generate ripples, or for waves to reflect off objects. For this reason, heightfield simulation techniques are often necessary to achieve realism, particularly in interactive applications. This requires the numerical solution of partial differential equations, which describe how the motion of the water surface evolves over time.
The 2D shallow water equations can easily be derived from the 3D Navier-Stokes equations by making a number of simplifying assumptions that are appropriate for the flow of thin layers of inviscid fluid [ESHD05]. These equations account for the horizontal flow of fluid induced by the hydrostatic pressure differences that arise whenever the water surface is uneven. Ground topography is easily handled and the flooding of previously dry areas can be simulated. However, since the equations do not capture the full velocity field within the water volume, they cannot model the dispersion relation. This results in less realistic wave animation than can be achieved with procedural methods.

The complete shallow water equations were solved by Layton and van de Panne [LvdP02] using an implicit, semi-Lagrangian approach that requires the solution to a linear system of equations at each time-step. Although stable, this technique suffers from excessive numerical dissipation, which makes it difficult to simulate detailed flows. Recently, Thürey et al. used similar techniques in a number of works targeting the animation of bubbles and foam [TSS+07], breaking waves [TMFSG07] and other large scale water phenomena [TRS06].

Kass and Miller [KM90] linearize the shallow water equations and drop the nonlinear advection terms by assuming that the water flows at a slow rate. The resulting equations are equivalent to the 2D wave equation and can be solved efficiently on a uniform grid using iterative algorithms such as the alternating-direction implicit method. This particular approach was later generalized by Wang et al. [WMT07] to handle flows over curved surfaces.

A solver for the 2D wave equation that instead uses explicit time integration and finite differencing is presented in [NN05]. Due to its speed and simplicity, this type of solver is almost certainly the most common method currently used to simulate water in interactive applications such as games. The accuracy of this approach was improved by Tessendorf [Tes04] by using a larger stencil in the finite difference discretization of the Laplacian operator. The coefficients of this stencil are further weighted by the fraction of each grid cell that is not occupied by an obstacle, a modification that allows waves to realistically reflect off boundaries. Kurtz and Tessendorf [KT07] combined this approach with the Fourier ocean synthesis technique [Tes04] by simply summing together the vertical displacements induced by each method.

O'Brien and Hodgins [OH95] also proposed an approach for simulating shallow
Figure 2.3: Vorticity visualization of a grid-based simulation of swirling fluid. The image on the left was computed using semi-Lagrangian advection with bilinear interpolation, while the right image used the monotonic bicubic interpolation scheme of [FSJ01]. The higher order interpolant uses additional data points to perform a sharper reconstruction of the advected velocity field which results in greater detail.

water flow that is mathematically equivalent to the 2D wave equation solvers. In this approach, virtual pipes connect each grid cell to the eight surrounding cells and the rate of flow in these pipes is determined by the hydrostatic pressure differences between adjacent grid points. This algorithm was recently extended to support flows over variable height terrain and mapped to the GPU [MDH07].

2.1.2 Grid-based Methods

Harlow and Welch [HW65] proposed a grid-based method called the marker-and-cell (MAC) method to solve the 3D Navier-Stokes equations, which was later introduced to the graphics community by Foster and Metaxas [FM96] in the first application of computational fluid dynamics techniques to the animation of 3D fluid flow. In this approach, the spatial derivatives of the Navier-Stokes equations are discretized on a fixed Cartesian grid using standard finite differencing and first order accurate forward Euler updates are used for discretization in time. Since this integration scheme is explicit, suitably small time steps are required for stability.

Stam [Sta99] relaxed this constraint with the introduction of an unconditionally stable model for fluids using semi-Lagrangian velocity advection [CIR52] and an implicit solver for viscosity. This model allows arbitrarily large time steps to be taken,
which effectively reduces computational costs by allowing simulations to be advanced over time at any speed. Unfortunately, semi-Lagrangian integration tends to dissipate a lot of energy due to the repeated averaging and interpolation within the discrete velocity field. This makes it difficult to simulate highly detailed, interesting fluid flow.

A number of modifications to counteract the artificial numerical dissipation inherent in semi-Lagrangian integration have been proposed. Fedkiw and Stam [FSJ01] use monotonic cubic interpolation in place of linear interpolation for sharper reconstruction of the advected velocity profile (see Figure 2.3). Song et al. [SSK05] use spatial derivatives to help reconstruct the advected function, while Kim et al. [KLLR07] take multiple advection steps, forwards and backwards in time, to predict an error term that can be subtracted to give better accuracy. Further techniques to combat dissipation include vorticity confinement [FSJ01], vortex particles [SRF05] and the particle-in-cell method [ZB05].

Another big problem with the earlier work of Foster and Metaxas [FM96] is the choice of massless marker particles to represent and track the liquid volume as it moves around. These particles are used to locate the liquid surface, which is required for rendering and the application of boundary conditions at the liquid-air interface. The problem is that these particles provide more detail than can be resolved by the grid-based simulator, resulting in noisy surfaces that are difficult to render convincingly. Foster and Fedkiw [FF01] solved this problem by implicitly capturing the liquid interface location as the zero isosurface of a signed distance function sampled on the simulation grid. This function is updated using the level set method [OS88], which requires the solution to an advection equation. As with velocity advection, this results in excessive numerical dissipation, which manifests itself as nonphysical volume loss errors (see Figure 2.4). Enright and Fedkiw [EMF02] reduced these errors by applying the hybrid particle level set method [EFFM02]. This approach combines the Eulerian level set with Lagrangian particles that are seeded in the vicinity of the zero isosurface and used to rebuild the level set in under-resolved regions. Lastly, Houston et al. [HNB06] recently introduced an efficient data structure that makes it practical to use very high resolution level sets for liquid representation.

Another very active area of research concerns the choice of boundary conditions that dictate how fluid behaves in the vicinity of static and dynamic objects. Early approaches such as [FM96] and [Sta99] involved voxelizing objects onto the Cartesian
grid, which is only accurate if object boundaries line up with grid lines. Since this is rarely the case, artifacts such as unnatural dampening of the flow occur as a result of the “stairstep” approximation of smooth surfaces. Several different methods have been suggested to manipulate velocity and pressure values in the vicinity of objects, in order to more faithfully match the object’s true shape. Foster and Fedkiw [FF01] use the object’s normal at a grid point to set velocities so that fluid can flow freely tangential to the object but is restricted from flowing into the object. Houston et al. [HBW03] and Rasmussen et al. [REN04] elaborated on this by using a constrained velocity extrapolation method to define velocities throughout solid obstacles. While these approaches improved the accuracy of velocity advection, they still resorted to voxelized object representations when computing pressure values. Roble et al. [RZF05] attempted to mitigate this problem by redefining the velocity divergence in a cell to take into account the subcell geometry of objects. A different approach, introduced by Batty et al. [BBB07], weights velocity and pressure values by the volume of fluid within cells after accounting for the presence of objects. This achieves very realistic results and subgrid resolution flows.

Another approach that achieves very accurate interaction between fluids and objects was proposed by Feldman et al. [FOK05]. They propose a hybrid method that combines a Cartesian grid in open regions with an unstructured tetrahedral grid in the vicinity of objects. Since these grids can be fitted to the exact shape of an object, it is trivial
to enforce accurate boundary conditions. Later work [FOKG05, KFCO06, CFL+07] abandoned Cartesian grids entirely in favour of unstructured grids. Unfortunately, these approaches have a number of disadvantages compared to regular grids, including slower data access, more expensive and dissipative interpolation operations, and the need to perform remeshing whenever liquid or object boundaries move.

Related to the definition of boundary conditions is the problem of computing two-way coupling between objects and fluids so that fluids respond to the movement of objects, and objects respond to forces arising from the fluid. Takahashi et al. [TFK+03] use a simple method to compute forces and torques on voxelized objects by integrating pressure over those cell faces occupied by an object. Carlson et al. [CMT04] treat the entire simulation domain (including rigid objects) as fluid. After updating the fluid, they compute the new motion of the objects by projecting the relevant fluid velocities back onto rigid body motions. A novel coupling approach that also treats infinitesimally thin and deformable objects was proposed by Guendelman et al. [GSLF05]. Their approach computes a smoother pressure field for improved coupling, but this requires solving an additional matrix equation for pressure, which significantly increases the expense of the method. Chentanez et al. [CGFO06] presented an approach that solves for fluid and deformable object motion simultaneously, instead of one after the other. This requires solving a large, non-symmetric linear system of equations, the upside of which is improved stability compared to previous coupling techniques. Finally, Batty et al. [BBB07] proposed a new coupling approach for rigid bodies that achieves very accurate and robust results even on relatively coarse grids. Their approach takes the kinetic energy of objects into account in order to compute fluid pressure values consistent with object motion.

Finally, in order to make more efficient use of computational resources, there have been recent attempts to replace uniform grids with data structures that dynamically adapt themselves to the visually interesting regions of fluid flow. Losasso et al. [LGF04] advocated using an octree data structure, where grid cells are coarsened away from the liquid-air interface to provide higher detail at the surface and less detail within the volume. Although fine scale detail is demonstrated, the authors do not report on the performance of their approach compared to uniform grids of the same effective resolution. A similar approach is taken by Kim et al. [KLL+07] with the use of a multigrid solver instead of the conjugate gradient method to improve convergence.
Irving et al. [IGLF06] take a slightly different approach that combines the efficiency of heightfield methods with the detailed surface motion made possible with grid-based liquid solvers. Their approach works by replacing groups of stacked cells that are far away from the surface with single, tall cells that assume hydrostatic pressure profiles.

### 2.1.3 Particle-based Methods

Particle-based methods describe fluid flow in terms of the positions and velocities of a collection of disordered points that move with the local fluid velocity. Fluid behavior is then modeled by computing forces between each particle and the other particles in its vicinity. Unlike grid-based methods, particle-based methods do not have any intrinsic difficulty representing sharp features in a fluid flow. Since they are not limited by the resolution or bounds of a grid, they can easily capture small-scale visual details and accurate interaction with collision objects anywhere within the scene. In addition, since velocity advection is performed in Lagrangian coordinates by simply updating the positions of particles, these methods do not suffer from the numerical dissipation problems that plague grid-based methods. As such, particle-based methods have become a popular alternative to grid-based methods for animating liquids.

Reeves [Ree83] introduced the idea of uncoupled particle systems for modeling simple fluid behavior. This approach is very fast and is used frequently in interactive applications to animate liquid phenomena where it is reasonable to assume that particles do not interact with one another. Hence, while it may be suitable for rain, fountains or jets of water, the absence of pressure forces between particles limits the range of effects that can be simulated. A number of papers extended the basic particle system idea with coupling forces acting between pairs of particles to simulate viscous fluids [MP89, Ton91] and melting objects [TPF89]. A particular particle-based method known as smoothed particle hydrodynamics (SPH) was first introduced to the graphics community by Stam and Fiume [SF95] to depict fire and other gaseous phenomena. This technique was later applied to the animation of deformable bodies [DG96], the flow of lava [SAC+99] and eventually water [MCG03].

The interaction forces described in [MCG03] are designed to provide a discretization of the Navier-Stokes equations without requiring any underlying grid. Pressure forces support the liquid by pushing particles apart so that they are equally spaced, while
viscosity forces simulate internal friction in the fluid by reducing the relative velocity between nearby particles. One of the big advantages of the SPH formalism is its simplicity; accurately tracking the fluid is trivial and mass conservation is automatically guaranteed since the fluid mass is represented by a fixed number of particles. Furthermore, there is no need to solve a complex system of linear equations to compute the pressure field since it is diagnosed directly from the spatial distribution of particles.

In recent years, a number of improvements have been proposed to the SPH algorithm presented in [MCG03]. Müller et al. [MSKG05] simulate air particles in addition to water particles to enable bubble formation and effects such as boiling. Clavet et al. [CBP05] suggest new ways to handle surface tension, elasticity and plasticity, as well as a new relaxation scheme to compute inter-particle pressure forces that results in faster and more stable simulation. Kipfer and Westermann [KW06] present an efficient data structure to simulate rivers with SPH, while Müller et al. [MST+04] and Solenthaler et al. [SSP07] both propose techniques for computing two-way coupling forces between fluids and rigid or deformable objects. Becker and Teschner [BT07] present an improved surface tension model and demonstrate improved realism due to reduced compressibility when the linear equation of state relating density and pressure is replaced with a non-linear Tait equation. Finally, Adams et al. [APKG07] present an adaptive simulation algorithm that uses a denser sampling of particles near the liquid surface and fewer particles deep within the volume.

While the SPH algorithm can simulate very realistic liquid motion, it cannot simulate truly incompressible liquids. This is due to the fact that pressure forces are "short range" forces, inferred from local density fluctuations within the fluid. These pressure forces take time to propagate through the medium, which results in the fluid appearing slightly compressible. While this may seem to be a better model for the physical nature of liquids, in practice it is difficult to reduce the compressibility of the fluid to the point where it looks incompressible and still remains stable. As a result, liquids simulated with SPH tend to appear "bouncy" when they come to rest under gravity and are therefore better suited to transient effects such as splashing liquid where compressibility is less noticeable. To counteract this problem, Premoze et al. [PTB+03] introduced a different particle-based discretization of the Navier-Stokes equations based on the Moving-Particle Semi-Implicit (MPS) method [KO96]. Similar to grid-based methods, this approach enforces incompressibility by computing the pressure field as the solution
to a large, linear system of equations. While this allows larger time steps to be taken and achieves greater realism, it is significantly slower than simulating compressible liquid flow using SPH.

2.1.4 Spray and Foam

There are many cases where the addition of fine scale bubbles and droplets can greatly enhance the realism and detail of animated water. This is particularly true of ocean environments where spray and foam play a crucial role in the visual representation of breaking waves. The early work of Fournier and Reeves [FR86] and Peachey [Pea86] both define conditions to indicate the onset of breaking waves and then spawn spray particles when these conditions are satisfied. O’Brien and Hodgins [OH95] generate droplet particles when the upward velocity of the water surface exceeds a threshold, while a similar technique is used for feature film visual effects in [KT07]. Froemling et al. [FGP07] use hand-placed emitters to trigger the generation of spray whenever the simulated liquid flow falls into their region of influence. Bredow [Bre07] and Thürey et al. [TMFSG07] generate particle systems at the lips of breaking waves and also along the intersection curve where breaking waves overturn fully and contact the water below. Takahashi et al. [TFK+03] augment a grid-based liquid simulator with the generation of spray particles in regions where the liquid curvature exceeds a threshold. These particles transition to foam particles when they fall back onto the water surface.

A popular technique for simulating splashing detail in the context of the particle level set method is to generate droplet particles in regions where the detail of the liquid surface is too fine to be resolved by the level set grid [KCC+06, GSLF05, GLR+06]. A similar approach is used to generate bubbles by detecting when air particles are trapped within the level set [GH04]. When these bubbles reach the liquid surface they are simulated using the model introduced by Kück et al. [KVG02]. In this approach, attractive and repulsive spring forces are applied to simulate the coalescence of bubbles into larger foam structures. Similarly, Thürey et al. [TSS+07] model the natural clustering of foam bubbles using an SPH simulation with surface tension.

More accurate models of foam have been developed by simulating bubbles as thin sheets of liquid, governed by the Navier-Stokes equations [ZYP06, KLL+07]. Cleary et al. [CPPK07] take an intermediate approach and couple the physical behaviour
of bubble particles to an SPH fluid solver. While these approaches achieve realistic results, they are often too slow to simulate dense surface foam at the scale of individual bubbles. As such, foam is usually approximated as non-interacting aggregate particles that represent larger collections of foam bubbles. In [GLR+06] and [FGP07], these particles are generated when spray particles collide with the water surface and are subsequently advected by the fluid velocity. In [BC06], foam particles are generated at the intersection between object geometry and the water surface. These particles then follow a simple procedural laminar flow field, which forces particles to flow around objects without penetrating them.

2.2 Liquid Rendering

The optical properties of liquids are understood very well in computer graphics and can be modelled efficiently using standard global illumination techniques such as ray tracing [Whi80], path tracing [Ka86] and photon mapping [Jen96]. These ray-based algorithms are particularly efficient at rendering highly specular liquid surfaces and can easily model the reflection and refraction of light that gives rise to complex phenomena such as caustics. However, these techniques are generally not applicable to real-time rendering of liquids. In fact, considering that most 3D fluid simulators are not yet capable of interactive performance, most of the real-time techniques used to render liquids are tailored for heightfield representations of the liquid surface. In their earlier work, Fournier and Reeves [FR86] used environment-mapping to model skylight bouncing off water surfaces. They also used bump-mapping to simulate high frequency wave detail and specular lighting to simulate sun glitter. Kass and Miller [KM90] added a refraction map to model light transmitted through the water and then blended the reflection and refraction contributions based on the Fresnel equations. These techniques are now used as standard to render oceans and other large bodies of water in real-time.

In recent years, a number of improvements have been adopted to increase the range of effects that can be modelled in real-time. Jensen and Golić [JG01] render reflection and refraction maps on the fly and use projective texturing when rendering the heightfield to achieve dynamic, local reflections and refractions of objects. They also render underwater godrays using slice-based volume rendering and simulate caustics by rendering the refracted projection of water surface triangles into a texture map.
and applying this to underwater objects. A similar approach to rendering caustics is presented in [Lov03]. Recently, more accurate methods that handle non-heightfield geometry have been presented for rendering refractions [Wym05, DW07] and caustics [SKP07]. Finally, Baboud and Décoret [BD06] encode the ground topography as a second heightfield with which they perform ray intersections on the GPU to compute accurate underwater refractions and light absorption.

In most cases, liquid needs to be treated as a participating medium to achieve a realistic appearance. Although absorption has been accounted for in liquid rendering, we are not aware of any work that presents a practical model for real-time light scattering in liquids. Furthermore, while heightfields provide many opportunities for simplifying the problem of computing interactions between light and liquid, the heightfield representation is limited to placid bodies of liquid and is not a suitable representation for 3D effects such as splashing. Therefore, the remainder of this chapter will discuss previous work that pertains to non-heightfield liquid representations and the handling of light scattering within liquids.

### 2.2.1 Liquid Representations

In order to render realistic 3D liquids, there are two requirements: the ability to locate the liquid surface and the ability to compute normal vectors at surface points. It is particularly important that these normal vectors are well-defined and vary smoothly over the surface. Since these vectors are required to compute the directions of reflected and refracted light, any irregularities in the normal field can show up as artifacts in what should appear as a smooth, glassy surface. In cases where liquid is represented as a collection of particles, methods have been proposed to render them using point-based techniques such as surface splatting [IDYN06, BT07] or ray tracing [AKP+05]. However, these techniques generally require a dense sampling of points to reconstruct a continuous and smooth surface and have therefore not received as much attention as the techniques described below.

In most 3D fluid simulators, a smooth implicit surface is used to represent the liquid at render time. The liquid surface is generally defined as the zero isosurface and normals are given by the gradient vector at any point. In the case of grid-based simulators that use the level set method, a signed distance field representing the liquid is automatically
available. In simulators that use marker particles, or in particle-based simulators, an implicit scalar function can instead be easily constructed from the positions of particles. For example, Müller et al. [MCG03] defined a smooth density field from the spatial distribution of SPH particles. One problem with their approach is the difficulty of representing sharp features and flat surfaces. Zhu and Bridson [ZB05] improved on this by using particles to construct a signed distance field on a high resolution grid and smoothing the result. Even better results were obtained by Adams et al. [APKG07] by assigning distances to adaptively sampled particles and using these distances for surface reconstruction. Shen and Shah [SS07] used velocity information to perform temporally coherent surfacing of particles, while Museth et al. [MCZ07] proposed optimizations to make these techniques practical on very high resolution grids.

Figure 2.5: The left image shows our implementation of a level set polygonized using marching cubes [LC87] and rendered in OpenGL. The right image shows the same level set raytraced using sphere tracing [HSK89].

Given an implicit surface representation of liquid, a very common approach to visualize the zero isosurface is to apply a contouring procedure to extract a polygonal surface approximation, which can then be raytraced or rasterized efficiently by graphics hardware. The original marching cubes algorithm was developed by Lorenson and Cline [LC87] to extract triangle meshes from general implicit functions (see Figure 2.5 (left)). Kobbelt et al. [KBSS01] later extended this algorithm to compute a better fitting surface by using normal vectors to identify and extract sharp edges and corners. Ju et al. [JLSW02] simplified this algorithm and adapted it to work with an octree discretization instead of a uniform grid. A very different contouring approach for level
set fluids was presented by Geiger et al. [GLR+06]. Instead of trying to fit a polygonal mesh to the zero isosurface, they instead compute a high resolution voxelization of the isosurface and then project each vertex of the exterior voxel faces onto the zero isosurface along the gradient direction. Finally, they apply Laplacian mesh relaxation techniques to fake the smoothing effect of surface tension. Lastly, a novel technique was recently introduced by Müller et al. [MSD07] to visualize particle-based fluids with automatic view-dependent level of detail. Their technique applies marching squares to compute a 2D mesh in screen space that is subsequently projected back into world space for rendering.

![Image](image-url)

Figure 2.6: The discrete signed distance function given by the union of a group of spherical particles is smoothed and raytraced using sphere tracing. The left image shows our implementation result using trilinear interpolation to reconstruct the continuous function and the right image shows the result using the tricubic Catmull-Rom interpolant of [ML94].

While contouring techniques provide a very convenient way to visualize implicit surfaces, non-adaptive algorithms such as marching cubes can be very inefficient for highly detailed liquids. A very fine spatial discretization is required to capture small scale detail and this can consume large amounts of memory. An alternative approach is to directly ray trace the implicit surface. A number of root-finding techniques from mathematics have been adapted for this purpose. Examples include polynomial root solving [Han83], interval analysis [Mit90] and Lipschitz methods [KB89]. Of particular relevance to liquid rendering is the method of sphere tracing [HSK89], which was designed by Hart et al. to provide efficient and robust ray tracing of implicit surfaces defined by signed distance functions (see Figure 2.5 (right)). Foster and Fedkiw [FF01]
were the first to use this technique to raytrace level set liquids. Enright et al. [EMF02] used the same approach in conjunction with photon mapping and irradiance caching to render realistic water. Depending on the accuracy required, they used either a trilinear or a tricubic filter [ML94] to reconstruct the continuous signed distance function from discrete samples (see Figure 2.6). They also supported motion blur by interpolating between signed distance functions to compute the surface at intermediate times between frames. Finally, Kim et al. [KCC06] adapt sphere tracing to perform on-the-fly reconstruction of a smooth surface from blended spherical particles representing turbulent splashing water, while Darles et al. [DCG07] adapt sphere tracing for the case of ocean surfaces defined by a sum of parametric waves.

2.2.2 Light Scattering

Liquids generally need to be treated as participating media in order to be rendered accurately. Although this is usually not necessary for rendering small scale water, it is certainly true for large bodies of water such as the open ocean where water molecules and tiny bits of organic matter cause significant amounts of absorption and scattering. These complex interactions make it very difficult to accurately model the behaviour of light within the liquid volume. Under normal circumstances, liquids can be assumed to be homogeneous materials, thus greatly simplifying the light transport problem. However, in this thesis we are also concerned with lighting inhomogeneous volumes for effects such as spray and foam. As such, we will now provide a brief review of the participating media literature that is most relevant to the aims of this thesis but will not review the large number of methods that have been used for volumetric lighting such as Monte Carlo integration, line integrals, discrete ordinates, finite elements and spherical harmonics.

Considering the complexity of computing light transport in translucent materials such as liquids, it is not surprising that a number of approximations have been proposed to simplify the problem. One of the first developments was the work of Hanrahan and Krueger [HK93] in which they present an analytical solution for single scattering in stacked layers of different homogeneous media, and an accurate Monte Carlo algorithm to compute the multiple scattering contribution. Bidirectional path tracing was extended by Lafortune and Willems [LW96] to render participating media. Pauly et al.
[PKK00] present a generalization of the Metropolis light transport method to include volume scattering, and Premoze et al. [PAS03] use path integral theory to approximate radiative transfer in forward-scattering, inhomogeneous media.

Jensen and Christensen [JC98] extend the two-pass photon mapping algorithm [Jen96] to scenes with participating media. In the first pass they trace photons through the scene, computing their interaction with participating media and storing the results in a volume photon map. In the second pass they compute direct and indirect illumination within media by using ray marching to compute the single scattering term and the volume photon map to estimate radiance due to multiple scattering. Although their method is practical and efficient in many cases, it is generally considered intractable for highly scattering media due to exorbitant memory requirements [DJ07].

More recently, Jensen et al. [JMLHO1, JB02] introduced a fast approximation for subsurface light transport in homogeneous media. Their approach combines an accurate single scattering computation with a dipole point source diffusion approximation for multiple scattering. Although their method generally produces excellent results for optically thick and highly scattering materials, the underlying assumptions cause inaccuracies for many geometrically complex objects, in particular, those with optically thin features. Donner et al. [DJ07] overcame these limitations by tracing photons inside the medium to capture inter-scattering between surfaces, and applying a quadpole diffusion approximation to improve accuracy.

Chen et al. [CTW"04] makes use of Jensen’s work to render inhomogeneous objects by splitting the volume into an inhomogeneous outer layer and a homogeneous inner core. The reflectance field of the voxelized surface layer is precomputed using volume photon mapping, while the homogeneous region approximates the inner core of an inhomogeneous object. This is modelled using the dipole approximation based on the assumption that the light reaching this inner core has diffused due to scattering. A similar approximation that uses a homogeneous inner core was employed by Li et al. [LPT05]. They propose a hybrid Monte Carlo method for subsurface scattering that combines path tracing to model scattering events occurring near the surface, with the dipole diffusion approximation to handle paths that penetrate into an isotropic core region. Their approach produces results comparable to a full Monte Carlo simulation at a fraction of the cost. More recently, Frisvad et al. [FCJ07] proposed a similar modification based on similarity theory to speed up Monte Carlo path tracing of homo-
geneous liquids. When ray paths penetrate far enough into the medium, they switch to an isotropic phase function and scale the scattering coefficient to compensate. In this way, they significantly reduce the number of scattering events while incurring a negligible amount of error. Another approach suitable for rendering quasi-homogeneous media such as liquid foam was presented by Moon et al. [MWM07]. Instead of simulating individual scattering events for paths inside a medium, they sample from precomputed scattering solutions which allows them to traverse the medium in large strides.

As well as techniques to accurately model light transport in volumes, many recent developments have focussed on rendering volumetric materials at interactive rates. Premoze et al. [Pre03] provides a good review of atmospheric scattering and presents methods to render realistic skies in real-time which are also pertinent to liquid rendering. Mertens et al. [MKB+03] introduce an importance sampling scheme to accelerate subsurface scattering in homogeneous media. Dachsbacher and Stamminger [DS03] render subsurface scattering effects by extending the concept of the shadow map to handle translucency. Hegeman et al. [HAP05] apply a fast lighting model to the more general case of inhomogeneous participating media, while Kniss et al. [KPHE02] present an interactive shading model that captures volumetric light attenuation and shadows.

For the specific case of ocean rendering, a number of models have been developed that consider the effects of light scattering in water. Nishita and Nakamae [NN94] render underwater caustics and shafts of light while taking into account the color of water. They derive an analytical function to compute the radiance along a ray after absorption and scattering, but they assume that in-scattered light coming from above the water surface is constant and non-directional. Premoze and Ashikmin [PA00] instead consider ocean surfaces seen from above. They simplify the light transport problem by ignoring directional in-scattering and instead treat the combined effect of scattering as a constant, diffuse radiance field. They integrate this field over the depth of the ocean while accounting for attenuation and scattering to achieve realistic results for a variety of ocean conditions. Although we are not aware of any ocean lighting models that take full multiple scattering into account, Iwasaki et al. [IDN03] do consider second order scattering effects using a two-pass approach. In the first pass, they precompute the radiance at sampled points on a set of horizontal slices within the water volume and in the second pass they composite these slices using an accumulation
2.2.3 Spray and Foam

Spray and other small scale splashing detail is usually best simulated using particles. However, for the purposes of rendering, wrapping a geometric surface around the particles or converting them to a level set for raytracing is usually impractical in terms of memory and processing requirements. In many cases, it suffices to assume that particles represent individual droplets which do not coalesce into larger volumes. Furthermore, by assuming that droplets are well-separated, it is often reasonable to ignore inter-reflections and light each independently of the rest. This makes it possible to render large particle counts at acceptable rates.

An obvious example where it is appropriate to explicitly render water droplets individually is the case of rain. There has been a large amount of work done in this area in recent years and, although none of the developed methods consider the effects of absorption or scattering, they do present accurate geometric and photometric models of rain drops that are very related to the issues involved in rendering splashes and spray. Rousseau et al. [RJG06] render real-time rain drops as normal-mapped billboards which they environment-map based on the directions of per-pixel refraction rays. Tatarchuk and Isidoro [TI06] use a similar approach but instead of rendering each rain drop individually, they render a quad covering the entire screen. This quad is mapped with a vertically scrolling normal map to give the illusion of falling rain. They also add glowing and motion blur effects via post-processing blur filters. Wang et al. [WLF+06] present a rain appearance model that uses precomputed radiance transfer to accurately light rain drops in real-time based on illumination from a background environment map. In order to capture the effects of deforming rain drops and motion blur, Garg and Nayar [GN06] use ray tracing to precompute a database of rain streak textures that are indexed by lighting and viewing directions. Their model captures shape distortions induced by periodic oscillations in falling drops and this results in realistic looking, broken specular highlights along rain streaks. Finally, in [GKN07], Garg and Nayar render water drop splashes by ray tracing a single transparent sphere from a given viewing direction, and re-use the result in combination with multi-sampled motion blur to render all droplets in a scene. This is motivated by the observation that,
since all droplets see roughly the same environmental illumination, their appearances can be assumed to be more or less the same.

While droplets can be rendered rapidly when lighting and shadowing are ignored, in some cases, volumetric illumination is required to achieve realistic results. For example, unlike rain, ocean spray and foam is usually modelled as a large collection of very small, closely packed particles. In this case, multiply scattering of light through droplets (in the case of spray) or bubbles (in the case of foam) produces complex effects such as subsurface scattering and volumetric shadows. Peterson [Pet03] uses particles to render a waterfall and foregoes the use of traditional surface lighting techniques. Instead he relies solely on shadows cast by a single light source to convey shape and texture. This idea of treating spray as an inhomogeneous participating medium without regard for specular surface reflections and refractions is often faster and produces better results than attempting to render spray as explicit surface geometry. The same argument holds for many other types of phenomena that are made up of a large number of tiny particles such as clouds, smoke or explosions. In these cases, the medium is usually represented as a 3D array of density samples that can be rendered using traditional volume rendering techniques. For example, Fedkiw et al. [FSJ01] render smoke at interactive rates by compositing volume slices and Rasmussen et al. [RNGF03] render explosions using ray marching. Directly rendering the variable density field is appropriate for these phenomena since they tend to have a smoothly varying appearance. However, in cases where finer detail is required, alternative methods have been developed. For example, Dobashi et al. [DKY+00] present a two-pass particle splatting algorithm for rendering self-shadowed clouds and this technique was later used to render particulate spray and foam by Takahashi et al. [TFK+03]. Another example is rendering fur or hair geometry that is too fine to sample on a grid. Bertails et al. [BMC05] and Mertens et al. [MKBR04] both sample hair strand geometry onto a 3D density field that is used to compute a coarse lighting solution, but they do not render the density field itself. We are not aware of any published work that considers light scattering for particle systems. In fact, there has been surprisingly little work done in the area of particle rendering and lighting since the original work of Reeves [Ree83]. However, deep shadow maps [LV00] and opacity shadow maps [KN01] provide a general means to perform fractional visibility queries and these have been used in feature films to successfully render large amounts of particles representing ocean spray [Bre07].
The problem of rendering surface foam is similar to the case of spray and the appropriate rendering techniques are dictated in large part by the scale at which the medium is observed. In the same way that spray is commonly rendered as an inhomogeneous medium under the assumption that individual water droplets are too small to be visible, foam is normally rendered assuming that individual bubbles are indiscernible. However, unlike spray, foam is not usually treated as a participating medium. Instead, all of the methods we have seen treat foam as a flat or displaced layer above the water surface and rely on simple techniques to achieve the desired look. This is usually adequate considering that, in many circumstances, spray particles will occlude the surface foam to some degree. Larsson et al. [LZD06] render foam trails using texture mapped procedural noise patterns placed according to the position of moving objects. Takahashi et al. [TFK+03], and Brown and Collier [BC06], render foam particles directly as surface oriented sprites. In [GLR+06], Geiger et al. use foam particles to create a layer of displaced, diffusely reflective foam. Froehling et al. [FGP07] use the density of foam particles at a point on the water surface to control the colour and displacement of the foam, and Kluyskens [Klu07] uses foam density to control the placement, deformation and fading of precomputed, procedural noise textures.
Chapter 3

Fluid Simulation Concepts

Many different techniques, such as vorticity approaches [ANSN06], and the Lattice Boltzmann Method [TIR06], have been explored as tools for solving the full Navier-Stokes equations with the goal of animating realistic fluid motion. However, two approaches have dominated the work in the computer graphics literature – the marker-and-cell (MAC) method (see Figure 3.1 for an example result) and smoothed particle hydrodynamics (SPH) (see Figure 3.4). These two approaches have complementary strengths and weaknesses and, as such, the choice of a method usually depends on its suitability for a particular application. For example, one problem with the MAC method is its inability to capture very small scale visual details on coarse grids. On the other hand, SPH is not limited by the resolution of a grid and is therefore much better at simulating effects such as splashing water and turbulent flows. Unfortunately, the cost per particle in these methods is much greater than the cost per cell in grid-based methods. The use of SPH is therefore usually limited to transient effects such as pouring or splashing liquids, rather than large scale effects such as a swimming pool or ocean that would require a prohibitive amount of particles. The MAC method is much more economical for these kinds of simulations that involve large volumes of water and therefore both methods have enjoyed success in computer animation applications. This chapter begins with a brief review of the equations of fluid motion and then describes how these equations can be solved using either the MAC method, or SPH. The theoretical ideas behind these numerical solution techniques provide important background for understanding the motivation behind the development of the novel fluid
solver presented in the next chapter.

3.1 Navier-Stokes Equations

A fluid is a continuum. To describe it, its properties need to be defined at every point in space that the fluid occupies. Furthermore, it is usually necessary to explicitly specify the fluid’s properties at boundaries where it meets other states of matter. These boundary conditions are necessary to model the interaction between the fluid and the adjacent matter, be solid, liquid, or gas.

From an animation perspective, the only property of a fluid we need to know about is its velocity, $\mathbf{u}(x, y, z, t)$. If we can determine a time-dependent velocity field, then all we need to achieve our goal of animation is to use some device (particles, a level set, etc.) to represent the fluid and update this over time. In practice, to simulate fluid motion, we also need to know its density, $\rho(x, y, z, t)$, and its pressure, $p(x, y, z, t)$.

The compressible Navier-Stokes equations provide a mathematical description for how these fluid properties evolve over time. The first equation, the momentum equation, describes how fluid mass accelerates due to self-advection and the local forces applied to it:

$$\frac{\partial \mathbf{u}}{\partial t} = -\mathbf{u} \cdot \nabla \mathbf{u} + \frac{\mu}{\rho} \nabla \cdot (\nabla \mathbf{u}) - \frac{1}{\rho} \nabla p + \mathbf{g}$$

where $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$ is the gradient operator, $\mu$ is the kinematic viscosity (rate at which velocity variations are damped out due to internal friction) and $\mathbf{g}$ is the external body force (e.g., gravity).

The second equation, known as the continuity equation, ensures that, independent of whether the fluid is compressible or incompressible, the total amount of mass in our closed system is conserved:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

The momentum equation defines how to update each component of the velocity field, and the continuity equation defines how to evolve the density field. In order to close the system of equations, a final equation of state (EOS) is necessary to determine the pressure field. For simplicity, we use an EOS that neglects the effects of
thermodynamics:

\[ p = f(\rho) \]  

(3.3)

The function \( f \) that relates density to pressure depends on the physical properties of the fluid and will be defined later. We now have a complete system of equations that can be integrated over time to simulate the behavior of the fluid. The last thing we require are boundary conditions to describe what happens at fluid surfaces. In the case of a solid obstacle, the velocity field should prevent fluid from flowing into or out of the obstacle. We can write this as:

\[ \vec{u} \cdot \vec{n} = \vec{u}_{\text{solid}} \cdot \vec{n} \]  

(3.4)

where \( \vec{n} \) is the normal vector at a point on the obstacle, and \( \vec{u}_{\text{solid}} \) is its velocity. Notice that, for the purposes of animation, we choose not to constrain the tangential component of velocity. This allows the fluid to slip freely past the surface without sticking to it.

In the case of a liquid-air interface or free surface, the air phase is typically treated as a vacuum instead of a second fluid and no constraints are placed on the velocity field at the boundary. Instead, the assumption is made that the pressure at the boundary is at constant atmospheric pressure. Since we are only interested in the pressure gradient in Equation (3.1), this constant can be chosen arbitrarily and the boundary condition (assuming no surface tension) can be written:

\[ p = 0 \]  

(3.5)

3.2 Marker-and-Cell Method

3.2.1 Incompressible Euler Equations

For simulating fluids such as water on grids, it is common to simplify the compressible Navier-Stokes equations in two ways. The first simplification is to neglect the viscosity term in Equation (3.1) to give:

\[ \frac{\partial \vec{u}}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla p + \vec{g} \]  

(3.6)
If desired, the viscosity term can easily be accounted for using, for instance, a finite difference discretization [FM96], or an implicit scheme [Sta99]. However, as discussed in Section 2.1.2, most solvers based on the MAC method already have a “built-in” viscosity term that arises artificially from averaging and interpolating quantities on the grid. Since this numerical viscosity is generally larger than the physical viscosity of liquids (such as water and milk) that we wish to simulate, it makes no sense to add any extra viscosity. When applied to inviscid fluids, the Navier-Stokes equations are instead referred to as the Euler equations.

The second simplification is to treat the fluid as being perfectly rigid instead of very stiff. This assumes that density variations are negligibly small which allows us to replace the spatially variant density field with a constant value. As a result, Equation
(3.3) can be ignored and Equation (3.2) can be simplified by zeroing the first term and removing the \( \rho \) multiplier from the second term to give:

\[
\nabla \cdot \vec{u} = 0 \tag{3.7}
\]

This equation enforces a divergence-free velocity field that eliminates compression or expansion in the flow. Since density is constant, an equation of state can no longer be used to diagnose the pressure field. Instead, it needs to be determined by solving for the pressure that results in an incompressible fluid, i.e., one that satisfies Equation (3.7).

### 3.2.2 Projection Method

In order to advance the velocity and pressure fields over a single time step \( \Delta t \) from time \( t_n \) to time \( t_{n+1} \) we use the projection method of Chorin [Cho68]. To explain this we begin by discretizing the \( \partial \vec{u} / \partial t \) term of Equation (3.6) using forward differencing\(^1\). This allows us to write Equation (3.1) as:

\[
\vec{u}^{(n+1)} = \vec{u}^{(n)} + \Delta t \left( -\vec{u} \cdot \nabla \vec{u} - \frac{1}{\rho} \nabla p + \vec{g} \right) \tag{3.8}
\]

where the superscripts indicate the time at which a variable is referred to, e.g., \( \vec{u}^{(n+1)} \equiv \vec{u}(t_{n+1}) \). In order to solve this equation, while at the same time respecting Equation (3.7) to enforce incompressibility, we first compute an intermediate velocity field \( \vec{u}^* \) that ignores the pressure term:

\[
\vec{u}^* = \vec{u}^{(n)} + \Delta t \left( -\vec{u}^{(n)} \cdot \nabla \vec{u}^{(n)} + \vec{g} \right) \tag{3.9}
\]

Equation (3.8) can now be rewritten as:

\[
\vec{u}^{(n+1)} = \vec{u}^* - \frac{\Delta t}{\rho} \nabla p^{(n+1)} \tag{3.10}
\]

\(^1\)This explicit, first order temporal discretization scheme is more commonly known as Euler's method.
This equation refers to the pressure at time $t_{n+1}$. The reason for this is that Chorin’s solution method is \textit{implicit} in pressure and \textit{explicit} in velocity. This means that the pressure field at time $t_{n+1}$ is computed directly from $\mathbf{u}^*$ based on the fact that the velocity field at time $t_{n+1}$ should be divergence-free (i.e., incompressible). This can be expressed mathematically:

$$\nabla \cdot \mathbf{u}^{(n+1)} = \nabla \cdot \left( \mathbf{u}^* - \frac{\Delta t}{\rho} \nabla p^{(n+1)} \right) = 0$$ \hspace{1cm} (3.11)

After rearranging this we get a so-called \textit{Poisson equation} for the pressure:

$$\nabla \cdot \left( \frac{1}{\rho} \nabla p^{(n+1)} \right) = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^*$$ \hspace{1cm} (3.12)

By solving this equation we can compute the pressure field at time $t_{n+1}$. Equation (3.10) can then be solved by subtracting the gradient of the pressure field from the intermediate velocity field to produce a final velocity field that satisfies both the momentum equation and the incompressibility condition at time $t_{n+1}$.

Notice that, since the pressure field is defined implicitly from the intermediate velocity field $\mathbf{u}^*$ and the incompressibility constraint, the scale of the pressure field doesn’t matter as long as we treat it consistently between Equations (3.12) and (3.10). This allows us to simplify these equations, and remove the notion of density altogether, by defining a scaled pressure of $p^* = p^{(n+1)} \Delta t/\rho$. We can now rewrite Equation (3.12) as:

$$\nabla^2 p^* = \nabla \cdot \mathbf{u}^*$$ \hspace{1cm} (3.13)

and Equation (3.10) as:

$$\mathbf{u}^{(n+1)} = \mathbf{u}^* - \nabla p^*$$ \hspace{1cm} (3.14)

\subsection{3.2.3 MAC Grid Discretization}

In order to solve the Euler equations numerically, the simulation domain is discretized by approximating the functions $\mathbf{u}$ and $p$ by a finite number of equally spaced samples. This makes it easy to evaluate spatial derivatives and interpolate quantities at arbitrary points within cells. However, instead of colocating the velocity and pressure samples, the MAC method uses a special grid arrangement that is necessary to rigorously enforce
mass conservation. In the two-dimensional case, the pressure samples are located
at the centre of grid cells, the horizontal component of velocities are located at the
midpoints of vertical cell edges, and the vertical component of velocities are located at
the midpoints of horizontal cell edges. This configuration of field variables is referred
to as the MAC grid and is illustrated in Figure 3.2.

The projection method can now to be applied to update the velocity field from
\( \vec{v}(n) \) to \( \vec{v}(n+1) \). First, the intermediate velocity field \( \vec{u}^* \) is computed from Equation
(3.9). Next, Equation (3.13) is solved to compute the pressure field \( p^* \). Finally, the
new divergence-free velocity field \( \vec{v}(n+1) \) is computed from \( \vec{u}^* \) and \( p^* \) using Equation
(3.14). The most straightforward way to perform these steps is to discretize all of the
differential operators that appear in the aforementioned equations using a combination
of forward and central difference approximations (see for example [FM96]). In partic­
ular, the discretization of the Laplacian operator in Equation (3.12) results in a linear
equation at each pressure grid point, coupling its value to its neighbours. This sparse
system of simultaneous equations can be written as a matrix equation of the form
\( A\vec{x} = \vec{b} \), where \( A \) is a matrix of integer pressure coefficients, \( \vec{x} \) is a vector of unknown
pressures, and \( \vec{b} \) corresponds to the right hand side of Equation (3.13). Conveniently,
this system has the properties of being symmetric and positive definite. A large num­
ber of standard techniques can be used to solve systems of this type. For example,
Foster and Metaxas [FM96] used successive overrelaxation (SOR) but this is certainly

Figure 3.2: The left image shows the locations of horizontal velocity samples, vertical
velocity samples, and pressure samples. The right image visualizes the same velocity
field in a more natural way.
not optimal in terms of convergence. Later, Stam [Sta99] proposed using Fast Fourier Transforms (FFTs) and, although very fast, it is only applicable to periodic domains with no internal boundaries. The most common method used by modern simulators is the incomplete Cholesky conjugate gradient algorithm [She94]. Although this iterative method has excellent convergence properties, solving the pressure Poisson equation is still, by far, the most computationally expensive part of MAC-based solvers, requiring approximately 50% of the run-time in the solvers we have implemented.

3.2.4 Boundary Conditions

In order to facilitate application of the boundary conditions defined in Section 3.1, grid cells need to be categorized at the beginning of each time step. Each cell can either be a liquid cell, an air cell, or a solid cell. Voxelizing boundaries in this way requires them to coincide exactly with the edges of computational cells which greatly simplifies the enforcement of boundary conditions. However, this is a rather poor approximation for most surfaces, especially on coarse grids, and a better approach for solid obstacles will be presented in the next chapter.

Although only those cells marked as liquid need to be updated each time step, it is still necessary to set physically meaningful velocity and pressure values in all cells adjacent to liquid cells since these may be accessed by the stencils used in finite difference approximations, interpolation operators, etc. The appropriate values can be inferred from the conditions given by Equations (3.4) and (3.5). For example, in the context of the Poisson solver, Equation (3.4) is respected by requiring that the normal pressure gradient equals zero at edges that lie between solid and liquid cells, and Equation (3.5) is satisfied by setting the pressure equal to zero in air cells.

In cases where the fluid does not fill the entire computational domain, it is necessary to track the location of the liquid-air interface as it moves around. By tracking the location of this moving boundary, we can determine which cells are air and which are liquid and thereby apply boundary conditions at the appropriate locations.

The original MAC method uses massless marker particles to represent the liquid. These are advected based on the fluid velocity locally interpolated at their positions. For example, a particle located at position \( \vec{x} \) can be updated using Euler integration according to the equation \( \vec{x}^{(n+1)} = \vec{x}^{(n)} + \Delta t \vec{u} \left( \vec{x}^{(n)} \right) \). Liquid cells can then be identified
as those cells that contain at least one particle. Figure 3.1 shows an example that uses marker particles for tracking and rendering.

An alternative to marker particles that has become very popular in recent years is to capture the liquid-air interface as the zero level set of a dynamic signed distance function \( \phi \) that is sampled on the simulation grid [FF01]. This provides a smoother representation of the liquid surface that gracefully handles extreme deformations and topological changes. In order to advance the liquid surface based on the fluid velocity field, the discrete \( \phi \) samples need to be updated according to the advection equation

\[
\frac{\partial \phi}{\partial t} = -\bar{u}^{(n+1)} \cdot \nabla \phi.
\]

The grid cells can then be classified based on the sign of \( \phi \) at each cell. The downside of the implicit surface representation is that noticeable volume loss occurs due to numerical dissipation when solving the advection equation. This problem can largely be overcome however by using the particle level set method [EFFM02] which augments the signed distance field with particles in order to correct undersampling errors in the position of the zero level set. Examples of liquids simulated and rendered using level sets can be seen in Figures 5.4, 5.6 and 5.7.

### 3.2.5 Semi-Lagrangian Advection

In Section 3.2.3, we discussed using finite differencing for the spatial derivatives of the advection term \(-\bar{u} \cdot \nabla \bar{u}\) of Equation (3.9). However, when used in conjunction with explicit time integration, this discretization imposes a severe restriction on the size of the time step that can be used to advance the simulation. If too large a time step is chosen then the velocity values will oscillate and the simulation will become unstable. In order to guarantee stability, the time step must adhere to the Courant-Friedrichs-Lewy (CFL) condition which limits the maximum allowable time step to:

\[
\Delta t = \frac{\Delta x}{\max(|\bar{u}|)}
\]  
(3.15)

where \( \Delta x \) is the spacing between grid points. This condition says that if a quantity is transported over a grid, it must be integrated using a time step small enough to ensure that no grid points are skipped. Notice that no time step restriction is associated with the pressure derivatives since Chorin's projection method computes the pressure field implicitly to satisfy the governing equations.
Figure 3.3: A field $q$ is sampled at the centre points of grid cells. To compute $q^{(n+1)}(\bar{x})$, we trace backwards through the velocity field and interpolate $q^{(n)}$ at the point at which we end up, $\bar{x} - \Delta t \bar{u}(\bar{x})$.

In order to avoid any unnecessary computation, we would generally like to choose as large a time step as possible. In [Sta99], Stam introduced an unconditionally stable technique known as semi-Lagrangian integration for computing the advection term. This allows any time step to be used without regard for the CFL condition (though obviously accuracy is dependent on the time step). To see how this works, consider a scalar field $q$ that we wish to advect over a time step $\Delta t$ by the velocity field $\bar{u}$. The advection equation we wish to solve is of the form $\partial q / \partial t = -\bar{u} \cdot \nabla q$. Equation (3.9) can be rewritten in this form by splitting it into the two equations $\partial \bar{u}^* / \partial t = -\bar{u}^{(n)} \cdot \nabla \bar{u}^{(n)}$ and $\partial \bar{u}^* / \partial t = \bar{u}^* + g$. The first equation describes the self-advection of the velocity field. In two dimensions, this requires the solution to two separate advection equations in which $q$ corresponds to either the horizontal or vertical component of $\bar{u}$.

According to the advection equation, the values of $q$ will remain unchanged as they are transported to new locations by the fluid’s velocity. So considering a point $\bar{x}^{(n)}$, which is advected to $\bar{x}^{(n+1)}$ after time $\Delta t$, this statement can be written $q^{(n+1)}(\bar{x}^{(n+1)}) = q^{(n)}(\bar{x}^{(n)})$. Taking $\bar{x}^{(n+1)}$ as a fixed grid point at which the field $q$ is sampled, we can determine the new value of $q$ at this point by determining the old value of $q$ at $\bar{x}^{(n)}$. Of course, since $\bar{x}^{(n+1)} \approx \bar{x}^{(n)} + \Delta t \bar{u} (\bar{x}^{(n)})$, we can reverse this and take an Euler step backwards in time to compute $\bar{x}^{(n)}$. Evaluating $q^{(n)}$ at $x^{(n)}$ gives the new value of $q$ at the grid point. This can be written as:

$$q^{(n+1)}(\bar{x}^{(n+1)}) = q^{(n)}(\bar{x}^{(n+1)} - \Delta t \bar{u}(\bar{x}^{(n+1)}))$$

(3.16)
and is illustrated in Figure 3.3. Note that, since $x^{(n)}$ will not typically coincide with a grid point, interpolation is necessary to evaluate $q^{(n)}$. Using linear interpolation for this purpose ensures stability since taking convex combinations of quantities defined at grid points guarantees that the largest value of the new field will never be larger than the largest value of the previous field no matter what time step is used:

$$\max(|q^{(n+1)}|) \leq \max(|q^{(n)}|)$$  \hspace{1cm} (3.17)

However, this also has the undesirable effect of smoothing out the advected field variables. This can be alleviated to some extent by using higher order interpolation schemes such as monotonic cubic interpolation [FSJ01], which also adheres to Equation (3.17). Although this provides a sharper reconstruction of the advected function, numerical dissipation is inherent in the semi-Lagrangian method and cannot be defeated entirely.

Finally, for non-constant velocity fields, forward Euler integration can be very inaccurate and unstable. Therefore, it is usually desirable to use more accurate integration methods to better preserve rotational motion in the flow field. A good alternative is the midpoint method which can be implemented by replacing the $x^{(n+1)} - \Delta t \bar{u}$ term in Equation (3.16) with $x^{(n+1)} - \Delta t \bar{u} \left( x^{(n+1)} - \frac{1}{2} \Delta t \bar{u} \left( x^{(n+1)} \right) \right)$.

### 3.3 Smoothed Particle Hydrodynamics

#### 3.3.1 Overview

Unlike the MAC method presented in the previous section, the SPH algorithm is normally used to compute solutions to the compressible Navier-Stokes equations. This requires a spatially variant density field and an equation of state to relate density and pressure. Furthermore, the viscous term that was dropped for the MAC method is now included since it provides a convenient way to artificially dampen velocities in a physically plausible manner. In our experience, this is critical to achieve stable, noise-free simulations. Recall that grid-based advection schemes such as semi-Lagrangian advection perform operations that undesirably smooth out high frequency details in the velocity field. Instead, SPH performs advection in Lagrangian coordinates which does not dissipate any energy. However, this tends to create problems such as small
scale vibrations in the particle positions. For this reason, adding a small amount of artificial viscosity to help keep nearby particles moving together is necessary to achieve realistic fluid motion.

While the MAC method determines how fluid velocities at fixed grid points vary over time, in SPH, the velocity field is defined by the velocities of a collection of mobile particles. Each particle represents the fluid in its immediate vicinity and interacts with neighbouring particles. These interaction forces are weighted by kernel functions and are carefully designed to match the pressure, viscosity and body forces of the Navier-Stokes equations.
3.3.2 Smoothing Kernels

In the SPH formalism, the continuous velocity, density and pressure fields are sampled at discrete points (the particle positions) which are distributed throughout the fluid. In the same way that the MAC method uses interpolation between grid points to evaluate field variables at arbitrary points within the fluid, these quantities (and their derivatives) can also be evaluated using interpolation between disordered sample points. This is done by blending the values of nearby particles in a discrete sum. The contributions of each particle are weighted by their volume \( \left( \frac{m}{\rho} \right) \), and their distance from the point of interest using a normalized, radially symmetric kernel function, e.g., a Gaussian. This weighting function is designed to have a finite support (i.e., it evaluates to zero beyond a certain distance) to avoid evaluating \( O(n^2) \) interacting pairs in a collection of \( n \) particles. The equations to estimate the value, gradient and Laplacian of a function \( q \) at a point \( \vec{x} \) are:

\[
q(\vec{x}) = \sum_j q_j \frac{m_j}{\rho_j} W(\vec{x} - \vec{x}_j)
\]

(3.18)

\[
\nabla q(\vec{x}) = \sum_j q_j \frac{m_j}{\rho_j} \nabla W(\vec{x} - \vec{x}_j)
\]

(3.19)

and

\[
\nabla^2 q(\vec{x}) = \sum_j q_j \frac{m_j}{\rho_j} \nabla^2 W(\vec{x} - \vec{x}_j)
\]

(3.20)

where \( j \) iterates over all particles, \( x_j \) is the position of particle \( j \), \( m_j \) its mass, \( \rho_j \) its density, \( q_j \) the value of \( q \) at \( x_j \), and \( W \) is the smoothing kernel.

Our implementation of SPH follows the standard algorithm of Müller et al. [MCG03] where the mass of each particle is set to the same constant, but the density varies and needs to be evaluated each time step. The equation to perform this evaluation is derived by substituting \( \rho \) for \( q \) in Equation (3.18):

\[
\rho(\vec{x}) = \sum_j m_j W(\vec{x} - \vec{x}_j)
\]

(3.21)

Notice that the mass of each particle is not concentrated at a single point. Instead it is spatially distributed or “smoothed” over a circular area in 2D, or a spherical
volume in 3D – since the kernel function is normalized, the integral over space of the particle mass times the kernel function equals the particle mass. By summing the mass contributions from all particles at a point (most of which will be zero due to the finite kernel support), the density at that point can be evaluated.

3.3.3 Artificial Compressibility

One nice property of the SPH method is that, since each particle has a constant mass and there are a fixed number of particles, conservation of mass is automatically guaranteed. This means that Equation (3.2) doesn’t need to be explicitly solved and the computationally expensive pressure Poisson equation is avoided. However, this does not mean that conservation of volume is achieved. In fact, the biggest difference between the MAC and SPH methods is that, in the MAC method, the idealization of incompressibility is made and the exact pressure field which satisfies this constraint is computed whereas, in the SPH method, this constraint is relaxed and the fluid is allowed to compress slightly. Here the pressure field is instead inferred by applying Equation (3.3) to the density field – the greater the local density, the greater the pressure, etc. Although there has been some debate over the best choice for the equation of state, Becker and Teschner [BT07] recently demonstrated that Tait’s equation gives better results for modelling water than the ideal gas equation that was popular in earlier work. Tait’s equation is defined as follows:

$$p = \kappa \left( \left( \frac{\rho}{\rho_0} \right)^7 - 1 \right)$$

(3.22)

where \(\rho_0\) is the rest density and the coefficient \(\kappa\) is a constant related to the speed of propagation of pressure waves through the fluid, i.e., the sound speed. Note that the incompressibility condition, Equation (3.7), is equivalent to assuming an infinite sound speed. This can lead to slow convergence even when fast iterative solvers such as conjugate gradient methods are used. The SPH method instead assumes a finite sound speed which can usually approximate incompressibility to a high degree. However, in practice, to maintain stability while avoiding an overly restrictive time step (given by a CFL condition similar to Equation (3.15)), this artificial sound speed has to be chosen conservatively and compressibility effects often result which make animated
liquids appear “bouncy”, an undesirable side-effect when simulating approximately incompressible liquids such as water, and probably the biggest disadvantage of particle-based simulators.

### 3.3.4 Lagrangian Discretization

Equations (3.18)–(3.20) can now be applied to Equation (3.1) in order to determine the individual accelerations that affect the velocity of each particle $i$.

The first term on the right hand side, $-\mathbf{u} \cdot \nabla \mathbf{u}$, describes the acceleration at a fixed point in space due to self-advection. This accounts for the fact that the very movement of the fluid in the velocity field will cause the velocity field itself to change. For example, consider the velocity $\mathbf{u}(\mathbf{x}, t^n)$ at a fixed point in space $\mathbf{x}$ at time $t^n$. In general, $\mathbf{u}(\mathbf{x}, t^{n+1})$ will be different than $\mathbf{u}(\mathbf{x}, t^n)$ since the fluid located at $\mathbf{x}$ at $t^n$ will be replaced by different fluid at $t^{n+1}$ due to advection. This Eulerian change in velocity is described by the non-linear self-advection term and is traditionally the most problematic step of the MAC method due to the intrinsic dissipation that occurs when computing its solution on a grid. Fortunately, in Lagrangian methods such as SPH, this term can be discarded entirely since we are now tracking fluid accelerations at moving, instead of fixed, points. Therefore, we can handle advection of the velocity field without incurring any dissipation by simply advancing the particle positions along with the fluid’s velocity.

The next term in Equation (3.1) is the viscous term, $\frac{\mu}{\rho} \nabla \cdot (\nabla \mathbf{u})$, for which an expression can be derived by substitution into Equation (3.20). Note however that the resulting expression needs to be symmetrized in some way so that each pair of particles is accelerated with equal and opposite forces, thereby conserving momentum. The natural way to do this for viscous forces is to use velocity differences instead of absolute velocities:

$$\frac{\mu}{\rho} \sum_j (\mathbf{u}_j - \mathbf{u}_i) \frac{m_j}{\rho_j} \nabla^2 W (\mathbf{x}_i - \mathbf{x}_j)$$

(3.23)

where the viscosity is assumed to be spatially invariant so that the original term becomes $\frac{\mu}{\rho} \nabla^2 \mathbf{u}$.

Similarly, the pressure term, $-\frac{1}{\rho} \nabla p$, can be converted to the SPH setting by substitution into Equation (3.19). Again, the resulting expression produces asymmetric
forces since the pressures of interacting particles will differ in general. Different ways to symmetrize this term have been proposed in the literature, but we simply use the average of the pressures as in [MCG03]:

\[
-\frac{1}{\rho_i} \sum_j \left( \frac{p_i + p_j}{2} \right) \frac{m_j}{\rho_j} \nabla W (\vec{x}_i - \vec{x}_j)
\]  

(3.24)

The final term, \(\vec{g}\), is a constant acceleration vector that acts throughout the body of the fluid (such as gravity) and therefore affects each particle independently.

### 3.3.5 Simulation Algorithm

To summarize the algorithm, each particle has a position, velocity and constant mass. At the beginning of a time step, we first calculate the density of every particle, \(\rho_i = \rho(\vec{x}_i)\), by evaluating Equation (3.21). An equation of state, such as Equation (3.22), can now be applied to compute each particle’s pressure from its density, \(p_i = f(\rho_i)\). These pressure values measure how densely particles are packed together. Next, the viscosity and pressure gradient accelerations acting between all pairs of interacting particles are computed\(^2\) by evaluating expressions (3.23) and (3.24). The viscosity terms resist changes in velocity, and the pressure gradient terms attempt to bring the fluid back to a state of constant density, i.e., a state in which all particles are evenly spaced. These terms are added to the body acceleration \(\vec{g}\) to give the total acceleration of each particle, and collision detection and response are performed to prevent particles from interpenetrating scene geometry. Note that, unlike the MAC method which handles collisions implicitly through boundary conditions, explicit collision detection must be performed on a per-particle basis in SPH. Finally, the position and velocity of each particle can be updated by integrating them forward in time using, for example, Euler’s method.

Although [MCG03] suggests performing collision response by applying impulses to particles, we achieve more stable results by preventing abrupt changes in velocities and instead performing “soft” collision response using the penalty method. In this approach, when a particle intersects an object, we update its acceleration with a delay.

\(^2\)Since these acceleration terms are symmetric by design, they only need to be computed once per pair of particles.
force directed along the collision normal, with magnitude proportional to the penetration depth, and add damping proportional to the particle’s relative normal velocity. Furthermore, for the purposes of rendering, we use particle positions that have been projected to the outside of collision objects to avoid artifacts.

We have also experimented with a novel approach to improve the accuracy of SPH in the vicinity of boundaries. As mentioned earlier, the smoothing kernels are required to be normalized functions with a finite radius $h$, beyond which they evaluate to zero. If a particle is closer than distance $h$ to a boundary then the smoothing kernel will intersect the boundary and provide an underestimate of the correct smoothing weight. This causes particles to clump together around boundaries instead of maintaining even spacing throughout the domain. Inspired by bilateral filtering techniques from image processing, we can fix this by re-normalizing the kernel to take into account the intersecting boundary geometry. This is done by numerically integrating the kernel function over a cube of side $h$, centered on the particle position. By setting those points that lie inside boundaries to zero, we can compute a new normalization scale factor that takes boundaries into account. Of course, it is impractical to perform this integration every time step, for every particle, so instead we precompute the kernel normalization constants at the grid points of a uniform grid and interpolate these values to the location of a particle whenever a kernel needs to be evaluated. Currently, this algorithm only works for static obstacles, but it does improve accuracy and efficiency by keeping particles equidistant, even near boundaries. In practice however, this modification adds a lot of complexity to the SPH algorithm without significantly improving visual quality.

### 3.3.6 Optimizations

To avoid a quadratic number of neighbour queries, it is necessary to use a spatial data structure to accelerate the determination of which particles are in the vicinity of others. Since all of the smoothing kernels we use have a finite radius $h$, we can reduce the search to linear complexity in the number of particles by mapping them to a uniform grid of spacing $h$ at the beginning of each time step. The full set of neighbours of a particle can then be determined by only considering the particles in it’s own cell and adjacent cells. Furthermore, similar to Teschner et al. [THM+03], we store the cells in a hash table keyed by the cell’s coordinates. This reduces memory requirements.
for large domains and frees the fluid to move anywhere in space, rather than being bound by the limits of a fixed grid. Using this approach for the simulation shown in Figure 3.4, approximately 45% of the total per-frame computation time is spent finding neighbouring pairs of particles, 40% is spent processing these pairs (computing densities and forces), 7% is spent populating the hash table, and the remaining time is spent on collision detection and time integration. To accelerate collision detection between particles and scene geometry, we also map all object triangles to a uniform grid using spatial hashing. For each particle, this grid can then be queried to quickly determine a small subset of triangles with which collision testing needs to be performed.
Chapter 4

Weakly Compressible, Heightfield Liquid Simulation

4.1 Introduction

In the previous chapter, we provided an overview of numerical methods commonly used to solve the Navier-Stokes equations at non-interactive rates. In this chapter, we employ a mixture of these ideas to achieve faster and more realistic heightfield animation of liquids than previous approaches. We base our method on the shallow water equations. Although a mainstay in the field of computational fluid dynamics, these equations are rarely used in computer graphics due to their computational expense. The main exception is the work by Layton and van de Panne [LvdP02] which proposed solving the equations using an iterative, implicit integration scheme.

Generally, interactive heightfield liquid animation is achieved by solving the simpler 2D wave equation. This equation can easily be derived from the shallow water equations (see [KM90] for example), and is written:

\[ \frac{\partial^2 h}{\partial t^2} = c^2 \nabla^2 h \] (4.1)

where \( h \) is the height of the liquid surface at a point and \( c \) is the speed at which waves propagate across the surface. Notice that \( c \) is constant which means that the dispersive relationship between wavelength and wave frequency cannot be modelled
with this approach. The wave equation can be solved in a straightforward and efficient manner using Eulerian finite differences and explicit time discretization [NN05]. Its main drawback however is that it is only valid for flows with small velocity variations in space since it ignores the advection of fluid – the nonlinear advection terms are dropped in the simplification of the shallow water equations allowing for linearization. This limits the effects that can be modelled and the realism that can be achieved. By instead solving the full shallow water equations, we can handle fast moving fluid, generation of realistic vortices and wakes, and accurate interaction with moving objects.

In this chapter, we present a novel, density-based solver for the shallow water equations that uses a fast and compact Lagrangian advection scheme, and a non-iterative, Eulerian pressure solver based on artificial compressibility. Although our scheme sacrifices accuracy and unconditional stability for speed, we show that our algorithm is sufficiently stable and fast enough for real-time applications. Furthermore, the method we present is easy to understand and implement, has very low memory requirements compared to iterative and particle-based solution methods, and has the advantage of evolving velocity and pressure fields in time which can be used for many purposes, including accurately simulating buoyant objects interacting with the fluid and advecting objects such as leaves or foam particles over the liquid surface.

4.2 Governing Equations

The shallow water equations are a simplified version of the 3D Navier-Stokes equations, suitable for animating heightfield-based bodies of liquid such as ponds, lakes or oceans. There are two equations. The first describes conservation of mass and the second describes conservation of momentum:

\[
\frac{\partial h}{\partial t} = -\bar{u} \cdot \nabla h - h \nabla \cdot \bar{u} \tag{4.2}
\]

\[
\frac{\partial \bar{u}}{\partial t} = -\bar{u} \cdot \nabla \bar{u} - g \nabla h \tag{4.3}
\]

Here, \( h \) is the height of the water above ground level, \( \bar{u} = (u, v) \) is the velocity and \( g \) is the acceleration due to gravity. For the moment, we will assume that the ground has flat topography, although supporting variable height ground is trivial.
We can draw an analogy between the form of these equations and the inviscid, 2D Navier-Stokes equations with no body forces:

\[
\frac{\partial \rho}{\partial t} = -\bar{\mathbf{u}} \cdot \nabla \rho - \frac{1}{\rho} \nabla \cdot \bar{\mathbf{u}} \quad (4.4)
\]

\[
\frac{\partial \bar{\mathbf{u}}}{\partial t} = -\bar{\mathbf{u}} \cdot \nabla \bar{\mathbf{u}} - \frac{1}{\rho} \nabla p \quad (4.5)
\]

where \( \rho \) is density and \( p \) is pressure. We consider the fluid to be slightly compressible and introduce an artificial equation of state (EOS) to close the system:

\[
p = c^2 \rho \quad (4.6)
\]

Here, \( c \) is the speed at which pressure waves propagate through the liquid. The pressure gradient in equation (4.5), \(-\nabla p/\rho\), now becomes a density gradient, \((-c^2 \nabla \rho)/\rho\). In this form, all of the derivatives in both pairs of equations match and solvers developed for one pair of equations can generally be applied to the other pair with little modification.

In our solver, we further simplify Equations (4.2) and (4.4) by assuming that the fluid is nearly incompressible such that the divergence term, \( \nabla \cdot \bar{\mathbf{u}} \), can be treated as negligible. Hence, we set the right hand side of Equations (4.2) and (4.4) to zero. An additional benefit of this is that, provided a conservative scheme is used for advection, the total height/mass will be constant. This same approximation is used in the SPH algorithm and, in practice, we find it difficult to tell the difference between results that include and exclude this term. Of course, this term can easily be included for greater accuracy if desired and its inclusion is important for simulating compressible liquids. However, in our simulations, the magnitude of this term is much smaller than the other terms, thus justifying its removal. In this form, \( h \) corresponds to \( \rho \) and \( g \) corresponds to \( c^2/\rho \). By substitution, we can further see that \( c^2 = \sqrt{gh} \). In other words, the speed of propagation of pressure waves corresponds to the speed of surface waves and varies as a function of gravity and the depth of the water.

Our method solves Equations (4.4) and (4.5) and then computes \( h \) by linearly scaling \( \rho \). This relation comes from the assumption of hydrostatic pressure, which the shallow water equations are built on. For example, if \( \rho \) is sampled on a grid with cell size \( \Delta x \) then \( h = \rho/(\rho_0 \Delta x^2) \), where \( \rho_0 \) is the constant density of the liquid (1,000 kg/m\(^3\) for
water). This corresponds to the fact that in regions of high density (compression), the liquid height will be raised due to mass conservation.

4.3 Solver Overview

We consider an algorithm to solve the coupled equations, (4.4) and (4.5). Traditionally, these equations would be solved on a grid by replacing Equation (4.4) with the incompressibility condition, $\nabla \cdot \vec{u} = 0$, and using the projection method to compute a mass-conserving pressure field. This would require solving a Poisson equation, which is generally the bottleneck in most fluid simulators, accounting for as much as 50% of the total computational time in our experience and incurring a large memory overhead when solvers such as the conjugate gradient method are used. Instead, we dispense with the Poisson equation and use an artificial equation of state relating fluid density to pressure, similar to SPH. As discussed in Section 3.3.3, one big disadvantage of using SPH to animate liquids is that they often appear unrealistically compressible. This is difficult to avoid and is a result of the fluid having to constantly fight against density changes induced by the downwards pull of gravity. Since the shallow water equations assume a depth-averaged velocity field, there is no explicit gravitational field acting on the fluid and therefore, even though we are no longer solving for a truly incompressible fluid, we believe that compressibility in the context of shallow water simulation does not adversely affect visual quality, especially considering our method still achieves global mass and volume conservation. In fact, we believe that a small degree of compressibility actually produces more visually appealing heightfields by allowing pressure waves to persist longer than they would in an incompressible fluid. Surprisingly, compressible flow simulation on grids has received little attention from the graphics community. In fact, the only work we are aware of is that of Yngve et al. [YOH00] which solved the compressible Navier-Stokes equations to animate explosions at non-interactive rates.

The overall structure of our fluid simulator is very simple. At the beginning of each iteration of the time-stepping loop, the velocities and densities of the fluid may be modified, for example, to simulate the effect of wind or rain on the liquid surface.

\[^1\] Initially, we attempted to solve the shallow water equations in this way, but found that the infinite sound speed led to pressure waves that damped rapidly, resulting in a severe lack of surface detail. This issue will be revisited in Section 4.7.
Figure 4.1: Shallow water simulation on a 100x100 grid. On the left, the user interactively applies forces to the fluid to form a whirlpool. The corresponding density and velocity fields are shown on the right.

Next, the advection step transports $\vec{u}$ and $\rho$ along the velocity field from time $t^n$ to time $t^{n+1}$ by solving $\partial \rho / \partial t = -\vec{u} \cdot \nabla \rho$ and $\partial \vec{u}^e / \partial t = -\vec{u} \cdot \nabla \vec{u}$. Note that operator splitting has been used here (see Section 3.2.5) so that different techniques may be employed to treat the advection and pressure terms of Equation (4.5). Finally, in the pressure step, we solve $\partial \vec{u} / \partial t = \vec{u}' - (c^2 / \rho) \nabla \rho$. Here, the pressure gradient is calculated by scaling the density gradient, and this is then subtracted from the velocity field to give $\vec{u}$ at time $t^{n+1}$. At the end of the update, the new heightfield can be calculated by simply scaling the density field as described in Section 4.2.

Since we use an equation of state to model pressure, which is solved for explicitly, our solver is only conditionally stable and must adhere to a CFL stability condition which says that sound waves must not travel further than a cell length $\Delta x$ in one time step. In other words, the maximum time step (assuming subsonic flow, i.e., $\max (|\vec{u}|) \ll c$) is given by

$$\Delta t = \frac{\Delta x}{c}$$

(4.7)

This is the downside of computing pressure explicitly, as opposed to implicitly (for instance, using the projection method). It may seem as though this time step restriction will result in very inefficient simulation. For example, the speed of sound in water is approximately 1,500 m/s. This means that if we want to use a grid at a resolution of 10 cm, the time step would be limited to about 0.005 s. In practice however, we can achieve visually plausible results using a slightly reduced sound speed and update...
the fluid in a single time step equal to the full elapsed time per frame in a real-time simulator running at approximately 60 FPS. We note however that, for high grid resolutions or fast moving fluid, adaptive sub-stepping may be necessary to respect the CFL condition and maintain stability.

In the next section we discuss the choice of grid structure we use to arrange state variables. This is followed by a description of the advection and pressure steps that are invoked each iteration to update the fluid’s state.

## 4.4 Grid Configuration

The state of our fluid is defined by its 2D velocity and density fields as shown in Figure 4.1. These fields need to be sampled on uniform grids. As discussed in Section 3.2.3, the standard approach used in computer graphics is a staggered MAC grid, where horizontal/vertical components of velocity are located at the centre of vertical/horizontal cell faces respectively, and density/pressure variables are located at cell centres. This staggered configuration avoids the classic “checkerboard” problem [Min96] that arises from decoupling between the pressure and velocity fields since the location of pressure gradients naturally coincide with velocity variables, and the location of velocity gradients coincide with pressure variables.

In our solver, we abandon the staggered grid in favor of a collocated arrangement where all variables are located at the centre of grid cells (see Figure 4.2). This configuration has been shown to suffer from instabilities and spurious oscillations in the
Figure 4.3: Using central differences on collocated grids can lead to checkerboard instabilities, examples of which can be seen in the density field on the left, and the heightfield on the right.

pressure field. Specifically, central difference approximations to pressure and velocity gradients will not use adjacent nodes in the grid and, as a consequence, pressure fields can be calculated which satisfy the discrete equations but do not satisfy the continuous equations. This results in non-physical pressure values which manifest themselves as distracting artifacts on the liquid surface as shown in Figure 4.3.

We expect however that large computational savings can be made if the collocated scheme can be stabilized to avoid decoupling between the velocity and pressure fields, even if this means compromising physical accuracy. Such a tradeoff may be unacceptable in other scientific fields, but our goal is to produce plausibly realistic animation as fast as possible. Furthermore, the collocated configuration is less geometrically complex and easier to understand, implement and debug, which makes it attractive from a practical standpoint.

4.5 Advection Step

Given that the pressure gradient term in Equation (4.5) will be computed explicitly, the main computational burden now rests on advecting the \( \bar{u} \) and \( \rho \) fields. Normally the semi-Lagrangian advection scheme is used in fluid animation because it is fast and remains stable irrespective of the time step size. However, this scheme has a number of drawbacks. Firstly, it introduces dissipative error due to interpolating the new velocity field from the old one. This tends to smear the fine details in the flow, which
requires an additional technique such as vorticity confinement [FSJ01] to counteract. In our experience, the semi-Lagrangian method performs very well for velocity advection. However, the artificial damping incurred by interpolation causes inaccuracy when used to advect densities, and we found that the divergence term we omitted from Equation (4.4) was necessary to get this to work. Another problem is that the semi-Lagrangian method does not work well for compressible flow in certain situations. For example, consider a velocity field sampled on a grid which is zero at all grid points except one. Since the semi-Lagrangian method works by back-tracing particles and interpolating, this field will either remain at rest or dissipate unrealistically, depending on the time step. Since density is not advected correctly in this case, an explicit pressure scheme will fail. Chorin’s projection method can solve this problem by enforcing zero divergence velocities in all grid cells, causing problematic velocities to spread out realistically. However, for the reasons discussed earlier, we wish to avoid solving for an incompressible fluid. A final problem with the semi-Lagrangian method is that it is not conservative, meaning that mass loss will occur when this scheme is used to advect density.

![Figure 4.4](image-url)

Figure 4.4: All cell variables are advected together by forward-tracing $N \times N$ packets, where $N = 2$ in this example (left). After advection, the quantities carried by the packets are accumulated on a new grid (right).

To overcome these problems, we propose a new conservative advection scheme which is sufficiently stable for our requirements and, similar to the semi-Lagrangian method, avoids an advective time-step restriction due to cell size. Our approach also combines the Lagrangian and Eulerian methodologies by using particles to accurately advect quantities stored on a grid without requiring any explicit discretization of the advec-
tion terms. However, while the semi-Lagrangian method works by tracing the trajectories of particles upstream (against the flow), our scheme works by tracing particles downstream (with the flow).

More specifically, with reference to Figure 4.4, let us consider the advection of a field variable $q$ from a grid cell of area $(\Delta x)^2$, which has initial value $q_0$. According to the advection equation, $\partial q/\partial t = -\vec{u} \cdot \nabla q$, the function $q$ is constant along trajectories of particles moving with the fluid velocity. Hence, we partition the cell into an $N \times N$ subgrid ($N$ determines the accuracy of the advection scheme) and, for each subcell, we trace a packet of area $(\Delta x/N)^2$ from the centre of the subcell, forward along the velocity field, and deposit this packet at its final location. Each packet carries a fraction of the initial quantity, equal to $q_0/(N \times N)$, and, for the purposes of advection, is treated as a particle which follows the fluid according to the local velocity vector. We integrate these particles over time using either Euler's method or the midpoint method (see Section 3.2.5 for details), and "bounce" them off of colliding obstacles to prevent the quantity (e.g., density) from clumping in regions adjacent to boundaries. To deposit the packet, we overlay it on top of the grid and compute the area of overlap with each of the cells it intersects. Initially these cells correspond to a field at time $t_{n+1}$ in which $q$ is zero everywhere. However, as each packet is overlaid on the grid, we update the overlapping cells by adding an area-weighted fraction of $q_0$ to each cell's $q$ accumulator variable. Consider, for example, the configuration shown on the right of Figure 4.4. In this case, a packet overlaps four cells. Cell $A$ will be updated by adding $x_0 y_0 q_0$ to the accumulator variable $q_A$. Similarly, $x_1 y_0 q_0$ will be added to $q_B$, $x_1 y_1 q_0$ to $q_C$, and $x_0 y_1 q_0$ to $q_D$. After all packets from all cells have been traced and apportioned, the accumulator variables correspond to the new $q$ field. In the case of density advection, it is clear that this scheme conserves mass – the total density before advection will equal the total after advection since the density is simply moved from one place to another.

The solver we have developed uses this advection scheme, not only for density, but also for velocity. This is very convenient in the context of collocation since all variables (density and the two components of velocity) can be traced together using the same packets. This saves considerable computation as compared to performing semi-Lagrangian advection separately for velocity. In fact, compared to a semi-Lagrangian scheme on a standard MAC grid, our scheme cuts down on the number of required operations by a factor of approximately three (for $N = 1$). While the staggered ar-
rangement requires advecting three separate fields independently (since density and each component of velocity all use different sample locations), our approach enables all three fields to be advected simultaneously.

Finally note that, although our advection scheme conserves mass exactly, it does not conserve momentum. This is because, when we perform accumulation of velocities, positive and negative velocities will be summed which will cancel each other out, resulting in momentum loss. Momentum loss occurs when using the semi-Lagrangian method for the same reason – interpolation between velocities causes cancellation between positive and negative values. In the interest of conserving momentum, an interesting direction for future work would be to use momentum as a primary variable instead of velocity, and to reconstruct the velocity field each step from the momentum and density fields.

4.6 Pressure Step

In this stage of the algorithm, the scaled density (pressure) gradient is subtracted from the velocity, as in Equation (4.5). The natural way to compute this gradient is with standard, second order accurate central difference approximations. For example, using the subscripts $i$ and $j$ to refer to discrete locations on a Cartesian grid with cell length $\Delta x$, the pressure gradients used to update the horizontal and vertical component of velocities are defined as follows:

$$\frac{\partial p}{\partial x} \approx \frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x}$$

$$\frac{\partial p}{\partial y} \approx \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta x}$$

However, as discussed in section 3.2, these approximations ignore information stored at grid point $(i, j)$. Because of this, it is generally unstable to use central differences in conjunction with a collocated configuration due to the checkerboard problem.

We tried a number of different solutions to fix this. One approach is to average the cell-centred velocities to cell faces, then subtract the pressure gradient from the face velocities and finally average these velocities back to the cell centres. This was successful in avoiding the stability issues. However, averaging the velocities back and forth causes significant smoothing of the velocity field (even more than the semi-Lagrangian
method) which produces unsatisfactory results. Note that, to reduce dissipation, it may be tempting to avoid averaging the entire velocity from cell faces to cell centres, and instead only average the change in velocity [GSLF05]. However, this change in velocity is equal to the pressure gradient and averaging this gradient from the faces to the centre of a cell is equivalent to performing a central difference approximation of the gradient in the first place.

Another technique we investigated to help suppress numerical instabilities is Rhie-Chow interpolation [RC83]. This scheme defines a way to interpolate velocities to cell faces such that the checkerboard instability is avoided. We did not pursue this approach in the end due to the interpolation scheme being considerably more expensive than linear interpolation. However, it can be shown that this approach is equivalent to adding a pressure smoothing term, the effect of which is to eliminate pressure oscillations. Motivated by this, we tried simply to smooth the pressure field (while using the central difference pressure gradient approximation) and found that this was sufficient to avoid the checkerboard problem. Conveniently, this smoothing can be performed as part of the advection step (described in the previous section) at negligible additional cost. This is done by simply scaling the size of each packet by a small amount so that it is blurred out over a slightly larger area. So, instead of an area of \((\Delta x/N)^2\), the packet now has an area of \((s\Delta x/N)^2\), where values of \(s\) greater than 1 result in smoothing. To compensate for this increase in packet size, the quantities carried by the packet need to be scaled down by \(1/s^2\). This smoothing does introduce noticeable artificial dissipation to the velocity field which is undesirable. However, given the performance benefits of using a collocated grid versus a staggered grid, we consider this tradeoff to be worthwhile.

4.7 Discussion

The main source of numerical dissipation in our advection scheme is interpolating the velocity field to the locations of packets. Similar to SPH, it would be possible to instead treat the complete collection of packets as a system of particles whose state persists over updates, and only use the grid to compute the acceleration of particles due to pressure. This would greatly reduce dissipation. However, an issue with this approach is that, since the compressible flow field will have regions of positive divergence ("sinks"),
and regions of negative divergence ("sources"), the particles will have a tendency to converge, creating inefficiency due to oversampling, or separate, leaving gaps in the flow. This could be solved by reseeding particles at regular time intervals to maintain an even spatial distribution. However, doing so requires interpolation which is precisely the problem we are trying to avoid. In effect, our advection scheme reseeds particles every update by distributing them uniformly within each cell. While this may cause excessive dissipation of velocities, an advantage of this scheme is that it avoids having to store the particles, while providing a good sampling of the entire simulation domain.

It is important to realize that our advection scheme is more prone to instability than semi-Lagrangian advection since Equation (3.17) does not hold. The reason for this is that multiple packets may be traced to the same physical space and accumulated. Ordinarily, this indicates compression of the fluid which is resolved, in the pressure step, by accelerating the fluid away from regions of compression. However, since the amount of compression that can occur is unlimited in our scheme, too much compression can lead to unrealistic results and instability, especially when too large a time step is used. The artificial packet smoothing technique introduced in the previous section alleviates this problem by letting the fluid "spread out" a small amount into the space surrounding a packet. However, this smoothing is applied uniformly to all packets which causes excessive damping. A better approach would be to limit the amount of an advected quantity each cell can accept and only spread out the excess after that limit is reached. This would provide a non-uniform filtering of the advected field which should improve the visual quality of the simulation without compromising stability. Another approach that can be used to reduce instability is to take an area-weighted average of the advected packet quantities instead of accumulating them. This would enforce Equation (3.17) but would no longer be conservative. In practice, we achieve satisfactory results using the accumulation procedure in combination with a small amount of packet smoothing ($s \approx 1.05$).

As mentioned earlier, an alternative approach to solving the shallow water equations is to assume constant density and solve for an incompressible velocity field. This assumes that pressure waves propagate at infinite speed however which tends to cause pressure variations to die out very quickly, resulting in overly smooth heightfields. We have developed an alternative approach which also assumes constant density, but replaces the hard constraint of incompressibility with an artificial compressibility method.
By making the velocity tend toward incompressibility based on a finite speed of sound $c$, we can achieve detailed propagation of waves while avoiding the computational expense of advecting a density field and using an iterative solution method (such as the conjugate gradient method) to compute the pressure field.

To explain this method, we begin by converting the density time derivative in the continuity equation, Equation (4.4), to a pressure derivative. This is done by substituting $\rho$ for $p/c^2$, in accordance with the equation of state, Equation (4.6). This gives:

$$\frac{1}{c^2} \frac{\partial p}{\partial t} = -\bar{u} \cdot \nabla \rho - \frac{1}{\rho} \nabla \cdot \bar{u}$$

(4.10)

Since the pressure field will converge towards a state of constant density we can simplify this as follows:

$$\frac{\partial p}{\partial t} = -c^2 \nabla \cdot \bar{u}$$

(4.11)

This can be understood by considering that a negative divergence indicates that more fluid is flowing out of a cell than flowing in and this should result in a drop in pressure. Similarly, a positive divergence signifies that more fluid is entering a cell than leaving and this should result in pressure being raised. Over a sufficient period of time, the time derivative will go to zero and the resulting pressure field will enforce zero divergence in the velocity field. This approach is similar to the pressure solver used in [FM96] except that we do not solve Equation (4.11) to convergence each time step, and we provide a different interpretation of the relaxation coefficient used to scale the divergence field. We have implemented this approach by modifying our 2D fluid solver which is based on the incompressible, MAC method described in Section 3.2. Instead of computing the pressure field from scratch by solving a Poisson equation, we update the pressure field each time step according to Equation (4.11). The density and corresponding height fields can then be inferred by linearly scaling the pressure field as before.

Interestingly, we also had success using this simple pressure approximation for simulating free surface flows, albeit with a more stringent time step restriction. In spite of this, by avoiding the need to advect densities or solve a costly pressure Poisson equation, we believe that this approach could be adapted to work at near real-time rates, for example, by introducing appropriate limiters as described above to suppress the instabilities that arise when using larger time steps. This could be used in combination with a shallow water simulator to animate transient effects such as splashing (where
compressibility is difficult to perceive) and we would like to investigate this as future work.

4.8 Performance

<table>
<thead>
<tr>
<th></th>
<th>25x25</th>
<th>50x50</th>
<th>100x100</th>
<th>200x200</th>
<th>400x400</th>
</tr>
</thead>
<tbody>
<tr>
<td>[OH95]</td>
<td>40605.0</td>
<td>9969.6</td>
<td>2554.6</td>
<td>622.1</td>
<td>151.9</td>
</tr>
<tr>
<td>Colloc./N=1</td>
<td>13066.4</td>
<td>2998.3</td>
<td>738.8</td>
<td>183.7</td>
<td>36.0</td>
</tr>
<tr>
<td>Eqn. (4.11)</td>
<td>11310.2</td>
<td>2573.2</td>
<td>596.3</td>
<td>137.4</td>
<td>28.8</td>
</tr>
<tr>
<td>Stag./Semi-Lag.</td>
<td>7154.5</td>
<td>1640.6</td>
<td>397.2</td>
<td>96.4</td>
<td>22.1</td>
</tr>
<tr>
<td>Colloc./N=2</td>
<td>3226.7</td>
<td>737.8</td>
<td>177.0</td>
<td>43.6</td>
<td>10.2</td>
</tr>
<tr>
<td>[Tes04]</td>
<td>4179.3</td>
<td>516.1</td>
<td>97.3</td>
<td>21.4</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 4.1: Performance comparison of different solvers for different grid resolutions. All values are in updates per second.

<table>
<thead>
<tr>
<th></th>
<th>25x25</th>
<th>50x50</th>
<th>100x100</th>
<th>200x200</th>
<th>400x400</th>
</tr>
</thead>
<tbody>
<tr>
<td>[OH95]</td>
<td>15.9</td>
<td>62.2</td>
<td>246.5</td>
<td>981.3</td>
<td>3915.6</td>
</tr>
<tr>
<td>Colloc./N=1</td>
<td>15.3</td>
<td>61.0</td>
<td>244.1</td>
<td>976.6</td>
<td>3906.3</td>
</tr>
<tr>
<td>Eqn. (4.11)</td>
<td>11.2</td>
<td>43.1</td>
<td>169.1</td>
<td>670.3</td>
<td>2668.8</td>
</tr>
<tr>
<td>Stag./Semi-Lag.</td>
<td>15.6</td>
<td>61.8</td>
<td>245.7</td>
<td>979.7</td>
<td>3912.5</td>
</tr>
<tr>
<td>Colloc./N=2</td>
<td>15.3</td>
<td>61.0</td>
<td>244.1</td>
<td>976.6</td>
<td>3906.3</td>
</tr>
<tr>
<td>[Tes04]</td>
<td>9.8</td>
<td>39.1</td>
<td>156.3</td>
<td>625.0</td>
<td>2500.0</td>
</tr>
</tbody>
</table>

Table 4.2: Memory requirements of different solvers for different grid resolutions. All values are in kilobytes.

We evaluated the speed of our solver for different grid resolutions on an Intel Xeon 3.6 GHz PC. The results when tracing 1 packet per cell (N=1) and 4 packets per cell (N=2) are shown in Table 4.1. For comparison with our solver, we also timed the performance of the 2D wave equation solvers described in [OH95] and [Tes04], the simplified pressure projection algorithm described by Equation (4.11) in Section 4.7, and a version of our solver using semi-Lagrangian advection on a staggered grid instead of a colocated grid.

As expected, the basic wave equation solver of [OH95] achieves the best performance since no advection is required. However, [Tes04] achieves the worst performance due
to the fact that it discretizes the Laplacian operator in Equation (4.1) using a 12x12 stencil instead of the 3x3 stencil used in [OH95].

As can be seen from Table 4.1, our shallow water solvers perform at real-time frame rates making the approach a practical alternative to wave equation solvers. Although about 3.5 times slower than [OH95], our solver still achieves very high update rates considering the amount of extra work that is required to perform advection. In fact, for a 100x100 grid, the advection step in our method takes 98% of the update time and the pressure step only takes 2%. By avoiding a pressure Poisson equation we have managed to reduce the percentage of time spent updating the pressure field from ~50% to about 2%, while still maintaining sufficient stability to take time steps large enough to update the fluid between frames in a single step.

Also note that, compared to incompressible fluid solvers which require solving a linear system over the entire domain, our method is very easy to parallelize. To do this, we subdivide the uniform grid into subgrids, where each subgrid is updated in parallel. When packets need to be transferred across subgrid boundaries, they can either be added to a list (each thread maintains its own list to avoid contention) which is later processed when safe to do so, or, locks can be used to protect cells near subgrid boundaries. So far, we have only been able to test the former algorithm on a dual-core machine. For large grid sizes (e.g., 250 × 250), we achieve speedups of about 1.8x. Although we have parallelized the entire simulation update routine, the parallel efficiency drops significantly for small grid sizes. We have not yet investigated the reason for this loss of efficiency but we believe that close to optimal parallel scaling can be extracted from the algorithm.

Figure 4.5: Variation in wave detail on grids of different resolution (from left to right: 25×25, 50×50, 100×100, 200×200, 400×400). An identical forcing function was applied to the fluid in all cases.

The memory requirements corresponding to the various simulations from Table 4.1 are presented in Table 4.2. None of our solvers require more memory than [OH95] and do not require additional memory as the number of the packets per cell is increased.
Finally, Figure 4.5 shows how the appearance of the simulated heightfield is affected by the choice of grid resolution. This choice is very dependent on the nature of the application. In general however, higher resolution grids will provide greater detail and may be necessary if subtle details such as capillary waves need to be modelled. Of course, it is redundant to use too fine a grid if the sampled functions we wish to describe can be captured on coarser grids. For example, calm or low frequency waves will not require as fine a grid since the smooth functions are easier to reconstruct by interpolation. In comparison, steeper waves will require more resolution to accurately capture discontinuities. Alternatively, numerical diffusion on grids may be reduced by using higher order advection schemes instead of semi-Lagrangian methods, but this comes at greater computational expense. As seen in Figure 4.5, when the grid resolution is too low, excessive numerical diffusion occurs and wave detail is lost. As the resolution is increased, more detail becomes evident. Notice however that there is little difference between the results for the 200×200 grid and the 400×400 since at this resolution the velocity and density functions are being accurately sampled. Given the apparent difficulty in choosing an appropriate grid resolution, the next section will discuss how our solvers can be made to dynamically adapt the grid resolution to the flow, based on a determination of how well the various functions are being sampled in different regions.

### 4.9 Adaptive Simulation

One of the main benefits of our compressible fluid solver is that the algorithm maps conveniently to adaptive and unstructured grids. It is traditionally very complex to implement incompressible solvers on non-uniform grids due to difficulties in making convergence of the Poisson solver competitive with uniform grids (see [LGF04] and [Pop03] for example). However, our scheme can be adapted easily to new grid types. All that is required is the ability to compute the overlap between advected packets and grid cells, and a consistent way to evaluate pressure gradients at the centres of cells.

In this section, we discuss how to modify our fluid simulation algorithm to work on dynamic, adaptively refined grids instead of a single uniform grid. Adaptive grids have the benefit of focusing computational effort where it is most appropriate, while minimizing the time spent updating "uninteresting" regions of the flow. In this way,
they can facilitate high resolution simulations of complicated fluid flows without placing gratuitous demands on computational resources.

4.9.1 Multiresolution Grids

Level of detail schemes in computer graphics often use quadtree/octree methods to recursively subdivide grid cells. In previous work, we implemented an incompressible fluid solver which operates on a quadtree data structure. Each leaf node corresponds to a grid cell of the computational domain and we used a standard staggered arrangement of variables where pressure is sampled at cell centres and normal velocities are sampled at edge midpoints. Extensive use of linear interpolation operators are required to perform semi-Lagrangian advection, advance marker particles, compute velocity/pressure gradients, and transfer variables to newly coarsened or refined cells. Unfortunately, compared to uniform grids, the efficiency of interpolation is greatly affected by the use of quadtrees, and is further complicated by the fact that velocity variables are sampled at cell edges. Another issue is that the standard discretization of the pressure Poisson equation results in a non-symmetric coefficient matrix since edges may border cells of differing size. This precludes the use of fast linear system solvers such as the preconditioned conjugate gradient method. We have experimented with the multigrid algorithm of Popinet et al. [Pop03] to solve the non-symmetric linear system and, although we were able to achieve a 300-fold speedup in benchmark problems compared
to an iterative Jacobi solver, the overall performance of the fluid simulator (shown in Figure 4.6) was not competitive with an equivalent simulator running on a fully-refined uniform grid.

Losasso et al. [LGF04] were the first to introduce octree-based liquid simulation to computer graphics at around the same time we developed our incompressible quadtree solver. They proposed solutions to the two main problems we experienced. Firstly, to avoid the complexity of interpolating staggered velocities between grid cells of different size, they instead sample velocities at the corners of cells. Although this substantially improves the performance of interpolation, the downside is that the Poisson solver still requires velocities to be defined on cell faces so these must be continually averaged back and forth between the cell faces and corners each time step. The incurred numerical dissipation produces results inferior to a uniform grid at the same resolution [IGLF06]. Secondly, to avoid dealing with a non-symmetric linear system, they propose an alternate discretization of the Poisson equation which uses perturbed pressure samples to produce a symmetric system. While this reduces the time spent solving for pressure to 25% of the total time, they do not report any performance results, so it is difficult to tell if their overall approach is faster than running the full resolution simulations on a uniform grid.

We propose using tiled uniform grids of varying resolution as an alternative to quadtrees. There are many benefits to this approach. Firstly, they perform significantly better in many respects. For example, interpolation operators are trivial to implement on uniform grids, refinement and coarsening reduce to simple averaging, and random queries can be performed in $O(1)$ time, as opposed to $O(\log n)$ for quadtrees. Secondly, for the same level of refinement, they are more memory and cache efficient. Quadtree representations require significant overhead to store pointers for navigating between related cells and the indirection and manipulation of these pointers reduces cache coherency. Finally, parallelization, implementation and debugging are all much easier on uniform grids. Although we have not developed a version of our compressible solver that uses adaptive quadtree refinement, we believe that, for performance critical applications, the use of multiresolution uniform grids would substantially outperform the use of quadtrees.
4.9.2 Pressure Gradient Discretization

As mentioned earlier, two modifications are required to adapt our solver to work on juxtaposed uniform grids of different resolution. The procedure to redistribute advected packets across the boundary between cells of different size is trivial. However, computing the pressure gradients is slightly more complex and will be considered in this section for the x direction terms. The gradients in the y direction are defined identically. Also note that, for simplicity, we restrict the maximum difference in levels of refinement between adjacent grids to one. In other words, if one grid has a cell length of $\Delta x$, then adjacent grids must have a cell length of either $\frac{1}{2}\Delta x$, $\Delta x$ or $2\Delta x$. This is not a limitation in practice since this restriction should always be enforced to provide smooth transitions between different levels of refinement.

There are three cases to consider when computing the pressure gradient at a grid cell which lies on the boundary between different grids. The first case is where both grids are the same resolution. In this case, all cells will be the same size and the gradient can be evaluated using the usual central difference approximation given by Equations (4.8) and (4.9). The other cases arise when the two grids are of different resolutions. These two cases are exemplified by the configuration shown in Figure 4.7.

We will first consider a discretization similar to the one presented by Losasso.
[LGF04], but adapted for collocated grids. Their approach is based on the fact that $O(\Delta x)$ perturbations in the locations of pressure samples still yield convergent approximations. For example, in their approach, they would approximate the pressure gradient at $p_D$ as the finite difference $(p_F - p_C)/(\frac{3}{4}\Delta x)$, where $\frac{3}{4}\Delta x$ is the distance between $p_{BC}$ and $p_F$. Notice that a number of approximations have been made in this discretization in order to produce a symmetric formulation of the Poisson equation. Firstly, $p_C$ is used instead of $p_{BC}$; and secondly, the resulting gradient is actually defined at the midpoint between $p_{BC}$ and $p_F$, but is considered to be defined at $p_D$. According to Losasso, the error introduced by these approximations is not significant because they still converge when solving the Poisson equation. However, since our artificially compressible fluid solver does not solve for an incompressible pressure field, the errors present in Losasso’s approximation of the pressure gradient show up as small, spurious waves which reflect off the boundaries between different resolution grids. These artifacts are unacceptable so a different discretization is necessary.

We can avoid all of the perturbative errors introduced by Losasso, by instead evaluating pressure gradients by fitting a parabolic curve, $y = ax^2 + bx + c$, through three adjacent pressure samples, and evaluating its gradient, $dy/dx = 2ax + b$, at the location of the middle sample. Given three points defining a parabola, $(x_1, y_1), (x_2, y_2)$ and $(x_3, y_3)$, the gradient at $x_2$ is given by:

$$\frac{dy}{dx}_{x_2} = \frac{2x_2}{x_3 - x_2} \left( \frac{y_3 - y_1}{x_3 - x_1} - \frac{y_2 - y_1}{x_2 - x_1} \right)$$

(4.12)

So, for example, if we apply this formula to the points $(x_1, p_{i-1,j}), (x_1 + \Delta x, p_{i,j})$ and $(x_1 + 2\Delta x, p_{i+1,j})$, corresponding to the pressure values of three adjacent and equally sized cells, the result is $(p_{i+1,j} - p_{i-1,j})/(2\Delta x)$. This is identical to the central difference approximation of Equation (4.8).

We can apply the same procedure to the cases in Figure 4.7. To evaluate the gradient at $p_C$, we first compute $p_{DE}$ by interpolating between the pressure values $p_D$ and $p_E$. Equation (4.12) can now be applied to the three points, $(x_1, p_A), (x_1 + \Delta x, p_C)$ and $(x_1 + \frac{7}{4}\Delta x, p_{DE})$, to give:

$$\frac{\partial p}{\partial x} = \frac{1}{21\Delta x} (-9p_A - 7p_C + 16p_{DE})$$

(4.13)
Figure 4.8: 1D adaptive shallow water simulation. Notice that the height changes smoothly over the boundary between cells of different size.

Figure 4.9: Adaptive version of our 2D solver showing tiled uniform grids of different resolutions (chosen arbitrarily in this example). The green tiles are 8x8 cells, the red tiles are 16x16 cells, the blue tiles are 32x32 cells and the yellow tiles are 64x64 cells.

Similarly, to evaluate the gradient at $p_D$, it is first necessary to compute $p_{BC}$ by interpolation. Solving Equation (4.12) with the three points, $(x_1, p_{BC})$, $(x_1 + \frac{3}{4}\Delta x, p_D)$ and $(x_1 + \frac{5}{4}\Delta x, p_F)$, gives:

$$\frac{\partial p}{\partial x} = \frac{2}{15\Delta x} (-4p_{BC} - 5p_D + 9p_F)$$

Equations (4.13) and (4.14) are used in place of central differences for cells which lie on the boundary between grids of different resolution. As stated before, these quadratic approximations are necessary to produce smooth transitions across boundaries (see Figure 4.8 for a 1D example and Figure 4.9 for a 2D example).
4.9.3 View Dependent Dynamic Adaptivity

Using our tiled grid scheme, it is relatively easy to develop a flexible adaptive solver which can efficiently simulate large bodies of liquid such as the open ocean. This is implemented using a two-level data structure which consists of a coarse grid of pointers to finer subgrids (tiles). Each tile represents a square region of the domain and is stored as a uniform grid of an appropriate resolution. Due to the nature of the pressure gradient discretization described in the previous section, the resolution of adjacent tiles must not differ by more than one level, i.e., the tiles surrounding a tile of resolution $2^n \times 2^n$ must be either $2^{n-1} \times 2^{n-1}$, $2^n \times 2^n$ or $2^{n+1} \times 2^{n+1}$, where $n \geq 1$. Also note that not all cells of the coarse grid need to point to valid tiles. If a tile pointer is null, it indicates that the region corresponds to an (open) boundary instead of fluid. Furthermore, the coarse grid can be stored using spatial hashing [THM+03] to allow unbounded domains. This data structure is very efficient and provides $O(1)$ data look-ups for any point in the domain. Although we have not parallelized our adaptive solver, we also note that the tiles provide a natural unit of work for partitioning the workload and should perform well when used in the context of dynamic load balancing.

There are various options that can be used for refinement criteria. However, for the purposes of interactive animation, it makes sense to use the following \textit{view dependent} criteria to avoid wasting computational effort processing regions of the domain that are out of sight. Firstly, tiles which are contained within the view frustum and are closer to the camera than a threshold distance are refined to the maximum resolution. Tiles that lie outside the frustum or beyond the threshold distance are progressively coarsened down to a minimum resolution. In addition, tiles that contain either static or dynamic objects may be refined to higher resolutions. Further criteria are possible, although we have not experimented with these. One possibility is to periodically check tiles to see if $\max(|\vec{u}|)$ is less than a threshold. If this condition is met, the fluid contained within the tile may be considered to be at rest and deleted. Another possibility is to progressively coarsen or refine tiles within the view frustum based on a measure of the activity of the flow (e.g., as defined by the norm of the local vorticity vector [Pop03]).

Conveniently, the tile-based layout automatically provides a heightfield with view dependent level-of-detail (LOD) which we can render by triangulating the grid points of those tiles that pass a frustum culling test. This avoids having to interpolate the
heightfield to new sample points. As is common in LOD-based heightfield rendering algorithms, “popping” artifacts can result when the mesh is refined or coarsened. The terrain rendering literature in computer graphics has proposed a variety of solutions to this problem. However, we use a very simple trick to smooth out transitions whenever a tile’s resolution needs to be reduced. To see how this works, consider four cells arranged in a square with values $q_a$, $q_b$, $q_c$ and $q_d$. Upon coarsening, these four cells will be replaced by a new cell with value $(q_a + q_b + q_c + q_d)/4$. To avoid transitioning abruptly, we update $q_a$, $q_b$, $q_c$ and $q_d$ by blending them into their average value over a small number of frames $n$, immediately preceding the transition. In other words, on frame $i$ of $n$ we set $q_a = (1 - i/n) q_{avg} + (i/n) q_a$, where $q_{avg} = (q_a + q_b + q_c + q_d)/4$, and similarly for $q_b$, $q_c$ and $q_d$. Finally, it is worth mentioning that an alternate rendering approach that could be used to reduce vertex popping is to resample the heightfield each frame onto either a variable resolution mesh which has uniformly distributed vertices in screen space [HNC02], or a fixed mesh that is tied to the camera’s position and is pre-tesselated to provide decreasing resolution as distance from the camera increases [Mit07].

4.10 Thin Boundaries

Wave equation solvers generally handle obstacles immersed in the flow by voxelizing them and preventing fluid from flowing into those voxels that contain any part of an obstacle. This approach limits the size of objects that can be resolved by the grid to
the size of a cell. Objects that have features smaller than this will either be treated as too big, or too small, depending on how voxelization is performed.

We propose a new method to handle infinitely thin objects. Instead of associating a single density variable with each grid cell, we add extra density variables to those cells that contain an obstacle boundary (similarly for velocity). For example, consider the simple configuration shown in Figure 4.10. Here, cell A and cell B both have two density samples since the line connecting their centre points is intersected by a boundary. In this case, the line partitions each cell into two regions and each region has an associated density variable. When packets are traced during the advection step of our solver, first of all, we do not allow packets to penetrate walls and, secondly, depending on which side of a wall the packet ends up, we update one density variable or the other. Then, in the pressure step, the wall dictates which density samples are used to compute the gradients. For example, when cell A’s vertical pressure gradient is being calculated, it will use cell’s B density value that lies on the same side of the wall as cell A’s sample (see middle frame of Figure 4.10). Similarly, the vertical pressure gradient at cell B’s centre will be calculated using whichever of cell A’s density samples lies on the same side of the wall as cell B’s centre point (see right frame of Figure 4.10). In this way, neither the advection step, nor the pressure step, ever accesses information on the opposite side of the wall to the point being updated.

We have added the ability to define thin walls as a list of line segments in our solver. Figure 4.11 shows a simple example where waves reflect off the boundary without mixing in any way. To represent closed boundaries, we also support 2D signed distance functions which are sampled on Cartesian grids. Here, the infinitely thin boundary
corresponds to the zero isocontour of the implicit function. One side of the boundary is defined by the positive isovalues, and the other side by the negative isovalues. Using this representation, it is very fast to determine which side of the boundary a point lies on by simply interpolating the distance field at the point’s location and examining its sign.

4.11 Visual Enhancements

4.11.1 Choppy Wave Modification

Figure 4.12: From left to right: normal in-plane grid vertices; in-plane grid vertices deformed by pressure gradient; corresponding pressure field; 3D heightfield with (top) and without (bottom) choppy wave modification.

After converting the results of shallow water simulation to a heightfield, we can optionally displace the grid vertices in the horizontal plane to make the water appear more or less choppy. We experimented with using velocity and pressure gradient information to compute the displacements but we achieved the best results by simply displacing the vertex coordinates from \((x, z)\) to \((x + s\frac{\partial p}{\partial x}, z + s\frac{\partial p}{\partial z})\), where \(p\) is the pressure computed in Section 4.6, and \(s\) is a scale factor which controls the degree of chopiness. It is also useful to clamp the gradient to avoid excessively large displacements which would result in mesh tangling. Although this simple approach is not physically based, it produces plausible results. An example of the effect this modification has on a typical heightfield is shown in Figure 4.12.
4.11.2 Procedural High Frequency Waves

Generally, interactive shallow water will be simulated at resolutions insufficient to provide the detail of high frequency waves that one would expect to observe. However, similar to [KT07], these can be added as a procedurally-animated displacement on top of the shallow water heightfield. To generate the displacement map each time step we use the FFT ocean wave synthesis algorithm described in [Tes04]. Figure 4.13 shows an example of a heightfield with and without the procedural displacements. This example uses a 128x128 grid to compute both the shallow water simulation and the displacement map.

We also experimented with a simple approach to couple the shallow water simulation to the procedural waves. We synthesize a low frequency wave field which is converted to a height field. This height field then corresponds to the ground height in our shallow water simulation. We compute the vertical acceleration of each grid point on the procedural heightfield and add this acceleration to the gravitational acceleration in Equation (4.3). In this way, the shallow water simulation responds to the motion of the underlying procedural waves, producing interesting motion that would not be possible using either technique in isolation.
4.11.3 Spray and Foam

To simulate small scale splashing detail, we create particles representing spray and foam based on the motion of the simulated waves. We maintain separate lists for spray and foam particles and update each list in turn, after the shallow water has been updated. The update step for the spray particles uses the midpoint method for time integration. A particle's acceleration is determined by the particle's weight and, if desired, a turbulent wind force. After evolving a particle forward in time, we check if its height is below the liquid surface and, if so, remove it from the spray list and add it as a new particle to the foam list. The new foam particle inherits the position of the spray particle and is assigned a random time which indicates when the particle should be deleted. Following update of the spray particles, we next update each foam particle in turn. First we check if the particle's age exceeds the time assigned at creation. If so, it is deleted. Next we set the particle's velocity to the local interpolated liquid velocity and integrate its position. Finally, we move the vertical position of the particle to lie on the liquid surface. After updating the particle lists, we check if new particles should be created. We sample random points on the liquid plane using stratified sampling to maintain a good distribution and, for each point, we check if its vertical velocity exceeds a threshold [OH95], or if the liquid curvature at the point exceeds a threshold [TFK⁺03]. If either of these conditions is met, a new spray particle is created at the location of the sample point with velocity inherited from the liquid velocity.
The spray particles can be rendered using the particle rendering framework described in Section 5.7.2. To render the foam particles, we use them to generate a greyscale foam density map. This can then be used to texture map or displacement map the liquid surface. This map is created by treating each foam particle as a Gaussian blob and rendering them additively into a texture. Alternatively, this can be done on the GPU by rendering points into a texture which is subsequently blurred. The resulting density map can be processed in various ways to determine the presence of foam. The simplest option is to threshold the map and use it to mask a precomputed noise texture representing the appearance of dense foam. However, instead of using a noise texture here, we use the FFT wave synthesis algorithm to generate a set of textures that tile space and are periodic in time. By using an animated texture, we can create the illusion of foam moving in response to high frequency surface waves. We use the algorithm in [Tes04] to generate choppy wave fields and treat the resulting heightfield as a foam texture. This is motivated by the fact that surface foam will tend to slide down the sides of steep waves and gather in wave troughs. However, we cannot determine a heightfield from the waves directly since the FFT algorithm for choppy waves includes horizontal vertex displacements. Instead, we resample the wave field by converting the grid of vertices into a set of triangles and then mapping from 2D texture coordinates to heights using ray casting. An example foam texture generated using this process is shown in the left of Figure 4.14, and the result after masking is shown on the right.
Chapter 5

Liquid Optics

5.1 Introduction

Rendering realistic images requires an accurate model of the interaction between light and matter. Empirical reflectance models have been developed to efficiently handle most common solid materials. These models are based on the assumption that light which strikes a surface is either reflected away from the surface in various directions or absorbed. However, for certain types of materials these local illumination models are insufficient. In the case of liquids and other highly transmissive materials, some of the light is transported beneath the surface where it may interact with the underlying medium. Here, the light at any point within the medium can be transported to any other point. This complexity makes it very expensive to accurately evaluate the radiance at a liquid surface.

Rendering realistic liquids is also greatly complicated by their varying appearance at different scales. For example, an ocean surface seen from far away can exhibit a relatively uniform appearance. However, at a smaller scale the intricate detail of a breaking wave can look very different. Millions of tiny droplets may be seen as water crashes down upon itself. At an even smaller scale fine clouds of wind-blown mist, torn from the crests of waves, may be visible. The detail inherent in these components is apparent when considering that their white appearance arises due to multiple scattering of light through the complex geometric structure of the aerated water. It is very difficult to capture all of these effects using a single liquid representation. For example, a mesh-
based representation is suitable for an ocean surface but may not do a good job of capturing the detail of splashing water. On the other hand, particles or density fields can be used to efficiently render spray or mist, but are bad at representing well-defined, coherent liquid surfaces.

Our strategy therefore is to use a combination of rendering techniques to separately render the large scale bulk volume and the smaller scale detail. The main volume is treated as a homogeneous participating medium that is rendered either as a triangle mesh or the zero level set of a signed distance field. The assumption of homogeneity greatly simplifies the interaction between light and liquid, and makes it possible to rapidly simulate light propagation through the medium. For the smaller scale liquid components, the use of an inhomogeneous medium provides the ability to compute light transport through the liquid that is too fine to explicitly represent geometrically.

This chapter begins with an overview of the basic principles governing the interaction between light and liquids. It then reviews the standard raytracing algorithm and describes how this algorithm can be used to efficiently render liquids using an analytical single-scattering approximation adapted from atmospheric optics literature [Pre03]. An approach for path tracing through homogeneous volumes is then described, along with a novel technique for accelerating the expensive tracing steps. This is followed by a description of an OpenGL framework for rendering high quality animations using a software accumulation buffer. The raytracing approximations for liquids are then revisited and tailored for the special case of heightfields, thereby enabling their acceleration on the GPU. Finally, a number of techniques for rendering and lighting particles are proposed. In particular, we describe how a coarse grid can be used to compute fast self-shadowing and we provide an in-depth analysis of the optical phenomena that arises when the particles are treated as spherical water droplets.

5.2 Radiometry of Liquids

5.2.1 Surface Scattering

Understanding how unpolarized light interacts with liquids is easiest to explain in terms of photon optics (see Figure 5.1). When a photon arrives at a liquid-air interface, it can either bounce off the surface or pass through it. We make the assumption that
surface roughness at all scales is accounted for geometrically. This allows us to treat all interactions as perfectly specular.

The probability that a photon is reflected off the liquid surface, rather than being transmitted, is given by the Fresnel equations. These equations depend on the index of refraction of the liquid and the angle between the incident photon direction, \( \vec{I} \), and the surface normal, \( \vec{N} \). For incident directions perpendicular to the surface, the reflectance is almost zero, but as the incident direction becomes more and more parallel to the surface, the reflectance smoothly increases to 1. Schlick [Sch94] provides a computationally cheap approximation to this curve: \( F_0 + (1 - F_0) \left( 1 - \vec{I} \cdot \vec{N} \right)^5 \), where \( F_0 \) is a constant dependent on the index of refraction of the liquid.

Photons that bounce off the liquid surface are assumed to be reflected from a mirror plane defined by the surface normal. Their new direction is given by: \( \vec{I} - 2 \left( \vec{I} \cdot \vec{N} \right) \vec{N} \). Since we assume that the liquid is perfectly smooth, this is the only path that a reflected photon may take. Alternatively, liquids that appear glossy can easily be handled by sampling new directions from a cone centered on the mirror direction.

For photons that pass through the surface into the volume, refraction occurs at the interface. According to Snell's law, this causes the direction of the photon to bend by an amount dependent on the index of refraction of the liquid. In cases where the angle between \( \vec{I} \) and \( \vec{N} \) exceeds a threshold known as the critical angle, total internal reflection occurs and the photon is reflected off the boundary instead of being refracted.

The actions of reflection and refraction alone can produce many visually striking effects that are challenging to render. For example, a wavy water surface behaves like a lens where successive wave peaks and troughs focus and defocus transmitted light. This can cause uniform incident illumination to be transformed into complex patterns of bright and dark light beams that are visible as caustics when the light falls on a diffuse surface (such as the floor of a swimming pool), or sunbeams when the light illuminates the participating medium itself. These effects are further complicated by the fact that the index of refraction varies with the wavelength of light. This phenomenon, called dispersion, causes light to refract in different directions at the interface. Fortunately, in circumstances where the liquid appears optically thick, the light is often attenuated or scattered enough to make these complex effects indiscernible.
5.2.2 Volume Scattering

Photons that refract at the boundary and enter the liquid medium will continue along a straight line path until they either interact with the medium, or exit again through the surface. There are two ways that photons can interact with the medium.

The first type of interaction is absorption. This is described by a wavelength-dependent parameter, $\sigma_a$, equal to the probability that a photon will be absorbed within a distance of 1 unit. Absorption causes an exponential rate of reduction in the radiance, $L$, of a ray over a distance $ds$. It is described by the following differential equation:

$$\frac{dL}{ds} = -\sigma_a L \quad (5.1)$$

The second type of interaction is scattering. Again, this is described by the probability of its occurrence per unit length, $\sigma_s$. Photons may scatter many times as they propagate through a medium. For highly scattering media, this causes a diffusion of light that accounts for the soft, translucent appearance of many materials, including liquids such as milk and orange juice. Scattering events are usually separated into two kinds: out-scattering and in-scattering. Out-scattering occurs when a photon is scattered out of the direction of a ray causing a reduction in its radiance. The treatment of out-scattering is identical to absorption. In both cases, the coefficients, $\sigma_a$ and $\sigma_s$,
and $\sigma_s$, describe the fraction of radiance that is attenuated per unit distance. So, for out-scattering, the differential change in radiance $dL$ over a distance $ds$ is:

$$\frac{dL}{ds} = -\sigma_s L$$ (5.2)

While radiance is reduced due to out-scattering, in-scattering has the opposite effect. Photons that are scattered into the direction of a ray cause an increase in radiance. This situation is a lot more difficult to handle than absorption or out-scattering since, in this case, it is necessary to first determine the light that intersects any point on the ray, and then determine what fraction of that light scatters into the ray direction. At a single point on the ray, determining the light that potentially scatters into the ray requires integration over all directions, $\hat{\omega}$, on the sphere, $\Omega_{4\pi}$. For each of these directions, the amount of incoming radiance, $L_i$, scattered into the ray direction needs to be determined. This is dependent on the probability, $\sigma_s$, that a photon is scattered, and also the probability that this photon is scattered into the direction of the ray. The latter probability is described by a phase function, $p(\theta)$, where $\theta$ is the angle between the photon direction and the ray direction. The total change in radiance due to in-scattering is given by the following equation:

$$\frac{dL}{ds} = \sigma_s \int_{\Omega_{4\pi}} p(\theta) L_i(\hat{\omega}) d\hat{\omega}$$ (5.3)

The choice of phase function depends on the physical properties of the medium. Although tabulated phase functions are sometimes used in oceanography for specific types of water, a far more flexible option is to use the empirical Henyey-Greenstein phase function [HG41]:

$$p(\theta) = \frac{1 - g^2}{4\pi (1 + g^2 - 2g \cos \theta)^{3/2}}$$ (5.4)

In this equation, the parameter $g$ is the average cosine of the scattering angle and describes whether the medium is biased towards forward (positive $g$) or backward (negative $g$) scattering. This phase function is commonly used in oceanic optics [Hal02] and computer graphics due to its generality. In particular, Narasimhan et al. [NGD+06] have published a dataset of spectrally varying scattering parameters (including values of $g$) for over forty different liquids.
Finally, the total gains and losses described by Equations (5.1), (5.2) and (5.3) may be combined into a single integro-differential light transport equation:

$$\frac{dL}{ds} = - (\sigma_a + \sigma_s) L + \sigma_s \int_{\Omega_{se}} p(\omega) L_i(\omega) \, d\omega$$  \hspace{1cm} (5.5)

Considering the complexity of this equation, it is important to consider the circumstances in which correct modelling of absorption and scattering is required to render realistic looking liquids. In many cases, the optical properties of the medium dictate the importance of absorption and scattering effects. For example, scattering must be accounted for in milk since its white appearance is due to multiple scattering of light. Other situations are dependent, not only on properties of the medium, but also on the optical thickness of the medium. For example, the appearance of a glass of water will not be affected much by either absorption or scattering. Although both of these interactions occur in water, they have a low probability of occurring over the length scale of a glass. At a larger scale, such as the case of a swimming pool, absorption of red and green light by water molecules within the participating medium gives the water its bluish appearance. In this case, absorption must be simulated, but the effect of scattering is far less noticeable. For a body of water as large as an ocean, scattering becomes much more important and is mainly due to organic matter and minerals suspended in the water volume. For a deep ocean, in the absense of scattering, no light would exit from the volume since all of the blue light would be absorbed over the large distances that photons are required to travel before returning to the observer. As a result, the water would appear black. In a case such as this, accurately accounting for the effects of in-scattering is essential in order to produce the correct appearance.

5.3 Ray Tracing Liquids

5.3.1 Whitted Ray Tracing

The earliest ray tracing implementations were based on the recursive algorithm of Whitted [Whi80]. In this algorithm, eye rays are cast through the image plane into the scene. Whenever an object is intersected, a local illumination model is evaluated and, in the case of specular surfaces, new rays are spawned and traced recursively in the
Figure 5.2: Whitted’s recursive ray tracing algorithm accurately accounts for perfect specular reflection and refraction which makes it appropriate for rendering liquids.

directions of perfect specular reflection and refraction (see Figure 5.2). The recursion continues until either a predetermined recursion depth is reached, or the radiance of new rays is contributing little to the radiance of the eye ray. This simple algorithm is limited in that it only correctly handles specular surfaces. However, by weighting the radiance values of reflected and refracted rays by the Fresnel reflectivity and transmissivity coefficients, and accounting for the effect of the participating medium, it is possible to realistically render many types of liquids.

5.3.2 Absorption

Since the effect of an absorbing medium is to remove light from a ray, it is possible to exactly account for its effect by attenuating the radiance of any ray that passes through the medium. For a ray with initial radiance $L_0$, we can compute its radiance after travelling a distance $s$ through the medium by integrating Equation (5.1) from 0 to $s$:

$$L = L_0 e^{-\sigma_s s}$$  \hspace{1cm} (5.6)

This equation describes the exponential rate of attenuation of light with distance
Figure 5.3: White light turns blue when passing through an absorbing medium (left). For scattering media, additional light bounces into the path based on the angle it makes with the light source (right).

through a homogeneous medium ($\sigma_a$ is constant). For example, Figure 5.3 illustrates how white light transitions to blue light over increasing distance. To incorporate this effect into our ray tracer, we compute the path length of any ray that traverses the medium and modulate its radiance by the exponential decay factor. An example result is shown in Figure 5.4.

Figure 5.4: Ray-traced liquid featuring absorption.

5.3.3 Scattering

Unlike absorption, it is not possible to simulate the effects of scattering using the basic ray tracing algorithm. However, by introducing some simplifying assumptions, it is possible to produce plausible scattering effects at little additional cost. Assuming the phase function and absorption/scattering coefficients do not vary through space, integration of Equation (5.5) for a path of length $s$ yields the full volume rendering equation for the case of a non-emissive, homogeneous participating medium:
Figure 5.5: Correct evaluation of single scattering requires ray marching to account for attenuation of in-scattered light.

\[ L = L_0 e^{-(\sigma_a + \sigma_s)s} + \int_0^s \left( e^{-(\sigma_a + \sigma_s)(s-x)} \sigma_s \int_{\Omega_{4\pi}} p(\theta) L_i(x, \bar{\omega}) d\bar{\omega} \right) dx \]  \hspace{1cm} (5.7)

The first term attenuates the ray's radiance, \( L_0 \), to account for absorption and out-scattering. This is handled exactly as described in Section 5.3.2. The second term is an additive term that increases the radiance to account for in-scattering from all directions, at all points on the ray path. Notice that the in-scattered radiance at each of these points needs to be attenuated based on how far the light has to travel to reach the end of the ray. Also note that, since the incoming radiance from any direction will itself be subject to absorption and scattering, evaluating the in-scattering term requires recursive evaluation of Equation (5.7). This means that any part of the medium can be illuminated by any other part which makes it very computationally expensive to evaluate, even if only low-order scattering effects are considered.

The most common approach to compute in-scattering is to use numerical integration by converting the definite integral into a sum over a discrete set of samples. Each sample corresponds to a small segment of the path, with constant incident illumination. This approach is known as ray marching. For each sample point, the integrand needs to be computed, which requires integrating over the sphere of directions to compute the incident radiance at that point. This can also be done by summing over a discrete set of directional samples. However, considering the exorbitant number of samples required to evaluate in-scattering with full multiple scattering, ray marching usually
ignores indirect illumination altogether and instead only considers direct illumination. In this case, referred to as *single scattering*, in-scattering is due only to photons that come directly from the light source and scatter at the sample point into the direction of the ray. This greatly cuts down on computations by only considering directions towards light sources, instead of the entire sphere of directions. Note however that this necessarily ignores refraction at the boundary between the sample point and the light source, since this would require determining the boundary point that results in refraction towards the light source.

Single scattering is a reasonable approximation for optically thin media, such as water under most conditions. However, correct evaluation of single scattering still requires ray marching, even though we are dealing with homogeneous media. This is to account for attenuation of in-scattered light through the volume. As shown in Figure 5.5, at each sample point, a ray needs to be sent (though not marched) towards each light source to determine how much attenuation occurs. This increases the computational cost significantly. We can avoid ray marching entirely by assuming that light sources
are directional and that they are not attenuated by the participating medium. This allows the phase function and incident radiance to be treated as constants over the path. For a single light source with direction $\vec{u}$, the analytical solution to the integral is:

$$\int_{x=s}^{x=0} \sigma_s p(\theta) L_i(\vec{\omega}) \left( \frac{1}{\sigma_a + \sigma_s} \right) \, dx = \frac{\sigma_s}{\sigma_a + \sigma_s} p(\theta) L_i(\vec{\omega}) \left( 1 - e^{-\sigma_a - \sigma_s s} \right) \quad (5.8)$$

This cheap, single scattering approximation avoids ray marching at the cost of accuracy. Figure 5.3 (right) shows the effect it has on radiance as a function of $s$ and $\theta$. Compared to absorption alone, the approximation makes the liquid look a lot more visually interesting, while still giving plausible results as shown in Figures 5.6 and 5.7.

### 5.4 Path Tracing Liquids

#### 5.4.1 Monte Carlo Rendering

While ray tracing produces convincing looking liquids for absorbing media, the inability to accurately compute in-scattered light severely limits the realism that can be achieved. As we discussed in the last section, it is possible to account for light that scatters multiple times using ray marching. However, this is extremely time-consuming due to the large number of rays that must be spawned to sample incoming radiance at all points, on all paths, through the participating medium. While single-scattering approximations are acceptable in some cases, for highly scattering media, such as milk, they do not produce realistic results.

Kajiya [Kaj86] introduced a simple extension to the basic Whitted ray tracing algorithm that supports full global illumination by performing a Monte Carlo simulation of light transport. In addition to recursively tracing specular paths, path tracing also traces diffuse reflection paths. By stochastically sampling all possible light paths in this way, the algorithm can trivially handle colour bleeding, dispersion, caustics and other complex effects that are difficult to achieve with other algorithms.

The basic idea behind the algorithm is to shoot a large number of adjoint photons\(^1\)

\(^1\)When photons are traced from the eye to the light, instead of the other way around, they are
from the eye and let them scatter through the scene until they are either absorbed or hit a light source. Photons are seeded with a random pixel position, aperture position, time, and wavelength, which provides support for anti-aliasing, depth of field, motion blur and spectral effects such as dispersion. By averaging the response of a large number of adjoint photons, the incoming radiance over each pixel can be estimated.

The path tracing algorithm is very robust and general. It can simulate all types of light scattering and its unbiased nature ensures that it will produce a physically accurate result, given sufficient time to converge. The downside is that the images produced tend to look noisy unless a very large number of samples are taken. Hence, these methods are generally quite time-consuming.

5.4.2 Probabilistic Intersections

The Monte Carlo path tracing algorithm can be used to compute a full solution to Equation (5.7). We follow the approach advocated by Morley et al. [MBJ+06]. In particular, we do not explicitly compute direct lighting (for surfaces or volumes) and instead rely on importance sampling of BRDFs, phase functions, and light sources for referred to as adjoint photons [MBJ+06].
variance reduction. We also dispense with ray marching, preferring instead to simulate the scattering path each photon takes within a participating medium. Finally, each photon represents a single wavelength instead of an entire colour. This allows spectral phase functions (e.g., different values of the Henyey-Greenstein parameter, \( g \), for each wavelength), and the opportunity to use published oceanographic data for wavelength-dependent index of refraction [Hui97], \( \sigma_a \) [PF97] and \( \sigma_s \) [BHD94]. This generalized approach correctly handles refraction of multiple scattered light at boundaries and supports all types of scattering effects within media, including volumetric shadows and caustics. Some examples are shown in Figure 5.8.

The basic path tracing algorithm for translucent materials, such as liquids or glass, works as follows. When a photon collides with the boundary of the medium, the Schlick Fresnel function is sampled to determine if the photon should be reflected or refracted. If a photon is refracted and enters the medium, we probabilistically decide if an interaction occurs before the photon exits again. To do this we first compute the times at which the next absorption and scattering events occur by drawing samples from the exponential distributions describing the probabilities of absorption and scattering:

\[
\begin{align*}
    t_a &= -\log(\xi_1)/\sigma_a \\
    t_s &= -\log(\xi_2)/\sigma_s
\end{align*}
\]

where \( \xi_1, \xi_2 \) are uniform random numbers in \([0,1]\). We then compare these values to the time at which the photon will exit the medium. If the exit time is smaller than \( t_a \) and \( t_s \) then the photon passed through the medium unaffected; if \( t_a < t_s \), the photon is absorbed; otherwise the photon is scattered by sampling the Henyey-Greenstein phase function to determine its new direction. By repeating this over and over we can evaluate the scattering path that a photon takes through the medium until, eventually, it is either absorbed or refracts out of the medium and back into the scene.

### 5.5 Accelerated Light Propagation

#### 5.5.1 Motivation

While Monte Carlo path-tracing can produce very realistic images of liquid, it converges very slowly in optically thick configurations, especially when the probability of
a scattering event is much greater than an absorption event, or the medium is highly forward scattering. In these circumstances, photons that penetrate a liquid surface may bounce internally many times before exiting again. The deeper the photon travels into the object, the harder it is for it to find its way back to the surface and hence, the longer it takes to process these photons. This issue is typically handled in renderers by prematurely terminating the path after a certain number of bounces. However, doing so can lead to biased results or images that take a long time to converge.

To exemplify the potential expense involved in computing scattering paths, we sent photons into a large slab of milk, defined using the parameters from [NGD*06]. When the medium was treated as forward scattering ($g = 0.75$), the probability that a photon would be absorbed before exiting the object was 5%. On average a photon scattered 633 times before either exiting or being absorbed. In the worst case, photons scattered thousands of times. When an isotropic phase function was used instead, the paths were slightly shorter, although fewer photons were absorbed (2.5%). However, photons still scattered an average of 312 times. In cases such as this, computation can easily become dominated by the evaluation of complex paths of photons as they scatter through an object. In particular, rendering large bodies of liquid (such as an ocean) could be considered the worst case scenario for optically thick, scattering media found in everyday circumstances.

In the following sections, we present a new path sampling technique for photons propagating through a homogeneous participating medium, which results in improved rendering times for scattering media. Instead of simulating the path of a photon as it undergoes a series of scattering events, our technique is able to rapidly transport photons over arbitrarily large distances inside a volume, thus reducing the likelihood that a photon will get trapped scattering inside an object. Our approach enables photons that have penetrated an object to traverse space without requiring phase function sampling to determine new directions for scattered photons, or costly ray intersection tests to determine when photons exit the surface.

5.5.2 Overview

Consider a photon travelling through a participating medium. At some instant in time, the photon has position $p$ and direction $d$. Now consider a sphere whose radius
is equal to the distance from $p$ to the closest point on the boundary of the medium. Assuming the photon is not absorbed, it may undergo a series of scatters within the sphere. At some future time, the photon will cross the surface of the sphere at some random position, say $p'$, with random outgoing direction $d'$.

Our approach provides a way to "teleport" the photon from position $p$ with direction $d$, to a new position $p'$ with new direction $d'$. This is done in such a way that the outcome is statistically the same as if we had simulated the individual scattering events of the photon within the sphere. The information that determines the likelihood of a photon leaving the sphere at some point in some direction is precomputed and sampled at render time to determine new random positions and directions for photons.

Note that the new position $p'$ of the photon will not necessarily be any closer to the surface of the medium than $p$. However, by repeating this process of teleporting the photon between the surfaces of spheres of different radii, where the spheres are chosen to be as large as possible while still being fully contained within the medium, we avoid the processing which would otherwise be required to simulate the scattering of the photon within each of these spheres. The idea here is that, if the radii of the spheres are many times greater than the photon’s mean free path length, then the photon will bypass the majority of scattering events and traverse space with far fewer computations, as illustrated in Figure 5.9.

The following sections will discuss how we precompute the probability distributions that allow us to predict where, and in what direction, a photon will exit a sphere, how this data is used to choose new outgoing positions and directions on a sphere, and how this sampling scheme is used in the context of our path tracer to accelerate photon tracing within participating media.

### 5.5.3 Precomputation

The previous section discussed transporting a photon with position $p$ and direction $d$ to a new outgoing position $p'$ on the surface of a sphere of some arbitrary radius, with new direction $d'$. The probability of choosing a particular outgoing position and direction should equal the probability that the original photon would have exited the sphere with that position and direction. In other words, the sphere needs to be importance sampled to determine the new position and direction of the photon. To be able to
Figure 5.9: On the left, a photon enters a participating medium and undergoes a series of scatters before exiting; on the right, when the photon reaches a certain distance within the medium, it is “teleported” between spheres of different radii until it is back within reach of the surface. Less computation is required to compute the path of the photon on the right.

To perform this importance sampling we first need to compute a numerical probability density function (PDF).

The PDF we are interested in describes the probability of choosing a particular outgoing position and direction from the sphere, given an initial position and direction. This is a 9D function. Fortunately, we can reduce the dimensionality of this function to 3D, without loss of generality, by first defining a translation which maps \( p \) to the origin, and a rotation which maps \( d \) to the unit vector pointing straight up. Note that the inverse of this transformation can be easily applied, as we will see in the next section, to map the PDF back to the original coordinate system given by \( p \) and \( d \). Defining the PDF in this new coordinate system reduces it to a 4D function of outgoing position and direction, since the initial position and direction are now fixed, as shown in the left panel of Figure 5.10. For a sphere of a given radius \( r \), we write the outgoing position in spherical coordinates as \( (r, \alpha, \beta) \), the outgoing direction as \( (1, \theta, \phi) \), and the PDF as \( p(\alpha, \beta, \theta, \phi) \). Due to radial symmetry about the polar axis, the PDF is separable into \( p(\beta) = 1/2\pi \) and \( p(\alpha, \theta, \phi) \).

Using Monte Carlo simulation, we construct a 3D piecewise constant representation of \( p(\alpha, \theta, \phi) \), and also compute the probability that a photon will be absorbed before
Figure 5.10: The PDFs are defined in a coordinate system where the photon is located at the origin, directed upwards (left). A fraction of the photons that exit the sphere when precomputing a PDF are shown, along with their outgoing directions, which are visualized by mapping x, y and z coordinates to red, green and blue colour components respectively (right).

exiting the sphere. Algorithm 1 shows how to simulate a single photon. This procedure is run for a large number of photons, e.g., one million. The probability of absorption is computed simply as the fraction of photons absorbed. To compute $p(\alpha, \theta, \phi)$, we subdivide each of these three angles over their range to form a set of bins, i.e., each bin corresponds to a unique set of outgoing positions and directions. The angles $\theta$ and $\phi$ are subdivided such that each bin covers an equal solid angle, whereas $\alpha$ is uniformly subdivided to provide more resolution at the poles. This is beneficial, considering that many of the materials we are interested in are highly forward scattering.

We associate a scalar value with each bin, equal to the probability that a photon will exit the sphere through that bin. For each photon, we need to map its exit position and direction from Cartesian coordinates to $\alpha, \theta, \phi$ coordinates. Converting the position and direction into spherical angles gives four values but, as discussed above, we know that the azimuthal angle of the position has a constant PDF of $1/2\pi$. This implies that we can disregard that angle if we first rotate the position and direction of each photon to make its azimuthal angle zero. From here we can easily map the $(\alpha, \theta, \phi)$ value to its corresponding bin and hence determine the fraction of photons that exited the sphere through that bin, in order to compute its probability. The right panel of Figure 5.10 shows the distribution of a small fraction of the total exiting photons used.
Algorithm 1: Simulating a photon in a sphere.

\begin{verbatim}
pos ← (0,0,0)
dir ← (0,1,0) // up
state ← SCATTER
repeat
    t\text{exit} ← IntersectSphere((0,0,0), \text{radius}, \text{pos}, \text{dir})
    t_a ← -\log(\text{GetRandom()})/\sigma_a
    t_s ← -\log(\text{GetRandom()})/\sigma_s
    if min\{t_a, t_s\} >= t_{\text{exit}} then
        exit_pos ← pos + dir \times t_{\text{exit}}
        exit_dir ← dir
        state = EXIT
    else
        if t_a < t_s then
            state = ABSORB
        else
            // scatter
            pos ← pos + dir \times t_s
            dir ← ImportanceSamplePhaseFunc(dir)
        end
    end
until state ≠ SCATTER
\end{verbatim}

to compute these probabilities.

5.5.4 Sampling

Given an initial position and direction for a photon, which is located at the centre of a sphere of a certain radius, we first check if the photon gets absorbed before reaching the surface of the sphere. This is done by comparing a uniform random variable to the probability of absorption for the sphere. If the photon is not absorbed then we need to compute its new direction and position on the surface of the sphere. The multidimensional, piecewise constant probability density, \( p(\alpha, \theta, \phi) \), described in the previous section, provides us with the information we need to randomly sample according to the correct distribution. We begin by describing how to convert random numbers over \([0,1]^3\) to samples drawn from the 3D \((\alpha, \theta, \phi)\) distribution.

The inversion method [PH04] can be used to efficiently draw samples from a 1D
PDF by computing a 1D cumulative distribution function (CDF), $P(x)$, and evaluating its inverse $P^{-1}(\xi)$ at uniformly distributed random locations, $\xi \in [0, 1]$. Samples can also be drawn from multidimensional distributions by considering a “cascading set” of 1D CDFs [LRR05]. In this case, each dimension is sampled in turn, based on the values chosen for previous dimensions. In the case of $p(\alpha, \theta, \phi)$, we need to compute CDFs for $P(\phi | (\alpha, \theta))$, $P(\theta | \alpha)$ and $P(\alpha)$. Given these, we can then draw an $(\alpha, \theta, \phi)$ sample by first drawing an $\alpha$ sample from $P(\alpha)$. Using this, we sample $\theta$ from $P(\theta | \alpha)$. And finally, given $(\alpha, \theta)$, we draw a $\phi$ sample from $P(\phi | (\alpha, \theta))$. Algorithm 2 shows how this translates into pseudocode. Note that the Sample() function draws random samples from 1D CDFs. Its output is the index of the bin from which a sample was drawn, and the probability value in $[0, 1]$. Probability values in $[0, 1]^3$ can be converted to $(\alpha, \theta, \phi)$ values by multiplying by $(\pi, \pi, 2\pi)$.

The next step is to compute $\beta$, the azimuthal angle describing the outgoing position. As discussed earlier, $p(\beta)$ is constant, so we can compute its value by drawing a sample uniformly from $[0, 2\pi]$. This angle is then used to rotate the outgoing direction (by adding $\beta$ to $\phi$), so that it will point in the correct direction relative to the photon’s outgoing position. We now need to convert $(\alpha, \beta, \theta, \phi)$ into the outgoing position and direction by mapping from spherical to Cartesian coordinates. However, remember that $p(\alpha, \theta, \phi)$ assumed a photon located at the origin, directed upwards. Since we are really interested in photons located at different positions with arbitrary directions, the final step is to transform the photon’s new position and direction so that it is correct relative to the old position and direction, as shown in Algorithm 2.

### 5.5.5 Rendering

So far we have described how to transport a photon from the centre of a sphere of a given radius to a point on its surface. In this section, we will discuss how this is used in the context of our path tracer.

When a photon enters a participating medium, we evaluate its distance, $d$, to the boundary of the volume. If $d$ is less than a threshold $\tau$ (explained later), we use the traditional approach described in Section 5.4.2 to propagate the photon. Otherwise we do the following: First, we use a binary search to select a radius, $r$, from a finite set of radii, for which the PDF described earlier has been precomputed. We choose
Algorithm 2: Sampling a photon's new state.

input : pos, dir
output: exit_pos, exit_dir

if GetRandom() > absorption_prob then
    \( (\text{bin}_\alpha, t_\alpha) \leftarrow \text{cdf}_\alpha\text{-Sample()} \)
    \( (\text{bin}_\theta, t_\theta) \leftarrow \text{cdf}_\theta[\text{bin}_\alpha]\text{-Sample()} \)
    \( (\text{bin}_\phi, t_\phi) \leftarrow \text{cdf}_\phi[\text{bin}_\alpha][\text{bin}_\theta]\text{-Sample()} \)
    \( \alpha \leftarrow \pi t_\alpha \)
    \( \beta \leftarrow 2\pi \times \text{GetRandom()} \)
    \( \theta \leftarrow \pi t_\theta \)
    \( \phi \leftarrow \beta + 2\pi t_\phi \)
    \( (\mathbf{u}, \mathbf{v}, \mathbf{w}) \leftarrow \text{MakeOrthonormal Basis(dir)} \)
    \( \text{exit_pos} \leftarrow \text{pos} + r \cos \beta \sin \alpha \mathbf{u} + r \sin \beta \sin \alpha \mathbf{v} + r \cos \alpha \mathbf{w} \)
    \( \text{exit_dir} \leftarrow \cos \phi \sin \theta \mathbf{u} + \sin \phi \sin \theta \mathbf{v} + \cos \theta \mathbf{w} \)

end

the largest \( r \) possible that is less than \( (d - \varepsilon) \) (\( \varepsilon \) is a small value that ensures that the photon stays within the volume). We now consider the photon to be located at the centre of a sphere of radius \( r \). Since we have the PDF for this sphere, we use Algorithm 2 to determine if the photon is absorbed and, if not, where the photon moves to on the sphere’s surface. With the photon at its new position and direction, we reevaluate \( d \) and repeat the process until the photon exits the volume.

We choose \( r \) to equal the mean free path length for scattering, \( 1/\sigma_s \). This is also the minimum radius for which we precompute a PDF. The radius increment and maximum radius that define the rest of the PDFs depend on the scattering coefficients and the scale of the volumes we wish to render. However, we found that ten or more integral multiples of the mean free path length worked well. Also, since the PDF is dependent on the material properties of the volume, which themselves are wavelength-dependent, we need to compute separate PDFs for sets of radii corresponding to red, green and blue wavelengths. At render time, we sample from whichever of these is closest to the wavelength of the photon in question.

To evaluate the distance \( d \) to the closest point on the bounding mesh of the volume, we use a signed distance function, sampled on a uniform grid. We can compute a quick approximation to this function by exploiting our ray tracer’s ability to perform fast ray intersection tests. At each grid point for which we want to calculate signed distance,
we uniformly sample directions over the sphere and shoot rays in these directions to determine the distance to the closest intersection point. The sign is given by the dot product of the ray direction and the normal at the intersection point. Although this method is not robust, it is easy to implement and works well for the scenes we have tested. We can also avoid both the trilinear interpolation required to evaluate signed distances, and the search required to find the best radius, by precomputing a uniform grid where each cell stores either an index into the list of radii or $-1$ if the cell is too close to the boundary. By mapping the photon’s position to its corresponding cell, we can retrieve the appropriate radius in a single look-up, without having to evaluate the distance to the boundary at run-time.

5.5.6 Results and Discussion

We have used our Monte Carlo path tracer to render comparison images with and without the acceleration technique described in the previous sections. The images in Figure 5.11 were rendered at 300x600 resolution with 2,500 samples per pixel, while
Figure 5.12: The image on the left shows a participating medium rendered in 135 minutes using pure Monte Carlo path tracing. The image on the right shows the same scene rendered in 71 minutes using accelerated photon propagation. Again, there is no perceptible difference between the images.

those in Figure 5.12 were rendered at 500x500 with 1,600 samples per pixel. All renderings were performed on a Xeon 3.6GHz PC.

The Buddha model in Figure 5.11 is rendered with a translucent appearance by enclosing the participating medium within a dielectric boundary. This is exactly the same approach we use to render liquids, except that in this case an index of refraction of 1.5 was used, corresponding to glass. The medium properties were chosen such that the average scattering albedo is approximately 0.9, and the average cosine of the scattering angle is 0.85. These values describe a highly scattering medium that is similar to the properties of many natural materials such as skin. Note that the Buddha model we render is optically thin in many places due to its scale. Our approach performs significantly better for optically thick configurations due to the increased number of photon bounces within the volume, but even in this case, we achieve an approximately 2× performance increase.

The number of scattering events is reduced by 61% due to teleportation. Overall, 2.6 billion scatters were replaced by 0.7 billion teleports, due to the fact that each teleport aggregates many scattering events. In addition, with the accelerated algorithm, 57% of absorption events are detected using the conditional at the top of Algorithm 2. This test is much faster to perform than computing the time of absorption and comparing that to the time the photon exits the boundary, as described in Section 5.4.2.

Figure 5.12 shows a horse model rendered as a high albedo participating medium.
Again, the speedup is approximately $2\times$. In this case, 59% of the scattering events are avoided by teleportation. This means that the remaining 41% of scatters occur very close to the boundary of the medium and are not accelerated. For the scenes we have tested, this seems to be the limiting factor in achieving greater than $2\times$ speedup. Note that, for this scene and the Buddha scene, the average number of scattering events per pixel sample is approximately 5, i.e., each teleport does not avoid a substantial number of scatters. As we pointed out in Section 5.5.1, for materials such as milk, the average photon can experience several thousand scattering events within the medium. For liquids such as these, our algorithm will give substantially greater savings by aggregating hundreds of scatters (instead of a few) into each teleport.

For the Buddha scene, the time to precompute the 24 PDFs (a set of 8 radii for each of red, green and blue material parameters) was approximately 3 minutes. This preprocess only needs to be performed once for each material. Each PDF was quantized into $16\times16\times16$ bins and computed using one million photons. Although we have not examined sensitivity of results to the number of bins, we have not noticed any artifacts due to tabular sampling of the PDFs. In fact, considering that the PDFs are very smoothly varying functions, we do not expect more bins to give a noticable increase in quality. Finally, although compression techniques such as non-negative matrix factorization [LS00] could be used to reduce the storage space required for the data, we felt that this was unnecessary given that a very moderate 400 KB of storage was required for these experiments.

As we have shown, our acceleration technique results in improved performance and does not compromise the accuracy of a full Monte Carlo solution. In fact, the only bias our scheme introduces is due to sampling from discrete instead of continuous PDFs. Photons only begin teleporting when they are determined to lie within the medium, and teleporting a photon does not imply that an interaction with the medium occurred. For example, the PDF corresponding to a non-scattering medium will not induce scattering. In fact, teleportation has no effect on the probabilities of interactions occurring. Consider, for example, the probability, $p(A)$, that a photon will be absorbed along a path of length $d$: $p(A) = 1 - e^{-\sigma_d}$. Now consider that the photon is teleported by distance $t$, somewhere between its initial position and $d$. The photon must now traverse two subpaths, of length $t$ and $d - t$ (analogous to spheres of radii $t$ and $d - t$). Let $p(B)$ and $p(C)$ be the probabilities of absorption along each of these subpaths:
\[ p(B) = 1 - e^{-\sigma a}, \quad p(C) = 1 - e^{-\sigma d(d-t)}. \] Since the photon is tested for absorption along both subpaths, the probability that the photon is absorbed along the full path is
\[ p(A \lor B) = p(A) + p(B) - p(A)p(B) = 1 - e^{-\sigma d} = p(C), \] which shows that teleporting the photon does not affect the probability of absorption. The same argument holds for scattering events.

## 5.6 GPU Rendering

### 5.6.1 Exploiting Graphics Hardware

While ray tracing and path tracing can both produce very realistic images, they are often too slow to produce full, motion blurred animations, requiring hours or even days per frame. On the other hand, modern graphics hardware is specifically designed to render scenes at real-time frame rates. This usually necessitates the use of less sophisticated rendering methods compared to ray-based algorithms, thus resulting in inferior quality. However, in recent years, GPUs have gradually moved away from a fixed function pipeline and have become increasingly programmable, making it possible to implement advanced shading algorithms on the GPU. In fact, recently the GPU has also been explored as a tool for accelerating offline, production quality rendering [RKKS+07, PVL+05, WGER05]. However, these approaches generally treat the GPU as a general processing unit and, as such, do not leverage the full potential of the hardware. We advocate instead the use of traditional rendering techniques based on triangle rasterization and programmable shaders for offline rendering of liquids.

We have designed a unified framework built on top of OpenGL that allows both real-time previewing and high quality offline rendering. This enables the rapid generation of animations that include spatial anti-aliasing, depth of field and motion blur. In our experience, motion blur in particular is vital to the verisimilitude of a moving liquid. Motion blur causes the details in the frame to be smeared, making it more difficult to perceive the appearance of the liquid. Motivated by this, we forgo the inclusion of difficult to compute effects such as accurate multiple scattering (e.g., as computed by path tracing) and high order reflections and refractions, so that motion blur can be rendered instead. Overall, this trade off results in greater realism and is orders of magnitude faster than ray-based solutions.
Figure 5.13: Single frame from an animation of spinning cubes rendered using (a) no effects, (b) anti-aliasing only (64 samples per pixel), (c) depth of field only (64 spp), (d) box-filtered motion blur only (64 spp), (e) tent-filtered motion blur only (64 spp), and (f) all effects (1024 spp).

5.6.2 Multisampling

In our framework, multisampling is achieved by rendering the scene in multiple passes and averaging the results using a double-precision, software accumulation buffer. We do not use OpenGL accumulation buffers since they are not widely supported in hardware and we wish to avoid implementation dependent problems such as loss of resolution due to insufficient numerical precision. Anti-aliasing, depth of field and motion blur are supported as described in [HA90], and sampling rates can be set independently for each (see Figure 5.13).

Anti-aliasing is achieved by offsetting the position of the view frustum by random, subpixel amounts, determined using stratified sampling. An alternative approach for full scene anti-aliasing is to render to a higher resolution framebuffer and downsample
the result. While this substantially reduces the amount of vertex processing that has to be done, it has some big disadvantages. Firstly, it requires additional processing each frame to downsample the framebuffer, and secondly, it requires $N$ times the amount of framebuffer memory, where $N$ is the number of samples per pixel. This severely limits the number of samples that can be taken. For offline rendering we prefer instead to use multi-pass anti-aliasing to allow arbitrarily high sampling rates.

Depth of field is computed in a similar way to anti-aliasing. In this case, the viewpoint is jittered about a focal point. The random sample points used for jittering are computed by warping a set of points selected using stratified sampling so that they are uniformly distributed over a disk. This is done using Shirley’s concentric disk mapping technique [SC97]. The disk here corresponds to a circular viewing aperture. By choosing sample points according to different distributions, polygonal and other aperture shapes can also be modelled. The main advantage of multi-pass depth of field over alternative, image-based approaches is that occlusions and transparency are handled correctly. This is especially important for liquid rendering.

Finally, motion blur is supported by averaging images offset in time. We allow the user to record their movement through the environment in real-time so that it can later be played back. This provides a quick and easy way to script a camera’s motion when producing animations. In such a case, motion blur is correctly handled by smoothly interpolating the camera path between saved keyframes. Also, given the convenience of performing chronological scene updates, we do not require the ability to evaluate animated objects at arbitrary times. Instead we only require that the scene can update itself by a specified time-step. This time-step is usually a constant value that provides a uniform sampling of time. However, this assumes that the camera shutters open and close instantaneously and does not correspond well with the rate at which the aperture size changes in real cameras. Stephenson [Ste05] proposed an approach that takes into account the finite time it takes shutters to open and close by weighting the sample images non-uniformly using a tent filter to match the uneven distribution of exposure times. Of course, the same result can be achieved by instead warping the uniformly distributed subframe time samples in $[0, 1]$ by the same filter:

$$
tent(x) = \begin{cases} 
\frac{1}{2} \sqrt{2x} & \text{if } x < \frac{1}{2} \\
1 - \frac{1}{2} \sqrt{2 - 2x} & \text{otherwise}
\end{cases} \quad (5.10)
$$
Stephenson observed that the tent filter gives aesthetically improved results with significantly less blurring than the box filter as shown in Figure 5.13. As with depth of field, multi-pass rendering has the advantage of computing physically correct 3D motion blur and properly accounts for temporal changes to the lighting environment and view-dependent effects such as transparency, reflections and refractions, all of which are important for rendering motion blurred liquids. Other algorithms such as REYES [CCC87] by Cook et al. do not handle these cases correctly since they only sample lighting once per frame, nor do image-based techniques which only have information on the surfaces closest to the camera.

5.6.3 Liquid Shaders

The OpenGL Shading Language (GLSL) is used to implement real-time, per-pixel shading of liquid surfaces defined by triangle meshes. Modern GPUs are very flexible and most of the logic from Section 5.2, including absorption and single scattering, translates well to fragment shaders. Although there have been attempts to map Whitted style raytracing to the GPU, this is a difficult problem, especially without support for recursive function calls. We simplify this issue by only considering first-order reflection and refraction rays. This is a reasonable approximation in many cases. For example, in the case of an ocean surface, second-order rays will only appear when wave amplitudes are large and the light strikes the surface at close to grazing angles. As in Section 5.3.1, we compute the colour of a point on the water surface as the Fresnel-weighted sum of the reflected radiance and the transmitted radiance. We index a cube-map representing distant illumination to approximate the light arriving from directions of reflection and refraction. An additional Phong illumination term may be added to the reflected colour to account for specular highlights caused by, for example, direct sunlight. This could be handled implicitly by including the light sources in the cube-map. However, explicit treatment of this light is preferable to provide control over its appearance and to avoid aliasing from undersampling the high frequency spikes that correspond to the light sources in the cube-map.

In order to filter the transmitted radiance to account for the effects of absorption and scattering, we need to compute the distance $s$ that the refracted ray travels within the liquid. Given this distance, it is trivial to compute the attenuated radiance in the
fragment shader by solving Equation (5.7) and by using Equation (5.8) to approximate in-scattered light. In [Tes04], the path length $s$ is ignored and instead a constant value is used for the light which exits the surface at all points. This simplification does not consider the dependence of transmitted radiance on either the liquid geometry or the viewing direction. It is only suitable for very optically thick media since it assumes that light is filtered by the same amount irrespective of the distance it travels. For arbitrary geometry, it is very difficult to compute $s$ in a fragment shader. Instead, these distances need to be computed on the CPU for each vertex of the liquid mesh. This requires a single ray intersection test per vertex. The GPU can then interpolate the vertex data over fragments for use in the liquid shader. Fortunately, this expense can be avoided for bodies of liquid such as oceans that can be represented as a heightfield mesh. Here, the values of $s$ can be computed analytically at the fragment level by introducing some simplifying assumptions.

![Figure 5.14: Values required to compute the distance $s$ that the refracted ray travels between the water surface and the ground.](image)

Figure 5.14 shows the geometric configuration for a point on a heightfield surface. The $y$-coordinate, $water_y$, of the current fragment in world space coordinates corresponds to the height of the water, and the $y$-coordinate of the ground is $ground_y$. We assume that the ground is flat for simplicity. For variable height ground, $s$ could
be computed directly in the fragment shader by iteratively searching for the intersection point between the refraction ray and the ground as defined by a heightfield texture [BD06]. Referring to Figure 5.14, we see that \( \cos \theta = (r_x, r_y, r_z) \cdot (0, -1, 0) = \frac{(\text{water}_y - \text{ground}_y)}{s} \), where \( (r_x, r_y, r_z) \) is the normalized refraction vector. Therefore, \( s = \frac{(\text{water}_y - \text{ground}_y)}{-r_y}. \) Notice that \( s \) is dependent on the heightfield geometry, unlike in [Tes04] where it is assumed that the water surface is flat. We can now plug \( s \) into Equations (5.7) and (5.8) to compute the final transmitted radiance. Note that, since we do not perform recursive raytracing, \( L_0 \) in Equation (5.7) corresponds to the exitant radiance at the ground. This can either be specified as a constant colour, or a texture map that is indexed by the refraction vector. The left panel of Figure 5.15 shows the effect that in-scattering of intense red light has on ocean appearance (other effects such as specular highlights have been disabled for clarity). The red light gives the ocean a subsurface scattering quality that helps to convey it as a participating medium.

In Section 5.3.3, we rationalized the decision to ignore attenuation of in-scattered light as it travels from the light source and scatters into the direction of a refracted ray. Accounting for this would require ray marching to determine how much attenuation occurs at each point on the refracted ray. Even then, it would not be possible to handle refraction of light at the liquid surface. However, for heightfields we can handle these cases without requiring ray marching. By assuming that the ocean sur-
Figure 5.16: Values required to determine how in-scattered light is attenuated between the refracted ray and the light source.

face is flat between the light source and the refracted viewing ray, we can derive a new expression for in-scattered light. This is a reasonable assumption for computing underwater attenuation factors since, for most ocean conditions, the wave amplitudes will be small compared to the ocean depth. Figure 5.16 illustrates the geometry involved. The quantity labelled $x$ corresponds to the variable of integration in the in-scattering integral that was first introduced in Equation (5.7). Also, $(l_x, l_y, l_z)$ is the normalized light vector that is computed by refracting the original light direction at the water surface. The main quantity of interest that we require is $l$, the distance over which attenuation occurs between the light source and the refracted ray at each point on the integral. To compute this, we first need an expression for $d$. Since $\cos \theta = (-r_x, -r_y, -r_z) \cdot (0, 1, 0) = d/(s - x)$, we get $d = -r_y (s - x)$. Also, since $\cos \phi = (-l_x, -l_y, -l_z) \cdot (0, 1, 0) = d/l$, we get $l = d/-l_y = -r_y (s - x)/-l_y$. Referring back to the integral in Equation (5.7) and assuming, as we did in Section 5.3.3, that the phase function and incident radiance are constant over path length $s$, we can
reformulate the in-scattering integral as:

\[ \sigma_s p(\theta) \int_0^s L_i(\bar{\omega}) e^{-(\sigma_a + \sigma_s)(s-x)} dx = \sigma_s p(\theta) L_i(\bar{\omega}) \int_0^s e^{-(\sigma_a + \sigma_s)(1+s-x)} dx \]

There is now an additional exponential term, multiplied by \( L_i \), to account for the attenuation that occurs over distance \( l \). Finally, we can expand \( l + s - x \) in the above equation to \((1 + \frac{r_y}{l_y})(s - x)\) and solve the resulting integral to give:

\[ \frac{\sigma_s}{(\sigma_a + \sigma_s) \left(1 + \frac{r_y}{l_y}\right)} p(\theta) L_i(\bar{\omega}) \left(1 - e^{-(\sigma_a + \sigma_s)(1+\frac{r_y}{l_y})s}\right) \]

This equation is very similar to Equation (5.8), but it now depends on the refraction and lighting directions. The results produced by these two equations are compared in Figure 5.15. Notice that the left picture is a lot brighter due to the absence of attenuation between the refraction vector and the light source. This can sometimes create the illusion that the light source is underneath the water surface which is less obvious when attenuation is handled correctly. Finally, Figure 5.17 shows frames from an animated ocean surface with a moving ball that is reflected and refracted in the water. When multi-pass depth of field is disabled, this animation runs in real-time on a 1.6 GHz Intel Centrino processor with an ATI Radeon 9700 graphics card.
5.7 Rendering Whitewater

5.7.1 Overview

In the preceding sections we discussed how to render liquids represented by a well-defined surface such as a triangle mesh. However, when liquid is in violent motion it can break apart and disperse into many disjoint components, which may be as small as droplets. This is particularly evident in the case of rough seas or a fast flowing river where flying spray, hanging mist and floating foam all have the quality of being independent of the bulk water volume, thus setting these components apart from the appearance of the rest of the water. In theory, a smooth mesh could be used to represent and render the whitewater components as a homogeneous liquid. In practice however, memory and computational limitations make it very difficult to capture and manipulate these high frequency elements at sufficient resolution using meshes.

A volumetric representation is one possible alternative. Here, a density field sampled on a grid represents the small scale liquid. Each grid cell stores a scalar density value that provides a measure of how much liquid resides in that cell. This defines an inhomogeneous participating medium that requires a more advanced lighting simulation to render. One of the advantages of this approach is that the memory and processing requirements can be easily controlled by changing the grid resolution. This directly corresponds to the amount of detail represented. Unfortunately, uniform volumetric grids are very demanding in terms of memory since they scale with the cube of the number of cells per grid side. For example, a grid of resolution 200x200x200 with 32-bit densities would require 30.5 MB, while a 400x400x400 grid would require 244 MB. Clearly, trying to represent a droplet in a stormy ocean expanse covering a large area would require a huge amount of memory. Multiresolution storage could be used to alleviate this, but memory requirements would still be very high. Typically, volumes are better suited for representing "smoother", continuous fluid media such as gas, smoke and fire where the constituent particles are too small to be individually discernible.

A third alternative is to use particles to represent the whitewater. In this case, droplets in an ocean scene can be represented as individual 3D points. This highlights one of the biggest advantages of particle-based methods - the fact that memory is
only expended where the liquid exists and is not wasted on empty regions of space. Generally, particles are better suited than volumes at representing small scale, splashing detail since the detail is not limited by spatial resolution, only by the number of particles that can be stored and processed. This is particularly important when the liquid is animated since a very high resolution grid would be required to represent the same detail that can be achieved through the movement of a collection of particles. Consider for example the number of particles that can be stored in the same amount of memory as a 400x400x400 grid. Assuming that each particle is defined by a 12-byte point, 21 million particles can be stored. Of course, if we wish to draw that many particles every frame, there are other issues that need to be addressed. Firstly, in order to keep the per particle memory footprint as small as possible, we will assume that all whitewater components are represented as a list of 3D points and nothing else. Although it would be trivial to support additional per particle attributes such as colour, opacity or size, we are dealing with such high particle counts that any extra attributes will add a substantial memory overhead. For the same reason, we wish to keep per particle processing to a minimum. Drawing and shading the particles are the main challenges in this respect and will be discussed in the sections that follow.

5.7.2 Particle Rendering

We assume that particles are spherical in shape and that, on average, imaged particles will appear less than a pixel in size. There are many ways that spherical particles can be rendered. Approaches based on ray-casting are generally a bad idea since severe aliasing can occur due to undersampling of particles that project to subpixel areas on screen. When this happens, moving particles appear to scintillate which requires very high sampling rates to avoid. A better approach is to explicitly render each particle one at a time. Low resolution triangle meshes representing spheres could be used in conjunction with silhouette clipping [SGG+00] so that edges appear round no matter how close the sphere is to the camera. While accurate, this solution will not be fast enough to render millions of particles at reasonable speeds. Another alternative is to use texture-mapped, or normal-mapped, billboarded geometry to represent spheres [TJ06, RJG06]. While rendering quadrilaterals is a lot faster than fully geometric spheres, the transformations required to correctly draw a spherical particle on screen
are expensive to compute. The standard transformation rotates the billboard so that it faces the camera’s position. However, this alone is not enough to capture the true perspective projection of a sphere. It is also necessary to scale the billboard to account for its position relative to the camera, as shown in Figure 5.18. The scale factor is 

\[(d/r) \tan\left(\sin^{-1}\left(\frac{r}{d}\right)\right)\]

where \(r\) is the radius of the sphere and \(d\) is its distance to the camera. This has a subtle effect on the sphere’s appearance and its absence would be difficult to notice, but even without this, the billboard transformation is expensive to compute for each particle. The approach we use is to ignore these transformations and render all particles as anti-aliased, screen-aligned disks of uniform colour, irrespective of their size in screen space. Note that a level-of-detail scheme could easily be used to provide accurate rendering at all scales by choosing one of the above rendering methods based on the projected size of each particle. The main advantage of this is that non-spherical particles (such as grains of sand) could be rendered accurately. For spherical particles however, we achieve adequate results using circular disks and do not consider a level-of-detail scheme to be necessary.

In order to render the particles as disks, we initially developed a software renderer that uses a two-pass algorithm. In the first pass, all of the particles are projected into screen space. If the size of a resulting disk is less than one pixel, its opacity is scaled by the fraction of the pixel area it covers. The disk is then rasterized into fragments.
and each fragment is added to a list corresponding to the pixel with which it coincides. Anti-aliasing is achieved by softening the edges of the disk. This is done by applying a cubic smoothing kernel to the fragment opacity value to provide a soft fall-off. At the end of the first pass, there is a list of fragments for each pixel. In the second pass, the colour of each pixel is determined by sorting its list of fragments based on their depth values, and compositing them in back-to-front order. While this approach achieved the desired results, it was also very slow and memory intensive. For example, to render a data set consisting of 250,000 particles on a 1.6 GHz Intel Centrino processor required 16.8 seconds with depth-sorting (7.1s for the first pass and 9.7s for the second) and 8.5 seconds without (6.6s for the first pass and 1.9s for the second). The peak memory usage was 280 MB.

For comparison, we attempted to render the same data set using OpenGL GL_POINT primitives. On an ATI Radeon 9700, this data could be rendered at 12.5 FPS when the particles were sorted every frame, and 26.4 FPS when sorting was omitted. The total memory required was only 11.3 MB. Motivated by this, we have adopted the use of OpenGL point sprites for particle rendering. The necessary transformations, including accounting for the variation in disk size as distance from the camera changes, and anti-aliased rasterization are handled entirely by the GPU. Furthermore, the use of OpenGL fits in nicely with the multisampling framework we described in Section 5.6.2 to allow inclusion of effects such as depth of field and motion blur.

Since the particles we wish to render represent whitewater, they need to be rendered transparently using OpenGL alpha-blending. This requires them to be drawn in sorted order so that they can be composited correctly over the background and to properly account for the effects of absorption and scattering. Considering that our lighting simulation varies smoothly over particles (see Section 5.7.3), we originally experimented with using an approximate sort by mapping the particles to a uniform grid and iterating through the cells in back-to-front order from the camera. However, since simulated particles will generally maintain their relative order to some degree between frames, a better approach is to use a temporally coherent sorting routine. We use the templated STL function std::sort\(^2\), defined by the C++ standard, to sort our particles which are stored in an std::vector object. The sort orders the particles based on their distance

\(^2\)Our particular implementation of STL uses the Introsort algorithm [Mus97] whose average and worst case complexity is \(O(N \log N)\).
from the camera. This is done by defining a comparison operator that compares the lengths of the projections of each particle onto the normalized, camera direction vector. To give an example of how temporal coherence makes this very fast, we measured the time taken to perform the sort on a one million particle data set, viewed from an orbiting camera. For the first frame, the sort took 0.66 seconds, whereas the average time for subsequent frames was only 0.2 seconds.

5.7.3 Particle Shading

Lighting Inhomogeneous Media

Figure 5.19: From left to right: slice-based volume rendering; particle rendering with self-shadowing; particle rendering without self-shadowing; particle rendering with scattering parameters chosen to resemble liquid foam.

Although spray and foam reflect and refract light in the same way as other liquids, specular lighting off individual particles is usually not very noticeable and will be ignored\(^3\). Instead, the aggregate effect of light reflecting and refracting between particles will be modelled. This results in the characteristic diffuse, whitish appearance of spray and foam that is primarily due to the multiple scattering and attenuation of incident light.

\(^3\)There are two exceptions that occur in very particular circumstances and these will be considered in Section 5.7.4.
Since particles are points that do not have explicit normal vectors, it is generally very difficult to light them realistically. Instead, most applications make the approximation that all particles are uniformly lit and rely on colour variation or texture-maps to convey shape and shading. Since our whitewater particles are all of identical size, shape and colour, it is not possible to convey the volumetric nature of the liquid without lighting them realistically. Instead of performing a full simulation of light bouncing between individual particles, we simplify the problem, as we did in Section 5.3.3, and only consider light that is in-scattered from one or more directional light sources. Each particle is rendered by computing how much light reaches it, after being attenuated by the other particles through which it passes. In this way, particles are shaded based on occlusion and, as a result, they appear to cast shadows on one another.

Figure 5.19 shows a comparison of the Stanford Buddha model rendered using different techniques. The first image was rendered by a traditional, slice-based volume renderer written in OpenGL. The second image shows the same model rendered as one million particles and it clearly exhibits far greater detail than the volume rendered image. The data set used here was produced by converting the triangle mesh into a low resolution signed distance field on a grid and rejection-sampling random points chosen from the model's bounding box to find a collection that lies within the zero level set. We use this data set in all of the examples in this section rather than data representing whitewater because it is a familiar object with recognizable features. This makes it easier to perceive if lighting and shadowing are realistic. Note that this image was lit without the use of any normal vectors. The shadows and Lambertian surface appearance arise solely from self-shadowing. In comparison, the third image shows the same data rendered with absorption but without in-scattering. The lack of self-shadowing makes it impossible to perceive the intricate surfaces defined by the particles.

For the purposes of computing light interaction with the particles, we treat them collectively as an inhomogeneous participating medium and we relate the density of particles $\rho(x)$, at any point $x$, to the probability of attenuation at that point by a scale factor $C$. Although this constant could be determined automatically from the particle radius, it is more convenient to decouple these values to provide fine control over the amount of self-shadowing that occurs. The volume rendering equation for homogeneous media, Equation (5.7), was presented in Section 5.3.3. The corresponding equation that
takes the spatial variability of absorption and scattering coefficients into account is as follows:

\[ L = L_0e^{-\tau(x)} + \int_0^s \left( e^{-\tau(s-x)}\sigma_s(x) \int_{\Omega_t} p(\theta) L_i(x, \bar{\omega}) d\bar{\omega} \right) dx \]  

(5.13)

where the optical thickness \( \tau(x) \) is a measure of the transmissivity of the medium from starting point \( x \) to the end of the segment:

\[ \tau(x) = \int_x^s (\sigma_a(t) + \sigma_s(t)) dt = \int_x^s \rho(t) dt \]  

(5.14)

In order to compute correct lighting, Equation (5.13) needs to be evaluated for every particle. As in Section 5.3.3, we only compute single scattering from directional light sources under the assumption that the scattering coefficient is constant. Furthermore, to avoid evaluating the Henyey-Greenstein phase function for each particle, we assume an isotropic phase function. This implies that, in the absence of attenuation, each particle will be lit identically. We also assume that the particles do not bend light, i.e., they have an index of refraction of 1. This important simplification means that light will take a straight line path through the particles. This allows us to compute attenuation along the viewing direction by simply blending the particles correctly into the framebuffer, without requiring ray tracing.

**Preprocessing**

In order to compute accurate self-shadowing of particles, one approach would be to treat all particles as spheres. The extent to which each particle is occluded from light sources could then be calculated by casting a ray in the direction of lights and accumulating the path lengths through all intersected spherical particles. This would give an accurate measure of how much light reaches the particle and allow for very high resolution shadows. However, a better approach that sacrifices lighting resolution for speed is to use auxiliary, uniform grids to help calculate illumination at a set of discrete points in space. This data can then be interpolated to the position of a particle to determine how much light it receives.

The choice of grid orientation is an important issue. Rasmussen et al. [RNGF03] use a voxel grid aligned with the view frustum to accelerate camera ray traversals through
the grid. Although it may be tempting to use a view dependent approach to cull particles that lie outside the view frustum, this may introduce inaccuracies since culled particles can still influence the light that reaches visible particles. Another approach, used in [HBSL03] and [BMC05], is to align the grid with the lighting direction to accelerate the propagation of light through the grid. A disadvantage of this approach is that separate grids are required for each light source. Furthermore, as discussed below, we can perform very fast light propagation even without a light oriented grid. We choose instead to align the grid with the world coordinate system. Unlike the other alignments, this avoids expensive coordinate transformations when performing operations such as mapping the position of a particle to its corresponding grid cell, or interpolating data within the grid.

In order to evaluate the in-scattering integral of Equation (5.13), we precalculate the optical depth from the center of each grid cell to the grid boundary in the direction of the light source. This is calculated from Equation (5.14) which first requires the particle density $\rho$ at each cell. There are many ways these densities could be computed. We use the fast and simple approach of treating each particle as a cell-sized cube and adding volume-weighted density contributions to each cell with which the cube overlaps. The density of the cell centered at $(x, y, z)$ can be written as $\rho(x, y, z) = \sum (\Delta - d_x) (\Delta - d_y) (\Delta - d_z)$, where $\Delta$ is the grid spacing, the sum is over all particles $(p_x, p_y, p_z)$, and $d_x = |p_x - x|$, if $|p_x - x| < \Delta$, otherwise $d_x = 0$. $d_y$ and $d_z$ are defined similarly. If a smoother density field is desired, the grid values can be filtered as appropriate.

The optical depth at each cell, relative to a directional light, can now be computed by accumulating the densities along rays from the centre of each cell to the grid boundary. This is described by Equation (5.14) which can be solved numerically using discrete summation. More specifically, the integral can be evaluated by traversing the grid in the direction of the ray and adding up the lengths of the path through each cell, multiplied by the cell’s density. Although very fast algorithms have been developed to trace rays through grids ([AW87]), we exploit the fact that we are dealing with directional lights to accumulate densities in a single sweep through the grid. Consider, for example, light traveling through the grid from left to right. In this case, we can traverse the grid cells in left-to-right order and compute the optical depth at each cell by adding the current cell’s density to the cumulative density of the cell to its left.
Figure 5.20: The optical depth at each grid point is calculated in a single sweep through the grid. To account for an arbitrary lighting direction, densities need to be accumulated in four different directions. The first four images show the lighting results when each of these directions is considered by itself and the last image shows the result when all directions are considered together.

Since light is traveling from the left, we do not need to consider the density field to the right of the cell whose optical depth is being calculated. This can be generalized so that, for any given light direction, we can iterate through the cells in an order which ensures that cells closer to the light are visited before cells that are farther away. Since grid traversal requires 3 nested loops, one for each coordinate direction, there are 6 different ways in which these loops can be ordered. Additionally, for each of these orderings, there are 8 different ways through which the grid can be iterated depending on whether each loop advances forwards or backwards through the cells. This gives a total of 48 different orderings. By iterating through the grid using an ordering that is selected based on the light direction, we can compute the optical depth of each cell by taking a convex combination of cumulative densities from neighbouring cells that have already been visited. The weights are chosen based on the angles between the light direction and each coordinate axis. The first four images of Figure 5.20 show the lighting results when accumulating densities from neighbouring cells in each one of the four directions that need be considered, and the last image shows the result of taking a convex combination of cumulative densities from all four neighbours.
Volumetric Illumination

Figure 5.21: Since geometry and lighting are decoupled, grid resolution does not have a large effect on lighting quality, although shadows become increasingly blurry. The resolutions from left to right are: 20x40x20, 50x100x50, 100x200x100 and 200x400x200.

To render the particles, their colour and alpha values need to be determined. The colour of a particle is equal to the in-scattered light that reaches it, multiplied by the particle’s colour, which we treat as a constant. We can evaluate the in-scattering integral of Equation (5.13) to calculate the light at the centre of each grid cell. This is simply the light intensity, scaled by the constant scattering coefficient, and multiplied by the exponential decay factor that is calculated from the optical depth at the cell centre. To calculate the light at arbitrary points within the grid we interpolate (trilinearly or tricubicly) the light values defined at cell centres. Figure 5.21 shows the effect that grid resolution has on lighting quality when using trilinear interpolation. Notice that self-shadowing still conveys a lot of detail even when very low grid resolutions are used. This is further exemplified in Figure 5.19 which compares a sliced-based volume render with a particle render where lighting in both images is calculated using a grid of the same resolution. Clearly, a volume renderer would require a density grid of far higher resolution to reproduce the detail evident in the particle render.

As well as accounting for the attenuation of light on its path to each particle, we also need to consider exponential attenuation in the viewing direction. This is achieved
Figure 5.22: Attenuation through an inhomogeneous liquid medium is equivalently computed by considering attenuation through its constituent, homogeneous parts.

using OpenGL alpha-blending to composite the particles in back-to-front order on top of the background. The two issues that need to be addressed are the alpha values used when drawing particles and choice of OpenGL blending functions.

Traditional volume rendering techniques can be used to render grids of density values, such as the one we use to compute in-scattering. For example, the slice-based approach used in Figure 5.19 renders axis-aligned cross-sections of the volume using an OpenGL \texttt{GL_QUAD} primitive for each grid cell. Since each quad represents a volume with a thickness equal to the cell size, the alpha values of the quad are set based on the cell’s density value. However, in the case of particles, it would be incorrect to determine alpha values from interpolated densities. Although we use the inhomogeneous volume for lighting calculations, we do not render it directly. It is more appropriate to treat each particle as a homogeneous medium. As shown in Figure 5.22, if we consider light that passes through two disjoint, homogeneous volumes of liquid, the attenuated light is given by \( L = L_0 e^{-(\sigma_a + \sigma_s) d_1} e^{-(\sigma_a + \sigma_s) d_2} \). Each additional volume of liquid attenuates the light a little more based on the path length through the volume. For the case of spherical particles, we assume that this path length is always equal to the diameter of the particle and therefore we attenuate light along the viewing direction by the constant amount \( e^{-(\sigma_a + \sigma_s) d} \). While it would be possible to use a texture map to store the (refracted) path lengths as a function of position on the particle disk, this is unlikely to make any perceptible difference in image quality.

If \( L_0 \) is the colour of a pixel in the framebuffer and we wish to draw a particle on top of this pixel, the new colour is given by \( L = L_0 e^{-(\sigma_a + \sigma_s) d} + L_{in} \), where \( d \) is the diameter of the spherical particle and \( L_{in} \) is the attenuated light that arrives at the
particle. Note that, as we did for density, we decouple particle size from its opacity and treat $e^{-(\sigma_s+\sigma_t)d}$ as a user-defined, constant colour. This equation consists of a multiplicative component (a component-wise multiplication since the attenuation factor is a colour) and an additive component. We need to set the OpenGL source and destination blending factors to account for this. This requires sending two colours to the blending unit. On recent hardware this is possible by using the \glBlendTime\ OpenGL extension, which allows the specification of an additional constant colour to be used in the blending equation. Assuming the availability of this extension, the proper result can be achieved by setting the source and destination factors to \glOne\ and \glConstantColor\ respectively, and specifying the attenuation colour using \glBlendTime\ and the in-scattering colour, $L_{in}$, using \glColor3f\. Unfortunately, our tests indicate that a floating-point framebuffer would be necessary to blend a large number of particles with sufficient precision to compute exponential attenuation. Since floating-point framebuffers are not supported on our development hardware (an ATI Radeon 9700), we considered the alternative of using \glSrcAlpha\ and \glOneMinusSrcAlpha\ as the blending factors. The main motivation for this is that these are the recommended blending factors to be used in conjunction with \glPointSmooth\ for anti-aliasing points. To see how this behaves, let us consider blending $N$ overlapping particles into the framebuffer where the initial framebuffer colour is $L_0$ and each particle has colour $L_{in}$, and alpha $\alpha$. Applying the OpenGL blending equation recursively $N$ times gives:

$$L = \alpha L_{in} \left[ 1 + (1 - \alpha) + (1 - \alpha)^2 + \ldots + (1 - \alpha)^{N-1} \right] + L_0 (1 - \alpha)^N$$

As desired, the background colour is attenuated at an exponential rate based on the number of attenuating particles. Furthermore, we can compute $\lim_{N \to \infty} L$ by applying the geometric series $1/(1 - x) = 1 + x + x^2 + x^3 + \ldots$ to get $L = L_{in}$. In this case, all of the background radiance is absorbed and all that remains is the particle colour, irrespective of $\alpha$, as expected. As the example images in this section show, this approximation to attenuation gives plausible results. The main disadvantage is that attenuation in the viewing direction is monochromatic since it is specified using the alpha channel.
Figure 5.23: Fake multiple scattering can be achieved by “blurring” the lighting information stored at each grid cell. From left to right, the number of smoothing passes using a 3x3x3 box filter is 0, 1, 5 and 20.

**Additional Effects**

As light travels through the volume to each grid cell it will scatter multiple times off of particles causing a “blurring” of the light. In the previous section we ignored this effect and only considered single scattering. However, given the amount of attenuated light that reaches each grid cell, we can approximate multiple scattering very crudely by spreading some of this light evenly into neighbouring grid cells. This uniform diffusion of light can be described by the heat equation, $u_t = \nabla^2 u$, where $u$ is the light field. As noted in [Ebe02], this is equivalent to Gaussian blurring. However, we can perform convolution using a much simpler box filter and achieve almost identical results. Figure 5.23 shows the results of softening the light to varying degrees using a 3x3x3 box filter. This simple approximation assumes that the medium is isotropic, homogeneous and non-attenuating. It might be possible to achieve a better, empirical approximation by using a non-uniform filter where the kernel coefficients are weighted by the particle densities, although we have not tried this. Of course, accurate multiple scattering could be computed in this context by solving a diffusion equation numerically [SF95] or using the hierarchical, dipole diffusion approximation [JB02]. However, this would be significantly slower than the approach suggested here.
Another effect we can add to enhance appearance is ambient, image-based lighting using a low frequency environment map (e.g., of the surrounding sky). This is a cheap way to add more spatial colour variation to the particles. To compute this, we first require a direction vector for each particle. This is defined as the normalized gradient of the density field, $\nabla \rho / |\nabla \rho|$. We precompute this vector at each grid cell using central differencing of grid densities and apply box filtering (if necessary) to avoid abrupt changes between nearby directions. At render time, we compute a direction vector for each particle by trilinear interpolation and use this to index the environment map. The resulting colour is then added to the particle colour computed from in-scattering. Note that, to avoid a large performance overhead, we ignore attenuation of the environment lighting as this would require tracing rays through the density grid to compute optical depths in arbitrary directions.

![Figure 5.24: Frames from an animation of a particulate Buddha being entrained into a simulated tornado. Achieving a similar level of detail with purely volume-based or mesh-based approaches would be very difficult.](image)

Finally, in order to achieve realistic results, it may be necessary for particles to cast shadows onto other objects, or have shadows cast onto them. For any point in the scene, we can determine how much light reaches it after being attenuated by the particles as follows. If the point is inside the grid, we simply use the in-scattered light that is
computed in the same way as is done for particles. If the point is outside the grid, we trace a ray from the point in the lighting direction and determine whether it intersects the bounding box of the grid. If it does, then the attenuated light is determined as before using the intersection point. Figure 5.24 shows an example of a motion-blurred shadow being cast onto a ground plane that results from light being attenuated by one million particles. To handle the case of other objects casting shadows onto the particles, we modify the step that precomputes the grid of optical depths so that, instead of using constant light intensities, the light that enters the grid is determined by casting rays from the necessary boundary grid points in the direction of the light to determine if it is blocked by other objects or attenuated by other participating media (such as water waves).

**Performance**

<table>
<thead>
<tr>
<th>Grid resolution</th>
<th>20x40x20</th>
<th>50x100x50</th>
<th>100x200x100</th>
<th>200x400x200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute densities</td>
<td>0.75</td>
<td>0.75</td>
<td>0.79</td>
<td>0.92</td>
</tr>
<tr>
<td>Compute optical depths</td>
<td>0.01</td>
<td>0.03</td>
<td>0.11</td>
<td>0.96</td>
</tr>
<tr>
<td>Sort particles</td>
<td>0.36</td>
<td>0.35</td>
<td>0.35</td>
<td>0.35</td>
</tr>
<tr>
<td>Draw particles</td>
<td>1.79</td>
<td>2.23</td>
<td>2.47</td>
<td>2.31</td>
</tr>
<tr>
<td>Total</td>
<td>2.91</td>
<td>3.37</td>
<td>3.72</td>
<td>4.54</td>
</tr>
</tbody>
</table>

Table 5.1: Effect of grid resolution on performance. All times are in seconds.

<table>
<thead>
<tr>
<th>Number of particles</th>
<th>125,000</th>
<th>250,000</th>
<th>500,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute densities</td>
<td>0.12</td>
<td>0.24</td>
<td>0.47</td>
<td>0.92</td>
</tr>
<tr>
<td>Compute optical depths</td>
<td>0.92</td>
<td>0.94</td>
<td>0.93</td>
<td>0.96</td>
</tr>
<tr>
<td>Sort particles</td>
<td>0.03</td>
<td>0.07</td>
<td>0.16</td>
<td>0.35</td>
</tr>
<tr>
<td>Draw particles</td>
<td>0.23</td>
<td>0.50</td>
<td>1.04</td>
<td>2.31</td>
</tr>
<tr>
<td>Total</td>
<td>1.30</td>
<td>1.75</td>
<td>2.60</td>
<td>4.54</td>
</tr>
</tbody>
</table>

Table 5.2: Effect of particle count on performance when using a 200x400x200 grid. All times are in seconds.

We evaluated the performance of the complete particle rendering pipeline running on a 1.6 GHz Intel Centrino laptop with 1 GB of RAM and an ATI Radeon 9700 graphics card. The test scene uses the Buddha data set and is configured as shown in Figure 5.19. Table 5.1 shows how the run-time for various parts of the algorithm scales
with the grid resolution. As expected, the times for computation of optical depths increases with the number of grid cells. The other three steps instead scale with the number of particles. This is shown in Table 5.2 where the times for computation of optical depths remain constant while the other times increase linearly with the particle count. Our system makes it practical to render very high particle counts at reasonable rates. In cases where more particles need to be rendered than can fit in RAM, it is possible to partition the particles into slices based on their distance from the camera, render each batch independently and, at the end, composite the results in back-to-front order.

5.7.4 Specular Effects

Overview

In the previous section, we considered how to account for ambient and diffuse lighting of particles. However, particles that represent water droplets can also exhibit strong specular effects caused by reflecting and refracting light. In this section we investigate how directional light interacts with a spherical, dielectric particle and present a new method for lighting particles that exhibits the following effects:

**Highlights**: occur when light reflects directly into the direction of the observer;

**Rainbows**: occur when the observer is between the particles and the light source;

**Glowing**: occurs when the particles are between the observer and the light source.

There has been a lot of recent work devoted to the problem of realistic rain rendering, in which different schemes have been employed to handle specular lighting of raindrops. Garg and Nayar [GN06] and Garg et al. [GKN07] perform raytracing of raindrop meshes in a preprocess to generate rain streak texture maps for use at render time. Tatarchuk and Isidoro [TI06] computes specular highlights using a traditional, local illumination model. While Rousseau et al. [RJG06] and Wang et al. [WLF+06] handle reflections of light sources implicitly using an environment map. All of these approaches compute per-pixel lighting over the entire visible surface of each raindrop. This requires significantly more processing than rendering particles as points, which are mostly subpixel in size. Because we only compute one colour per particle, we need
a way to decide whether a particle picks up a highlight or not. Usually a tiny percentage of particles in whitewater create a perceptible reflection of the light source. These particles, although few in number, stand out as being much brighter than the rest. Both Peterson [Pet03] and Iversen and Sakaguchi [IS04] handle this effect by simply selecting a completely random subset of particles each frame and rendering them extremely brightly. Although it is difficult to say whether or not choosing the “correct” subset of particles will make a perceptible difference in the realism of an animation, in this section we will investigate using a physically motivated method to probabilistically decide whether a particle should be highlighted or not, based on the relative positioning of the particle, camera and light source.

The rain rendering literature also ignores wavelength-dependent phenomena such as rainbow formation. Although Glassner [Gla96] and Gonzato and Marchand [GM01] propose methods to render solar halos that arise from the interaction between sunlight and ice crystals, the only work we are aware of in the field of computer graphics that covers rainbow rendering is by Brewer [Bre04], whose approach is similar to that which we will describe. They precalculate a look-up texture using MiePlot software [Lav06] and use this to texture a full-screen quad. They also use a grayscale moisture texture to modulate the brightness of the rainbow based on the air’s water content. Their method only handles the primary rainbow however and they do not consider any higher order
effects. Lastly, we are not aware of any approaches in which the glow that occurs when sunlight passes through droplets without reflecting internally is computed.

**Precomputation**

In order to understand how optical effects such as rainbows arise, we will simulate the interaction between directional light and a solitary, spherical particle. MiePlot software [Lav06] can compute these interactions using an approach based on Mie theory and wave optics. While complex effects such as polarization and wave interference can be accounted for with this approach, we will ignore these effects and instead consider the problem using ray optics alone. This will allow us to analyze the behavior of light at a low level.

We perform a Monte Carlo simulation of several million random photons striking a droplet that is modelled as a perfect sphere. Ideal specular reflection, refraction, and Fresnel reflectance (as described in Section 5.2.1) are accounted for but we ignore absorption and scattering. Each photon is assigned a random wavelength from the visible spectrum with equal probability (corresponding to white light) and we use the equation proposed by Quan and Fry [Hui97] to compute a unique index of refraction for each wavelength. The overall algorithm is very similar to the spectral path tracer we described in Section 5.4.1. The main difference is that, in this case, photons are traced from a directional light source instead of the camera. We assume that all photons travel along the positive x-axis, and choose the initial position of a photon on the y-z plane to be uniformly distributed over a disk [SC97] whose radius is the same as the droplet. This guarantees that all photons will strike the droplet. For each of these photons, we compute its outgoing direction after interacting with the droplet and compute the scattering angle $\theta$ (in $[0^\circ, 180^\circ]$) between the incident and outgoing directions. As described in Section 5.5.3, we map this angle to a table bin, where each bin spans a fraction of a degree, and store the photon. Notice that this is equivalent to computing a tabulated, wavelength-dependent phase function for the water droplet. At the end, we can convert the spectral energies, given by the collection of photons that fall within each bin, into RGB colours to be used at render time.
Figure 5.26: The amount of light that exits a water droplet varies depending on the scattering angles. This data was computed by simulating 9 million photons for each wavelength and recording the results in a table of 1,800 bins.

Analysis

Figure 5.26 illustrates the results of simulating collections of photons corresponding to red, green and blue wavelengths. Nine million photons were used for each collection and the plot indicates how many of those photons exited at each scattering angle. The image below the graph shows the perceived colour at each angle when photons of all wavelengths are considered together and mapped to RGB colours. By considering the images formed by photons that interact once, twice, three and four times with the droplet, a number of optical phenomena can be explained.

Figure 5.27 (a) shows the image that results from direct specular reflection off the surface of the droplet. No focusing of the light occurs, so it is relatively dim. As
expected, the amount of reflected light increases with the scattering angle and reaches a peak when light hits the sphere at grazing angles. This is explained by the Fresnel equations.

Figure 5.27 (b) shows the simulation results when only two interactions are considered. In this case, photons enter the droplet by refraction and subsequently exit by refraction. No internal reflection occurs and there is not enough dispersion to separate the incident white light. The result is a broad, specular lobe which is observable in nature as an intense glow when looking at droplets in the direction of the sun.

Figure 5.27 (c) illustrates the case of a photon reflecting once inside the droplet. This causes incident light to fold back on itself at a range of angles and gives rise to
the primary rainbow at a scattering angle of between 40° and 42°, depending on the wavelength of the photon. For example, the specular spikes corresponding to 650nm, 510nm and 475nm are seen in Figure 5.26. Notice that these peaks occur at different angles due to the wavelength-dependent nature of refraction and this results in the seven colour bands of the rainbow. Also notice that at angles smaller than 40°, the outgoing light does not disperse enough to give a rainbow effect and instead causes the interior region of the rainbow to appear brighter than the exterior. In addition, very little light can escape between approximately 42° and 51° and, as a result, this region appears particularly dark.

Finally, Figure 5.27 (d) shows what happens when light exits the droplet after two internal reflections. Here the light is focused around 51° which forms a secondary bow. Notice that the colours of this bow are reversed and that the spacing between the peaks in Figure 5.26 is greater than in the primary bow. This is due to additional dispersion caused by the second reflection. This bow is also a lot dimmer than the primary. In fact, bows that result from increasing numbers of reflections appear fainter because with each bounce, some of the light escapes due to refraction. For example, a tertiary bow is barely visible around 138° in the colour plot of Figure 5.26. Although this bow is relatively bright, it is difficult to perceive because it is washed out by the underlying glare from the light source. Tertiary and higher order bows are practically impossible to perceive in nature.

Rendering Rainbows and Glowing

In order to use the results of the precomputation stage at render time, we perform the following additional steps after computing the particle colour using the approach described in Section 5.7.3. First, we compute the scattering angle between the lighting direction and the vector from the camera position to the particle position. We use this angle to index into the precomputed colour map shown in Figure 5.26. This gives us the RGB intensity of light that is scattered through the particle in the viewing direction. We modulate this colour by the in-scattered light that reaches the particle and add the result to the particle’s colour. An example result that was computed by tiling the image plane uniformly with 50 million particles is shown in Figure 5.28. The image shows a number of optical phenomena that are observable in real life (see Figure 5.25).
These include the bright region inside the primary rainbow, the dark band between the primary and secondary rainbows, and the inverted spectrum of the secondary rainbow.

**Rendering Whitewater Highlights**

As discussed earlier, in order to capture the effect of specular highlights in whitewater, we need a method to determine whether or not light is directly reflected or refracted in the direction of the camera. In the approach just described, specular highlights are handled implicitly by adding values from the colour map to the particle colour. However, this deterministic approach is only a valid approximation when the particles are very small spheres and inter-scattering between particles does not play a significant role in their appearance. It works well in cases such as ocean mist, spray from a garden hose, or raindrops.

In contrast, whitewater does not generally form coherent patterns of highlights and we cannot make the assumption that droplets are independent of one another. In fact, whitewater usually has a highly intricate geometric structure due to coalescing droplets and this causes most of the incident light to be scattered incoherently. Only a small percentage of particles reflect or refract light directly towards the camera. We therefore choose such particles stochastically by using a grayscale version of the colour map from Figure 5.26 to define the probability that a particle will pick up a highlight at each
scattering angle. Furthermore, we add a fixed constant to the map so that there is a small probability of a highlight occurring at all scattering angles. Finally, the map is scaled by a constant to provide artistic control over the number of particles that give rise to highlights. Notice that although this approach still assumes that particles are small spheres, it does not assume that all particles are small spheres. For a given particle, a highlight is stochastically determined to be either present or not. This accounts for the fact that most particles cannot be approximated as being spherical for the purposes of computing specular effects and corresponds well with our observation of real whitewater.

Particles that are selected as being highlighted are rendered each frame as very small, very bright points with the same colour as the light source. These particles are added to a separate list and each is assigned a small, random time, after which the particle is removed from the list. Letting the highlights persist over time in this way causes them to leave variable length streaks when rendered using multi-sampled motion blur. Another simple trick we use to improve the resemblance to real droplets is to draw broken instead of continuous streaks. This mimics the oscillatory deformation of moving droplets [GN06], which causes highlights to appear and disappear at different times. To do this we associate a flag with each highlight indicating whether or not it should be drawn. The state of this flag is toggled with low probability each frame.

Figure 5.29 shows frames from an animation of the particle Buddha dispersing in a turbulent wind field, animated using Perlin noise. We use this technique to simulate foam being ripped from the crests of ocean waves, where exposure to wind forces is greatest. The images show motion-blurred specular highlights, and the formation of primary and secondary rainbows which fade in and out based on the distribution of droplets in the air (see also Figure 5.25). The addition of these effects has a negligible impact on rendering time.

As future work, it would be interesting to try and develop a better model for light scattering in non-spherical and variable-sized particles, instead of assuming that surface tension overrides inertial forces, thus causing droplets to be spherical in shape. There are two ways in which this could be approached. Firstly, we could generate a time sequence of colour maps corresponding to the periodic oscillation of a droplet [GN06] and cycle between these maps when determining which particles give rise to specular highlights. This would automatically account for the stippled appearance of motion-
Figure 5.29: Frames from an animation of a particulate Buddha dispersing in the wind demonstrating broken primary and secondary rainbows and motion-blurred specular highlights.

blurred highlights. Secondly, we could compute the aggregate effect of light scattering through various collections of droplets [MWM07] and relate this to the spatially-variant particle density in the scene. This would help to simulate the multiple scattering that light undergoes when it passes through closely packed droplets.
Chapter 6

Conclusions and Future Work

The work presented in this thesis investigates many open problems related to simulating and rendering liquids. We have presented a framework for solving the shallow water equations in real-time which produces more realistic results than the alternative approach of solving the simpler, two-dimensional wave equation. The results presented show that our solver is a practical alternative to current techniques used to simulate liquids at interactive rates. We have also considered issues related to real-time and offline rendering of liquids, both at the scale of large bodies of liquid such as an ocean, and also at the scale of individual droplets. In this chapter, we provide an overview of the main contributions of this thesis, along with some suggested avenues for future work.

6.1 Summary of Contributions

1. We have developed an efficient solver for the shallow water equations based on artificial compressibility. This weakly compressible formulation is a practical alternative to the incompressibility condition which alleviates us from having to solve a computationally expensive pressure Poisson equation. We have discussed how stability and efficiency affect the choice of grid configuration, advection scheme and pressure gradient computation for a compressible fluid solver and, based on this, we presented a new solution method for the shallow water equations. We introduced an efficient technique for advecting velocity and density together on
a collocated grid and we discussed how pressure instabilities can be avoided by smoothing the density field during advection. Finally, an alternative solution method based on a simplified pressure update was proposed. This enables existing, incompressible MAC-based solvers to be transformed effortlessly into solvers for the shallow water equations. Compared to competing methods based on solutions to the 2D wave equation, our method provides greater realism since it is based on fewer simplifying assumptions. It is still limited in that it is only accurate for inviscid flows of thin layers of fluids—fully 3D phenomena cannot be simulated. Nevertheless, our approach provides further steps towards simulating the full three-dimensional Navier-Stokes equations in real-time.

2. The proposed fluid solver was extended in a number of ways. Most importantly, we modified the algorithm to work on tiled uniform grids of varying resolution and we justified this preference over the use of quadtrees. We derived accurate approximations for the pressure gradients across grid boundaries and discussed how view dependent refinement criteria can be used to make an efficient open ocean solver. We believe this is the first use of adaptive solution techniques for height-field liquid simulation in graphics. In addition, we introduced a new method for handling interaction with infinitely thin walls, a long standing problem in fluid animation. Finally, we presented some cheap modifications to augment the visual quality of heightfield-based liquids.

3. We discussed how raytracing can be used to realistically render liquids while accounting for light that is scattered into the viewing direction. We adapted these ideas to work in real-time for heightfields using fast analytical approximations and achieved very realistic ocean subsurface lighting effects which have not been previously modelled. In order to produce high quality animations of the results, we rendered the frames with OpenGL and used a software accumulation buffer to compute anti-aliasing, depth of field and motion blur.

4. Although our single-scattering approximations achieve fast and plausible results, an accurate lighting model for liquids needs to account for full multiple scattering. To achieve this, we developed a Monte Carlo path tracer which used a novel technique for accelerating photon propagation through homogeneous scat-
tering media without compromising accuracy. Our approach enables photons to traverse space in large steps without requiring phase function sampling to determine new directions for scattered photons, or costly ray intersection tests to determine when photons exit the liquid surface. Whereas most other volumetric rendering approaches make simplifying assumptions in order to gain efficiency, it is important to note that our method is a completely general acceleration scheme that computes equivalent results to a full path tracer in all cases.

5. Lastly, we presented a system for lighting and rendering particles based on OpenGL. Particles are commonly used to represent splashing whitewater in river and ocean animations and therefore a liquid rendering solution would not be complete without considering this important case. To achieve self-shadowing of particles, we computed lighting on a coarse grid which samples space more efficiently than a non-uniform distribution of points, and then interpolated the results back onto particles. We are not aware of any other work in graphics which discusses lighting particle systems to this degree of realism. For situations where it is appropriate to assume that particles represent spherical droplets (ocean mist, rain, etc.), we developed a model to capture many interesting optical phenomena that arises from specular interactions between light and liquid, including rainbow formation and glowing.

6.2 Future Work

There are a multitude of interesting avenues for future work stemming from the research described in this thesis. Here is a short list of some of the directions we would like to pursue in the near future:

- **Smoke and Fire:** As well as liquids, very realistic animation of smoke [FSJ01] and fire [NFJ02] can be achieved by solving the incompressible Navier-Stokes equations. Again, these algorithms are too slow to execute in real-time. However, it should be relatively easy to adapt these algorithms to instead use a 3D version of our compressible fluid solver in the hopes of achieving closer to interactive update rates. As for shallow water, we suspect that a high degree of compressibility can
be used for smoke and fire simulation to help ensure stability without adversely affecting verisimilitude.

- **GPU Acceleration:** We believe our compressible fluid solver can be mapped quite easily to graphics hardware by letting the GPU perform the overlap tests and accumulation between advected packets and cells using rasterization and additive blending. This could be achieved by rasterizing packets into a texture which is higher resolution than the grid dimensions, and then downsampling the result back to a texture whose dimensions are the same as the grid. Furthermore, this idea could quite easily be extended to work with unstructured grids that conform exactly to the shape of obstacles by rasterizing triangle-shaped packets instead of square packets.

- **Multi-layer Solver:** The shallow water equations solved by our simulation algorithm assume that the vertical acceleration of the fluid is negligibly small. It should be possible to model even more realistic wave dynamics by sampling the 3D velocity field as a series of stacked horizontal slices through the liquid volume, instead of a single depth-averaged slice. In this way, vertical accelerations could be resolved to some degree of accuracy without incurring much additional expense. Ultimately, by revoking the hydrostatic assumption, we would like to develop a new real-time heightfield method that computes a weakly compressible, hydrodynamic pressure field in order to capture the dispersion relation for wave propagation. At the same time, this would avoid many of the costs associated with a fully 3D MAC-based solver such as 3D interface tracking and setting free surface boundary conditions.

- **3D Coupling:** As remarked in Section 4.7, the non-iterative pressure step we proposed to update velocities without requiring a density field can also be used to simulate 3D free surface flows. We would like to try to couple this technique to our shallow water solver in order to allow 3D splashing and overlapping waves in regions that cannot be resolved by a heightfield alone. We believe that such a hybrid approach is the most promising way to achieve more realistic real-time liquid simulation since the expensive 3D simulator would only be invoked intermittently to solve for localized splashing effects.
- **Foam Rendering:** The accelerated photon propagation algorithm we developed to render homogeneous participating media can also be applied to *discrete random media* [MWM07]. It would be interesting to model light scattering through ocean foam using this technique. All that needs to change in the algorithm is the sampling process used to precompute the PDFs. So, instead of sampling a phase function to determine new directions for scattered photons, you would instead model the scattering medium geometrically (e.g., as packed foam bubbles in this case), and compute how photons scatter when they collide with this geometry.
Bibliography


139


[RKKS+07] Jonathan Ragan-Kelley, Charlie Kilpatrick, Brian W. Smith, Doug Epps, Paul Green, Christophe Hery, and Frédo Durand. The lightspeed auto-


[TSS+07] N. Thürey, F. Sadlo, S. Schirm, M. Müller-Fischer, and M. Gross. Real-time simulations of bubbles and foam within a shallow wa-


