2018-05-01

Adaptive Fluid Simulation Using a Linear Octree Structure

Sean A. Flynn
Brigham Young University

Adaptive Fluid Simulation Using a Linear Octree Structure

Sean A. Flynn

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

ABSTRACT

Adaptive Fluid Simulation Using a Linear Octree Structure

Sean A. Flynn
Department of Computer Science, BYU
Master of Science

An Eulerian approach to fluid flow provides an efficient, stable paradigm for realistic fluid simulation. However, its traditional reliance on a fixed-resolution grid is not ideal for simulations that simultaneously exhibit both large and small-scale fluid phenomena. Octree-based fluid simulation approaches have provided the needed adaptivity, but the inherent weakness of a pointer-based tree structure has limited their effectiveness. We present a linear octree structure that provides a significant runtime speedup using these octree-based simulation algorithms. As memory prices continue to decline, we leverage additional memory when compared to traditional octree structures to provide this improvement. In addition to reducing the level of indirection in the data, because our linear octree is stored contiguously in memory as a simple C array rather than a recursive set of pointers, we provide a more cache-friendly data layout than a traditional octree. In our testing, our approach yielded run-times that were 1.5 to nearly 5 times faster than the same simulations running on a traditional octree implementation.

Keywords: fluid simulation, physics-based animation, adaptive discretization, linear octree, water

ACKNOWLEDGMENTS

Thanks go to Parris Egbert, Seth Holladay, and Jeremy Oborn for their collaboration. I would also like to thank my wife and children for their support and sacrifice.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Realistic visual fluid simulation research is increasingly important as high-performance physical simulation work becomes more prevalent in a variety of fields including animation, visual effects, and video games. Over the past few decades, there has been great success in creating convincing fluid phenomena including ocean waves, smoke plumes, water splashing into containers, and others. By taking a physically-based approach, the burden on artists needing to hand-animate complex fluid behavior has been alleviated. Additionally, by providing controllable parameters, modern fluid simulation techniques provide artists some control over simulations. However, because physically-based fluid simulation is computationally intensive, efficient algorithms and data structures are crucial to support the ever-increasing demand for new types of simulation.

Depending on the art direction of a simulation, the simulator's configuration details can vary widely. For example, simulating large-scale ocean waves at the resolution needed to accurately resolve tiny droplets would be very inefficient, and attempting to simulate droplets with a very coarse resolution would not have the necessary accuracy. This is a problem when a fluid simulation must simultaneously capture large and small-scale phenomena.

With a traditional fixed, uniform Eulerian simulation grid the user must specify the resolution of the grid before simulating. Visual detail is tied to the resolution of the grid. Liquid phenomena such as droplets and thin sheets require a very high resolution. Because increasing resolution has a large impact on simulation time, the lowest possible resolution that still achieves the desired effect will ideally be specified. As previously mentioned, if the simulation requires both large and small details, a grid with fixed resolution will either be computationally inefficient or lack the necessary accuracy for the desired visual details. Despite this inefficiency, the uniform simulation grid has remained the preferred data structure for fluid simulation because it is straightforward to implement and is cache-friendly.

To illustrate the need for locally adaptive fluid simulation, a wave exhibiting both large and small-scale phenomena is shown in Figure 1.1. While the upper region of the image has fine details like droplets and spray, the lower region is smooth without much going on. Simulating this type of phenomena on a low resolution grid would be fast and efficient for the lower region, but the complex interactions occurring in the highly detailed regions would be lost. A low resolution grid is insufficient for these details because it does contain the necessary density of signed distance and velocity samples. Signed distance and velocity are stored at fixed locations at each cell. Signed distance indicates the distance from the grid location to the fluid surface and is necessary for rendering and to compute effects like surface tension. The velocity field governs the motion of the fluid. To resolve the curling motion at the top of an ocean wave requires many more velocity samples than the smooth surface below the crest of the wave.

Previous approaches to solving the adaptivity problem include the use of octrees, far-field grids, multiple grids of varying resolutions, adaptive particle-based approaches, and a variety of others. While each technique has alleviated some aspect of the need for adaptive simulation, each also has its respective drawbacks. These issues include inefficient use of memory, high computational overhead, difficulty of implementation, or data structures that are overly complex. Our approach seeks to minimize these drawbacks while achieving their strengths.

We present a linear octree-based method that allows for regions of both high and low resolution in a single grid (Figure 3.1). We differ from existing octree-based approaches in that we store our octree contiguously in memory as a simple C array. By avoiding the use of a recursive set of pointers that is fundamental to tree data structures, we reduce the level of

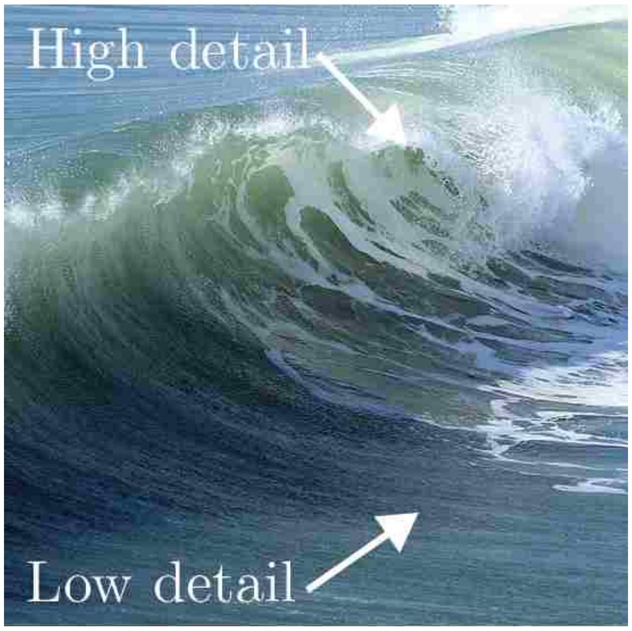

Figure 1.1: An ocean wave with regions of both high an low detail is shown above. A locally adaptive fluid simulation grid is necessary to capture the fine details at the top of image while remaining efficient in the low-detail region at the bottom.

indirection in the data, and significantly speed up the computation time. With the familiar interface of a traditional octree, our linear octree grid structure can leverage the algorithms and literature that already exist for octrees.

Chapter 2

Related Work

Modern fluid dynamics from a computer graphics perspective began in 1996 when Foster and Metaxas presented a comprehensive Eulerian fluid simulator [10]. Their work was mainly based on an earlier computational physics paper [14]. Jos Stam improved upon this work while attempting to create a production fluid simulator [24]. Rather than relying on explicit Eulerian schemes which required small timesteps and suffered from instability, Stam developed a method that made the simulation unconditionally stable for the first time. This was done by adding a Lagrangian technique now commonly known as the backward particle trace. These seminal papers revolutionized visual fluid simulation by efficiently approximating the Naviér Stokes equations and allowed for a wide range of fluid effects without the need for specific logic. Stam's work remains the foundation for Eulerian fluid simulators in computer graphics today.

Previous approaches to free-surface tracking were mostly done with either marker particles or with height-field techniques. While these approaches provided a general approximation of liquid surfaces, they suffered from mass dissipation. Foster and Fedkiw presented a method that used level sets to accurately track the fluid surface while avoiding mass dissipation [9]. This level set method approach was improved by Enright, Marschner, and Fedkiw by using particles on both sides of the free surface to perform a thickening step [6]. This improvement shifted the liquid rendering paradigm from a volumetric approach to a surface based approach which allowed for more convincing ray tracing techniques to be used.

Lagrangian particle-based approaches are also commonly used today. Foundational among these is Smoothed-particle Hydrodynamics (SPH), which was initially introduced by Desbrun and Gascuel [4] and then adapted for efficient computer graphics simulations by Muller, Charypar, and Gross [18]. SPH tracks the fluid as a set of discrete particles without the use of a grid as in the Eulerian approach. The properties of the fluid are computed using weighted contributions from nearby particles. Lagrangian approaches benefit from reduced numerical dissipation, but suffer from stability issues.

The simulation technique we use in this paper is a semi-Lagrangian technique called the Fluid-implicit Particle (FLIP) method, which was originally developed by Brackbill [3] and adapted for computer graphics by Zhu and Bridson [29]. FLIP uses particles to store and advect the fluid velocity, but uses an Eulerian grid for the other fluid simulation steps like the pressure projection. This method maintains stability and reduces numerical averaging. FLIP fluid simulations allow a more free-flowing fluid behavior which is crucial for low-viscosity fluids like water.

Any fluid simulation technique that uses a grid has the potential to benefit from improvements to the design of the grid. This is obvious in pure Eulerian approaches, but also applies to a variety of hybrid techniques including FLIP. Additionally, many modern fluid simulators use multiple grids for ancillary computations like fluid surface re-meshing [26], or surface tension computation [25].

Losasso, Gibou, and Fedkiw presented an alternative to uniform grids by performing fluid simulation on an adaptive octree grid [17]. With the use of octrees, the simulation space is discretized non-uniformly which allows for regions of both high and low resolution in a single grid. Losasso's approach built upon a previous octree method [21] by extending it to use unrestricted octrees and adding support for free-surface computation. Despite the grid being non-uniform, they were able to reformulate the pressure solve and maintain a symmetric positive definite discretization. However, this approach is known to suffer from instability due to oscillatory spurious velocities at coarse-to-fine interfaces within the octree. This instability was improved by Olshanskii, Terekhov, and Vassilevski with the use of a low-pass filter [20]. Our approach to the pressure projection is based on an earlier octree-based approach which used a hierarchical pressure solve [23]. The FLIP fluid method was also adapted to these octree-based approaches [15]. Octrees are a well-studied data structure that provide improved efficiency for large-scale simulations with varying levels of detail. However, due to the fragmented memory layout of a tree structure and the multiple levels of indirection in the data, these structures can suffer significant memory fetch slow downs when compared to uniform grids.

Recently there have been various approaches to adaptive fluid simulation that maintain a more cache-friendly data layout. Far-field grids were presented as a way of preserving uniform grid performance while allowing adaptivity [28]. A far-field grid is similar to a uniform grid except it allows a single rectangular region of the grid to have higher resolution. Our approach seeks these same uniform-like performance benefits, but with an arbitrary number of locally refined regions.

SP Grid [1, 22] allows adaptivity similar to an octree, but uses a custom data structure consisting of a pyramid of paged sparse uniform grids. Similar to this approach, we store our data linearly in memory, but rather than using multiple grids, we store ours in a single non-uniform grid to avoid the need to keep track of the data in multiple locations.

Chimera grids [5] use multiple overlapping Cartesian grids to provide adaptivity and structure with an emphasis on allowing parallelization. However this approach requires specifying the regions of high resolution upfront without allowing dynamic local adaptivity during simulation time.

Adaptive hexahedral grids and tetrahedral meshes have also been used for adaptive fluid simulation [2, 7]. These approaches have provided improved efficiency for adaptive fluid simulation but rely on non-traditional data structures that have not caught on in industry. Because we are maintaining an octree interface, we hope to avoid this problem.

In another recent approach, FLIP fluid simulations were performed on a uniform grid data structure, but with an increased number of particles near the surface [8]. This approach only treats particles near the surface as FLIP particles. Particles further from the surface are used simply to indicate the presence of the fluid rather than to store and advect velocity as in FLIP. Similar to our approach, this method focuses computation time near the surface, but rather than adapting the size of the cells in the grid, they adapt the way particles are treated.

Finally, Nielsen and Bridson presented a linear adaptive tile tree to provide efficient adaptive FLIP simulations [19]. This approach is the most similar to ours in that it stores an octree-like structure linearly in memory in a single grid. However, it does not adhere strictly to the octree interface in that each cell is not subdivided into 8 children. This removes the ability to use the existing algorithms and literature for octrees that our approach benefits from.

2.1 Contribution

Our technique is an extension of the octree-based approaches. Rather than using a recursive set of pointers, our linear octree structure is stored contiguously in memory. Our grid structure provides all of the adaptivity benefits of the octree approaches, but avoids the cache and data access inefficiency inherent in these methods. Unlike the mentioned recent custom adaptive data structures, we maintain the familiar octree interface so that the wide array of octree algorithms that already exist can easily be ported to our structure. Our technique provides the following contributions:

- An alternative way to represent a fluid simulation octree in memory that provides a significant computational speedup
- A novel combination of previous octree techniques [15, 17, 23]

With our linear octree we observed simulation run-times that were at least 1.5 times as fast and in some cases nearly 5 times as fast as a traditional octree implementation. Our approach achieves this speedup at the cost of additional memory when compared to the traditional octree. However, our approach uses no more memory than a uniform grid, and modern machines are no longer heavily constrained by memory. Because the runtime improvement is so significant, we believe that the memory cost is well worth the speedup.

Chapter 3

Linear Octree Structure

3.1 Overview

Fluid simulations that require both large and small-scale details are inefficient on traditional fixed, uniform grid structures. This problem necessitates the use of a data structure that will allow detail where it is needed while being efficient in regions where very little is occurring. Existing octree-based approaches provide the needed adaptivity, but suffer from inefficiency due to the limitations of a recursive pointer-based tree structure.

3.2 Data Layout

Our approach uses a linear octree structure without the use of a recursive set of pointers. We store our "tree" in a contiguous, linear chunk of memory and set parameters on each cell to determine the effective size. Our structure provides the same interface as a traditional octree but is more efficient. Fundamental to this improvement, we reduce the number of levels of indirection to the simulation data and avoid needing to repeatedly traverse a tree to iterate over the cells and find neighbors. A diagram showing the differences between a traditional and linear octree is shown in Figure 3.2.

To allow grids of arbitrary rectangular dimensions rather than the cube shape of a single octree, we use an array of linear octrees, technically making it a "forest" of octrees. The memory for our grid is allocated as a simple C array of node objects. Note that allocating this memory contiguously requires that we specify the minimum and maximum cell sizes upfront. As a result, individual cells sizes can change at runtime, but the global resolution of

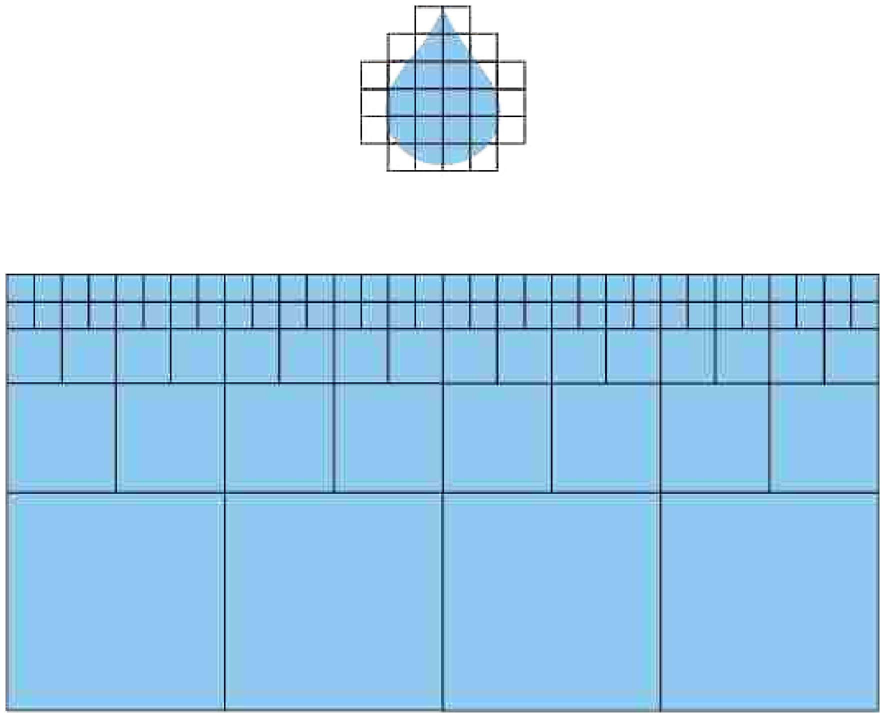

Figure 3.1: Shown in 2D for clarity, a droplet is shown falling into a body of resting water. Our linear octree structure allows multiple levels of refinement. In this example, the surface of the water and the droplet will need more resolution than the regions of fluid beneath the surface.

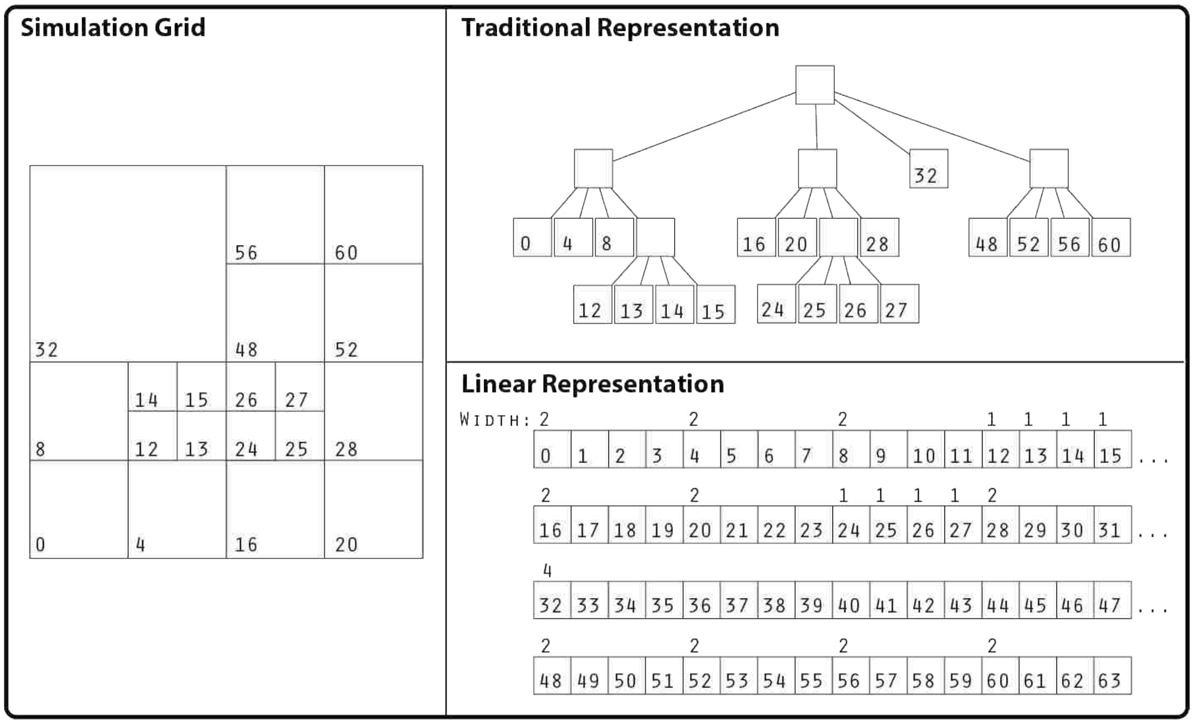

Figure 3.2: Shown in 2D for clarity, a simple adaptive fluid simulation grid is shown on the left. The top right portion of the figure shows the manner in which the data is stored in a traditional octree data structure. Note that the links between nodes are pointers that must be traversed for each data access. The bottom right portion of the image shows our storage of the data. The octree is stored at the highest resolution with indices indicating the current width of each cell. This approach greatly reduces the time required to access data over the traditional octree structure.

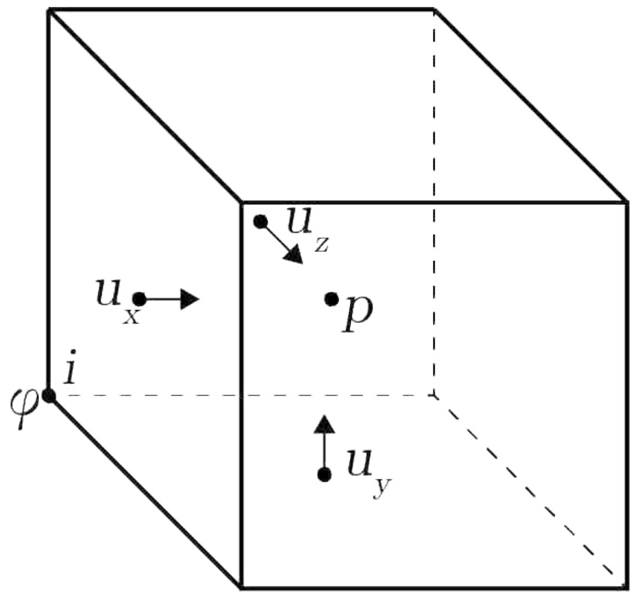

Figure 3.3: A node in our linear octree grid is shown above. The components of velocity u_x, u_y, and u_z, are stored on the minimal cell face centers as with the standard Marker and Cell (MAC) grid. Signed distance ϕ is stored at the corner of each cell to simplify interpolations. Pressure p is stored at the center of the cell.

the grid cannot. This also maximizes memory utilization because the grid is effectively being stored as if it were a uniform grid of the maximum resolution. While our grid does use more memory than a traditional octree, the speed improvement using our method is significant.

Each cell in our linear octree grid stores parameters that allow us to effectively treat it as if it were of different sizes at runtime depending on the simulation requirements. The width parameter specifies how large a cell is. For example, if a cell has a width of 8, the next $8^3 - 1$ cells will be ignored while iterating over the grid during simulation, and the cell will effectively represent a cell with volume 8^3. Only powers of two are allowed as width values to maintain the strict traditional octree interface. The other parameters on our nodes are stored similarly to the method used by Losasso et al. [17]. We store each component of velocity u_x, u_y, and u_z at the minimal cell face centers as with the standard MAC grid approach [13]. Pressure p is stored in the center of the cell. Signed distance ϕ is stored at the minimal corner of the cell rather than at the center to simplify interpolation. A diagram of a node in our grid is shown in Figure 3.3.

3.3 Local Grid Refinement

To refine the grid locally in regions where detail is needed, we simply change the width value of the cells being refined rather than deleting or allocating memory as is required with a traditional octree. For example, when a cell of width 2 is subdivided, the cell and the following 7 "child" nodes' width parameters are set to half of this cell's width. Velocity and other parameters of the "child" nodes are set to the values of the "parent" cell. When a set of eight cells is combined into a single "parent" cell, the first cell's width is doubled and the subsequent 7 cells' width values are set to 0. The other parameters of the "parent" cell are set to the average of the children.

Because the surface is the most important visual feature of simulating a liquid like water, our criteria for grid refinement is based on the signed distance field. If the zero level set is contained in a cell, it will be fully subdivided, and the further away from the surface the fluid is, the less refined the grid will become. Additionally, restricting the surface to lie within a buffer of cells of the same size eliminates the issues that arise when dealing with coarse-to-fine boundaries at the fluid-air interface.

3.4 Traditional Octree Implementation

For comparison, we also implemented a traditional octree data structure. The simulation logic for both data structures is shared in our implementation. The differences are in the data layout in memory, the grid iteration scheme, the way we subdivide and combine nodes, and in how we get the neighbors of each cell. As with the linear structure, our traditional octree is really a forest of octrees to allow non-cube shaped grids. Each node, starting with the root nodes, has either 0 or 8 pointers to child nodes. Iteration is done depth first in the same effective order as the linear structure. When subdividing a node, 8 new nodes are allocated, and when combining a node, the 8 children are deleted. The parameters on the nodes affected by refinement are set in the same way as previously described for the linear structure. For neighbor lookups, rather than constant time index arithmetic with the linear octree, the traditional octree must be traversed up to a common parent and then down to each neighbor node.

Chapter 4

Simulation

Our intention in creating this simulator is to showcase the need for high resolution in certain areas while allowing low resolution in others. Low-viscosity water-like fluid phenomena, especially with highly detailed splashes and thin sheets at the surface are a good test case. Therefore our simulator must support these features.

Our fluid simulation algorithm is based on the semi-Lagrangian FLIP method [29]. With FLIP, the use of particles greatly alleviates numerical averaging, allowing fast flowing fluid like water to be simulated. Our fluid surface representation is done using the level set method [9] with the modifications mentioned in the referenced FLIP paper. We use the fast sweeping method [27] to initialize our signed distance field at each timestep using the particle locations. For surface tension effects we use the ghost fluid method [11, 16] with mean curvature computed using the Laplacian of our signed distance field. Level set methods for the fluid-air interface are known to have trouble resolving fine details at the surface without a very high resolution grid. This difficult case provides a good test scenario for our linear octree grid structure.

4.1 Hierarchical Pressure Projection

The pressure projection is the process by which we enforce non-divergence in our fluid. That is, our fluid must be incompressible, and we can enforce that by setting up a linear system of equations for the pressure at each cell. This system of equations is then solved iteratively as a sparse, diagonal matrix. For large simulations, this is the speed bottleneck. Each fluid cell in the simulation gets a row in the pressure solve matrix. The primary benefit of using an adaptive approach like ours is that the total number of fluid cells in the simulation is reduced, and thus the most computationally-intensive step of the simulation takes less time.

Our pressure projection is based on a previous octree-based fluid simulation method [23] except we adapt it to liquids rather than smoke. Rather than attempt to solve for the pressure non-uniformly in one step as in some recent techniques [17], we found it more straightforward to perform the pressure projection as a series of uniform solves starting with the largest size in the grid then progressing down to the smallest. The psuedo-code at each timestep t is shown in Algorithm 1.

Algorithm 1 Hierarchical Pressure Projection

```
1: Find the list of $sizes$ in the grid
2: Determine the largest size $s_{max}$ in $sizes$
3: Compute pressure at $s_{max}$ for the entire grid (section 4.2)
4: for each size $s_i$ in $sizes$ < $s_{max}$ do
5:     Compute the pressure at $s_i$ (section 4.2)
6:     Store the pressure at each cell in the grid at size $s_i$
7: end for
8: Subtract the pressure gradient (section 4.3)
```

4.2 Pressure Boundary Conditions

At the largest size s_{max}, pressure is computed with boundary conditions at the solid and air interfaces identically to the traditional uniform ghost fluid approach with surface tension. That is, the pressure p at the fluid-air interface is equal to the mean curvature κ multiplied by a surface tension constant γ (Equation 4.1).

$$p = \gamma\kappa \tag{4.1}$$

The mean curvature κ is computed using a 3D Laplacian kernel on interpolated values of the signed distance field at the fluid-air interface, where $\phi = 0$ (Equation 4.2). The location where $\phi = 0$ is at a distance θ from the center of the cell being considered (Equation 4.3).

Note that because we store signed distance values at the corners of cells, the signed distance at the cell centers in the computation for θ must also be interpolated.

$$\kappa = \nabla \cdot \nabla \phi \tag{4.2}$$

$$\theta = \frac{\phi_{i,j,k}}{\phi_{i,j,k} - \phi_{i+s,j,k}} \tag{4.3}$$

For the subsequent smaller sizes s_i, solving for pressure requires special consideration at the t-junctions between cells of size s_i and s_{i-1}. At these boundaries, the pressure is explicitly set to the interpolated value of pressure from the previous solve at size s_{i-1}. An example of this is shown in Figure 4.1

4.3 Pressure Gradient

The fundamental behavior that drives fluid motion is that fluid tends to move from regions of high pressure to low pressure. To model this behavior algorithmically, we must compute the gradient of the pressure field and subtract it from the velocity field. While we computed the pressure hierarchically with several independent matrix solves, we do not use a hierarchical approach when computing pressure gradients. Because velocity is stored at one size for each cell individually, we must compute the gradient of the pressure field non-uniformly in one pass. We will now show how the non-uniform pressure gradient is computed.

To compute the pressure gradient there are 7 cases that must be considered. The cases which require special consideration are shown in Figure 4.2 and each case is described in Equations 4.4-4.11. These cases are listed below:

1. Neighboring cells of the same size.
2. A smaller fluid cell bordering a larger fluid cell.
3. A larger fluid cell bordering smaller fluid cells.

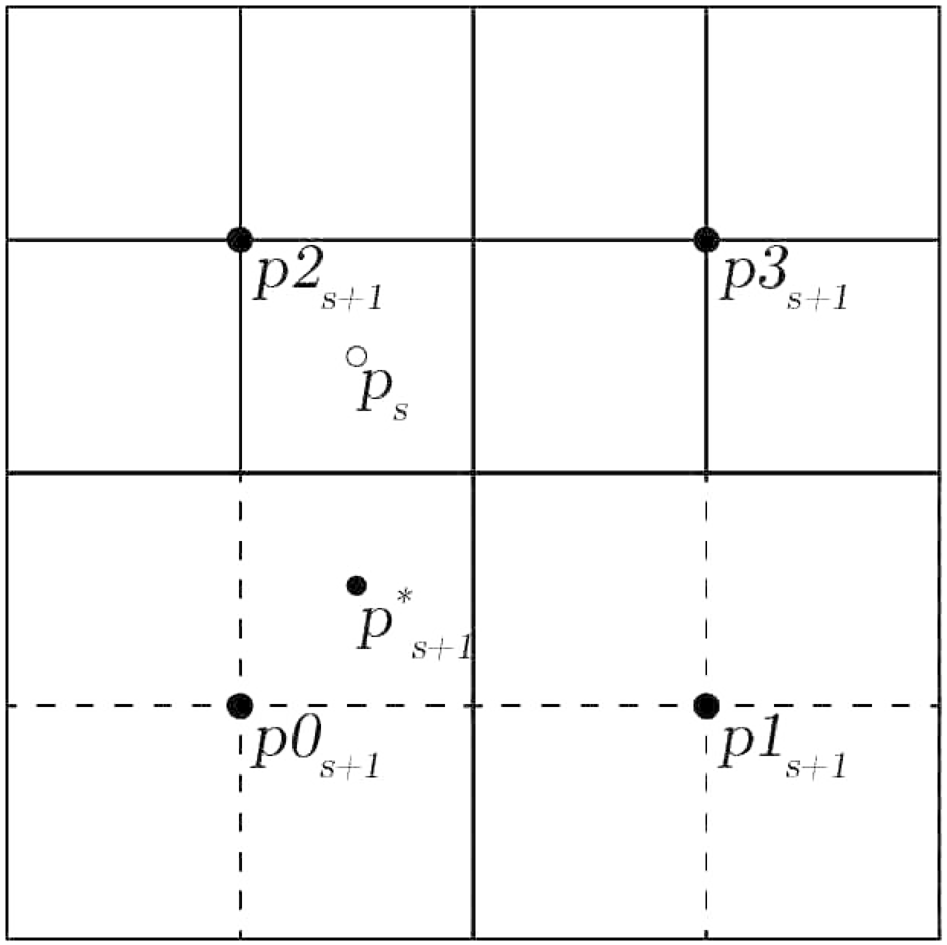

Figure 4.1: Shown in 2D for clarity, an example of solving for pressure is shown in this diagram. In this example, we are currently solving for the pressure at size s. The pressure for size $s+1$ has been computed in the previous iteration of the hierarchy and is shown stored at the locations $p0_{s+1}$, $p1_{s+1}$, $p2_{s+1}$, and $p3_{s+1}$. When setting the row in our pressure projection matrix to solve for p_s, we interpolate the value of $p*_{s+1}$ using $p0_{s+1}$, $p1_{s+1}$, $p2_{s+1}$, and $p3_{s+1}$, and explicitly set it as the pressure below p_s.

4. A smaller fluid cell bordering a larger air cell.

5. A larger fluid cell bordering smaller air cells.

6. A smaller air cell bordering a larger fluid cell.

7. A larger air cell bordering smaller fluid cells.

For boundaries between cells of the same size, the gradient is computed normally with the stored values for pressure and subtracted from each component of velocity (Equation 4.4).

$$u(x, y, z) = u(x, y, z) - \frac{\Delta t}{s_i} \nabla p(x, y, z) \tag{4.4}$$

At t-junction boundaries, special consideration is necessary. When a smaller fluid cell borders a larger fluid cell, the neighboring pressure is interpolated from the values stored at the next highest size in the grid. This interpolated pressure is then subtracted from the pressure at the smaller cell to compute the gradient (Case 1, Equation 4.5).

$$\nabla p_1 = p_s - p^*_{s+1} \tag{4.5}$$

When a larger fluid cell neighbors smaller fluid cells, the pressure gradient is computed by subtracting the average pressure of the neighboring cells from the cell's pressure (Case 2, Equations 4.6 and 4.7).

$$\nabla p_2 = p_s - {p^a}_{s-1} \tag{4.6}$$

$${p^a}_{s-1} = \frac{p0_{s-1} + p1_{s-1} + p2_{s-1} + p3_{s-1}}{4} \tag{4.7}$$

When a smaller fluid cell borders a larger air cell, the pressure gradient is computed by subtracting the ghost pressure at the fluid-air interface. This is computed using the signed

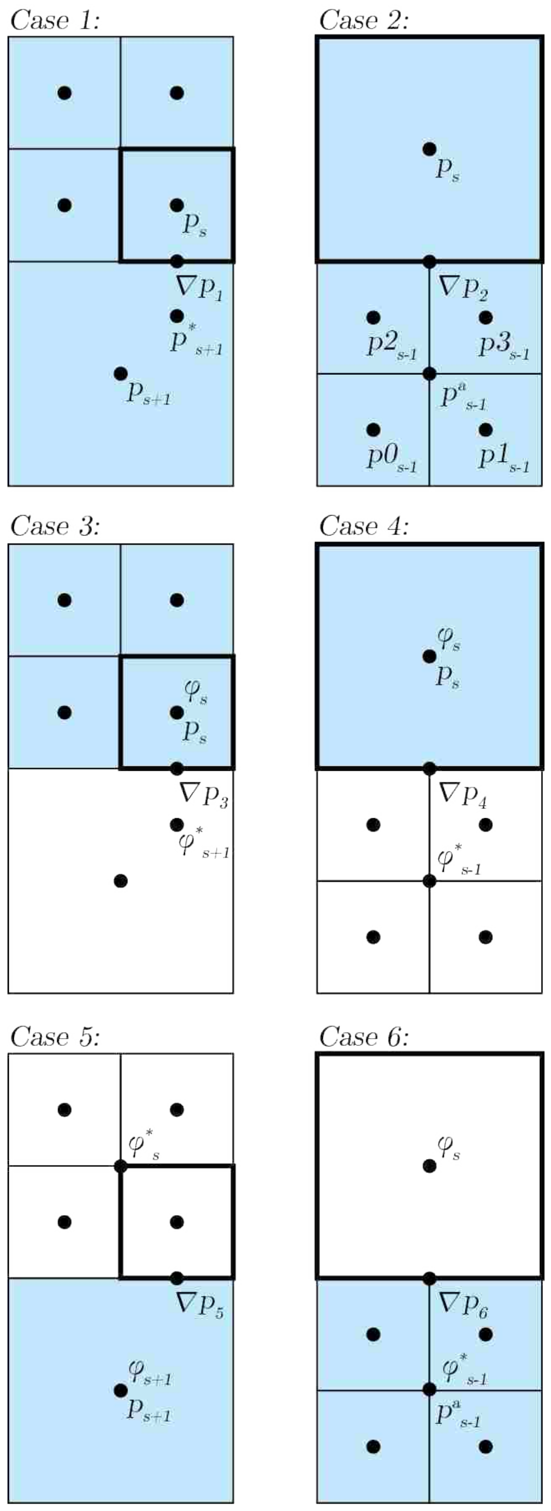

Figure 4.2: Shown in 2D for clarity, the six t-junction cases for computing the non-uniform pressure gradient are shown. The filled cells shown in blue are fluid cells and the white cells are air cells. The equations for computing ∇p_i are shown in Equations 4.4-4.11.

distance value of the fluid cell and the interpolated signed distance at the location below the fluid cell center (Case 3, Equation 4.8).

$$\nabla p_3 = p_s - \frac{\gamma\kappa}{\theta} + \frac{\phi^*_{s+1}}{\phi_s} p_s \tag{4.8}$$

When a larger fluid cell neighbors smaller air cells, the signed distance field is used to compute the ghost pressure at the center of the smaller cells. This ghost pressure is then used to compute the pressure gradient (Case 4, Equation 4.9).

$$\nabla p_4 = p_s - \frac{\gamma\kappa}{\theta} + \frac{\phi^*_{s-1}}{\phi_s} p_s \tag{4.9}$$

In the case where a smaller air cell borders a larger fluid cell, we use the signed distance at the center of the cells that the air cell is a part of to compute the ghost pressure at the fluid-air interface (Case 5, Equation 4.10).

$$\nabla p_5 = \frac{\gamma\kappa}{\theta} + \frac{\phi^*_s}{\phi_{s+1}} p_{s+1} - p_{s+1} \tag{4.10}$$

Finally, when a larger air cell borders smaller fluid cells, the signed distance of the air cell and the signed distance at the center of the bordering fluid cells are used to determine the ghost pressure. The gradient is then computed using the ghost pressure and the average of the pressures of the fluid cells, ${p^a}_{s-1}$ (Case 6, Equations 4.11 and 4.7).

$$\nabla p_6 = \frac{\gamma\kappa}{\theta} + \frac{\phi_s}{\phi^*_{s-1}} {p^a}_{s-1} - {p^a}_{s-1} \tag{4.11}$$

We have now shown each case for how the pressure gradient is computed on a non-uniform octree grid. We then subtract the pressure gradient from each component of velocity at each cell in the grid as was previously shown in Equation 4.4. This ensures that our fluid is now divergence free.

When simulating fluids on our linear octree structure, we observe a significant runtime improvement when compared to a traditional octree. We provide the benefit of local grid refinement, but we don't rely on a recursive set of pointers to do so. Instead, our linear octree structure is allocated contiguously in memory and we use fast index arithmetic and set parameters on each cell to provide adaptivity. The result is a more time efficient method for adaptive fluid simulation while maintaining a familiar data structure interface.

Chapter 5

Results

For our results, we implemented both the traditional and the linear octree structures and shared the simulation code between the two while customizing the data structures as previously described. There was little time spent optimizing our code. With additional work, especially on the simulation code, both the traditional and the linear implementations would likely see a decent speedup. We used Eigen [12] for our vectors and matrices, including their conjugate gradient sparse matrix solver. SideFX Houdini was used to generate the final fluid surface and the grid representations. We used Houdini's Mantra renderer to produce our final images.

To verify the difference in speed between the traditional octree and our linear octree we used three simulation setups: resting water, crown splashes, and a container filling up with water. Table 5.1 shows the average timestep runtime and the memory utilization. As expected there is a significant speedup when using a linear octree versus using a traditional octree. The resulting simulation meshes and particles were identical when comparing the linear to the traditional octree structure.

Our first test case, shown in Figure 5.1, is a simulation of resting water. This allowed us to verify that the fluid remained stable. For various configurations of auto-subdivision, with the restriction that the fluid surface always lie within a buffer of uniform cells, we observed that the fluid remained at rest.

Next, we tested several different simulation configurations for crown splashes, as shown in Figures 5.2, 5.3, and 5.4. Crown splashes are an ideal candidate for benefiting

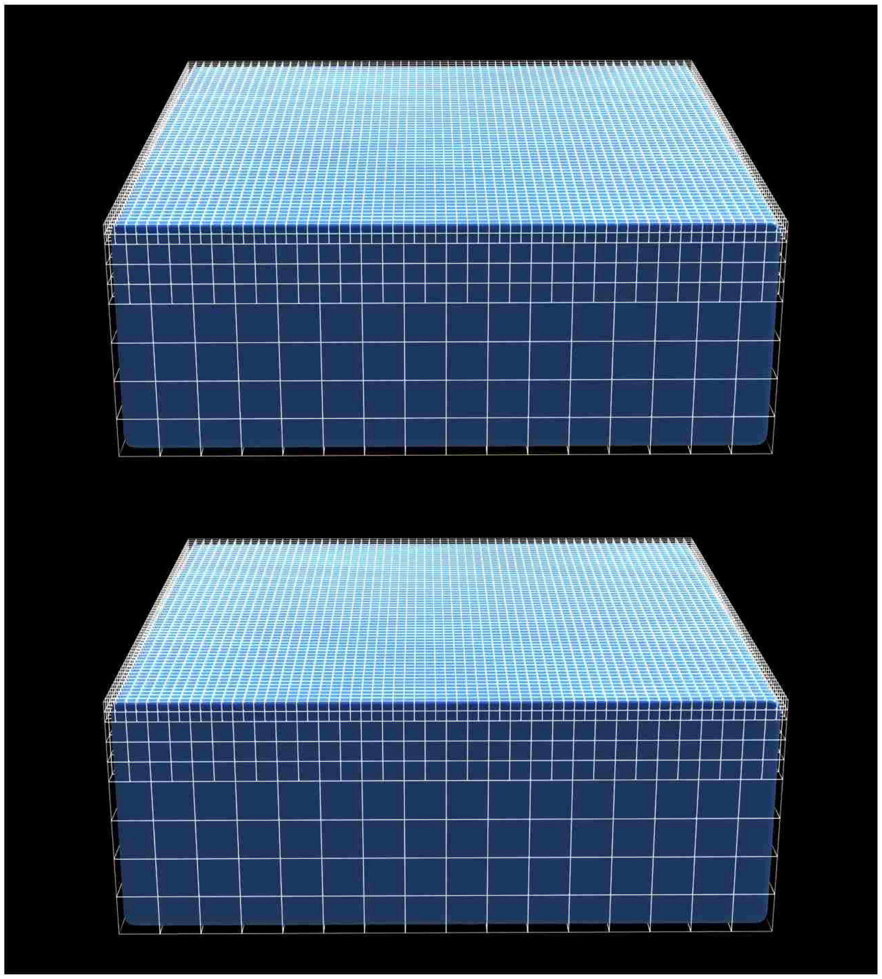

Figure 5.1: A locally refined resting water simulation is shown above. Using both the linear and traditional octree structures, the water remains at rest. The top shows the first frame of simulation and the bottom shows the last frame.

Simulation Setup	Traditional Avg. Seconds Per Timestep	Traditional Memory Utilization	Linear Avg. Seconds Per Timestep	Linear Memory Utilization
Resting water (64x64x64) auto-refined	6.57s	15.52MB	3.66s	97.52MB
Crown splash (64x64x64) no refinement	15.27s	97.52MB	9.24s	97.52MB
Crown splash (64x64x64) auto-refined	8.69s	27.24MB	4.39s	97.52MB
Crown splash (128x128x128) no refinement	249.14s	780.14MB	180.44s	780.14MB
Crown splash (128x128x128) auto-refined	144.87s	230.72MB	89.36s	780.14MB
Container filling up (48x80x48) auto-refined	11.69s	58.30MB	2.47s	68.57MB

Table 5.1: This table shows the results for our test simulations. In the left column is the simulation setup with the effective dimensions of the grids. We performed each simulation on our linear and traditional octree implementations. The resulting fluid simulations were identical. The average elapsed time per timestep is shown in addition to the memory utilization. In all cases the linear octree was significantly faster than the traditional octree.

from adaptive fluid simulation because they require a high amount of resolution in order to form the crown shape, but don't require much resolution elsewhere. For each crown splash simulation, we simulated on both the traditional and the linear octree grids. We ran the simulations both with auto-refinement turned on and without auto-refinement, making the simulation behave as if they were uniform simulations. The first thing to note in general about our results is that the auto-refined simulations are nearly identical to the uniform simulations. This means that for these cases, auto-refinement does not cause any noticeable artifacts, and that the areas of low resolution are in fact having very little effect on the surface. For crown splashes, the speedup seems to scale linearly with the size of the grid and number of particles, with the linear octree simulations running in about 50-70% of the time that the traditional octree takes.

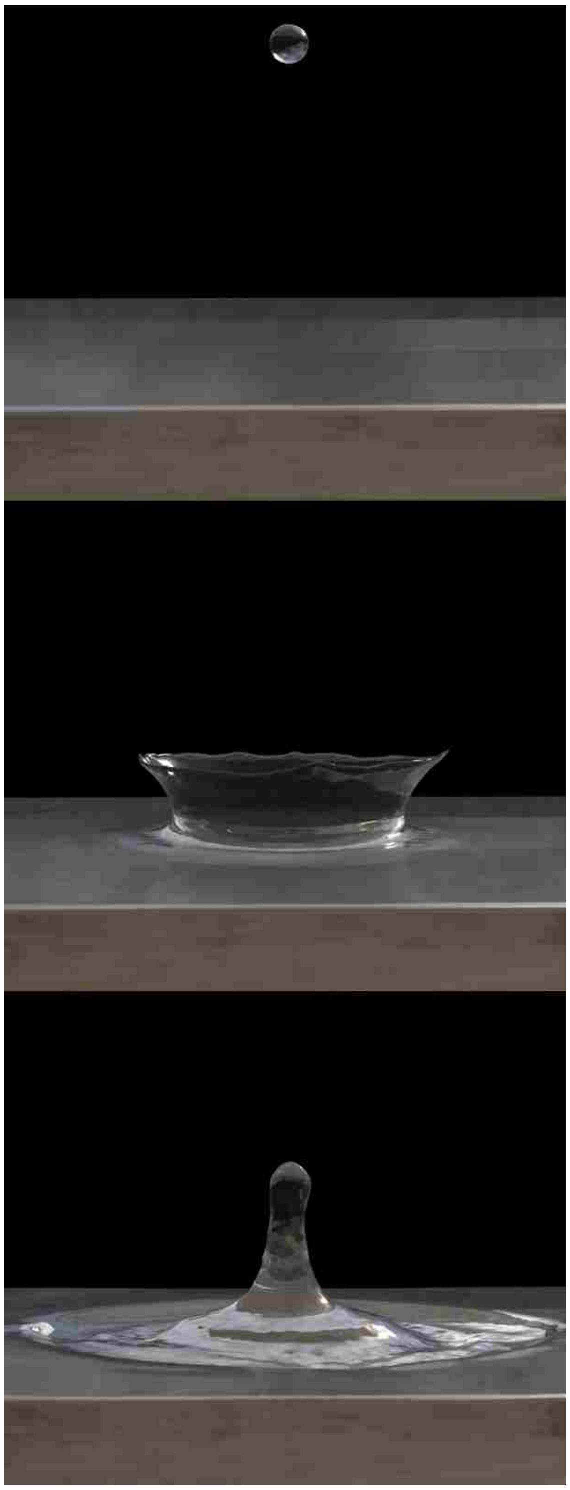

Figure 5.2: An example of a crown splash simulated using our linear octree structure. Notice that the fine details occur at the fluid surface while there is very little going on below the surface.

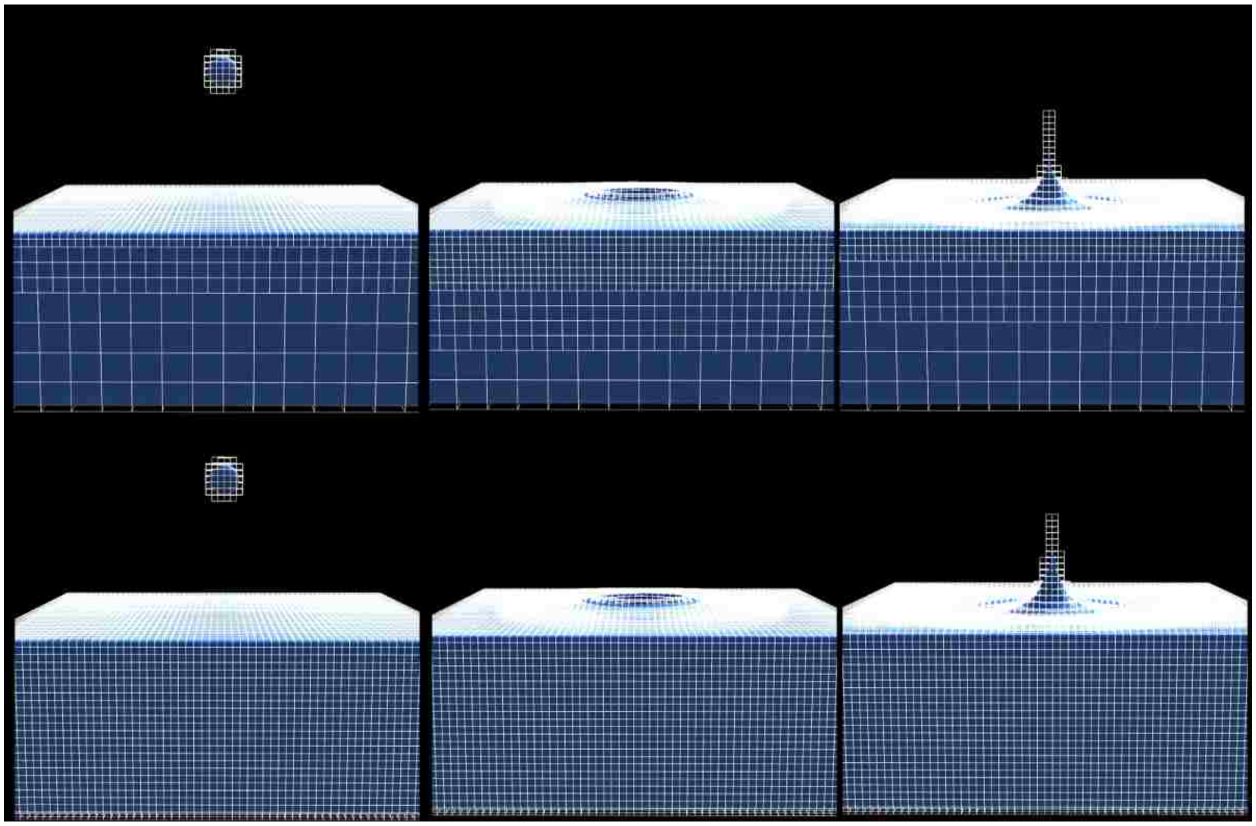

Figure 5.3: A crown splash with a low effective resolution (64x64x64) is shown above. The top row shows the splash with local auto-refinement turned on, and the bottom row shows the same simulation configuration, but with the grid always fully refined, making it behave as a uniform grid.

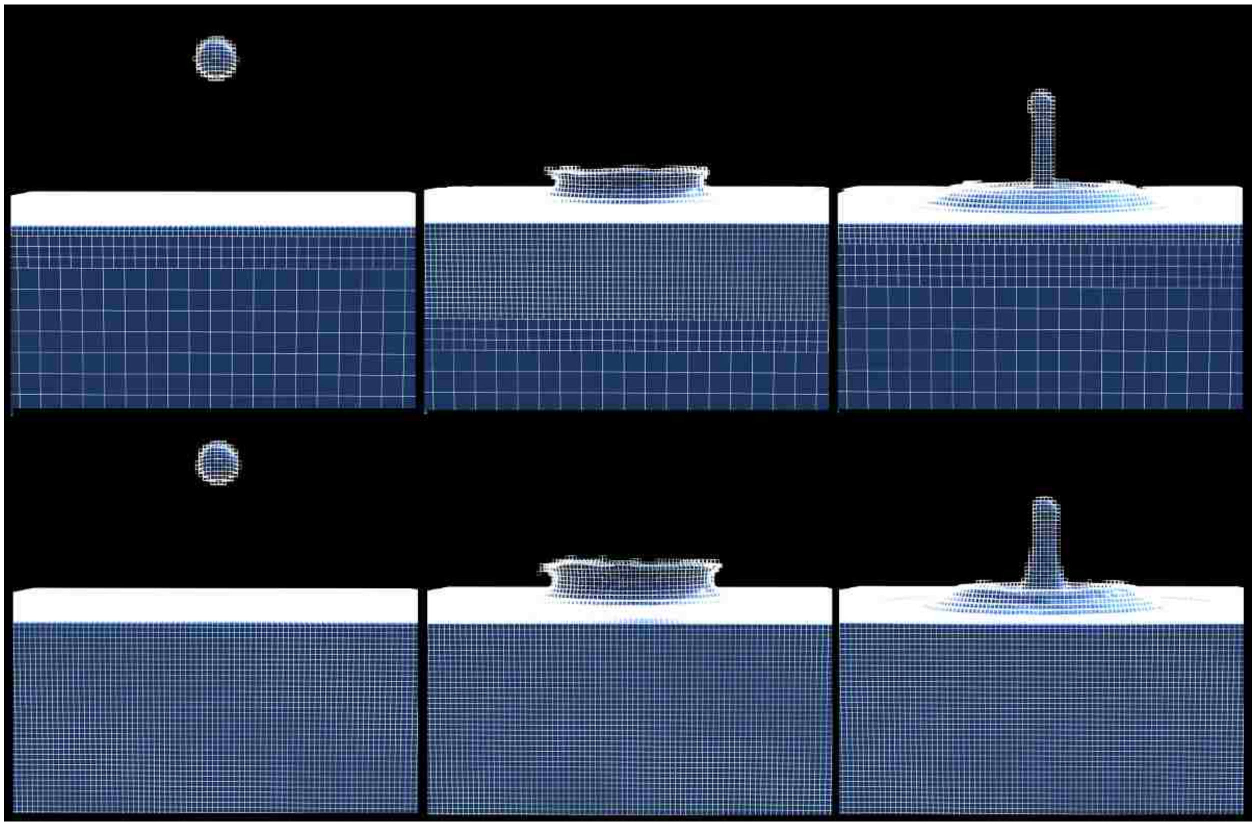

Figure 5.4: A crown splash with enough effective resolution (128x128x128) to begin to see the defining details appear. Like Figure 5.3, the top row shows the linear octree simulator running with local auto-refinement, and the bottom is shown uniformly refined. Notice that while the simulations differ slightly, both were able to capture the crown shape despite the simulation on top taking less than half of the runtime as the simulation shown on the bottom.

Finally, shown in Figure 5.5, we compared the results of simulating a container filling up with liquid between the traditional and linear octree grid structures. We used a lower FLIP to PIC ratio in this case, so the fluid is more viscous. This is another good candidate for local adaptive grid refinement. At the beginning of the simulation the whole grid needs high detail, but as the container fills, the regions at the bottom are having less of an impact, and so they do not need as much resolution. The speedup with the linear octree structure was even better with this type of simulation than with crown splashes. The linear octree structure was nearly 5 times as fast as the traditional in this case. This difference in performance is likely due to the number of fluid cells in the grid constantly increasing, which requires processing larger portions of the grid as the simulation progresses.

We did experience some issues with simulations of containers filling up. Because the fluid is flowing so quickly from the top of the fluid to the bottom, the liquid below the surface is having some degree of impact on the surface. A better criteria for where the grid should be locally refined could alleviate this. Perhaps rather than only refining at the surface, the high velocity in cells should be considered as an event when refinement should occur as well. There were also issues with the fluid-solid boundaries in coarse regions of the grid having gaps between the fluid and solid. This is partially an artifact of simulating on a very low resolution grid. It is also a result of the technique we are using for these boundary types. With further consideration, this could be improved.

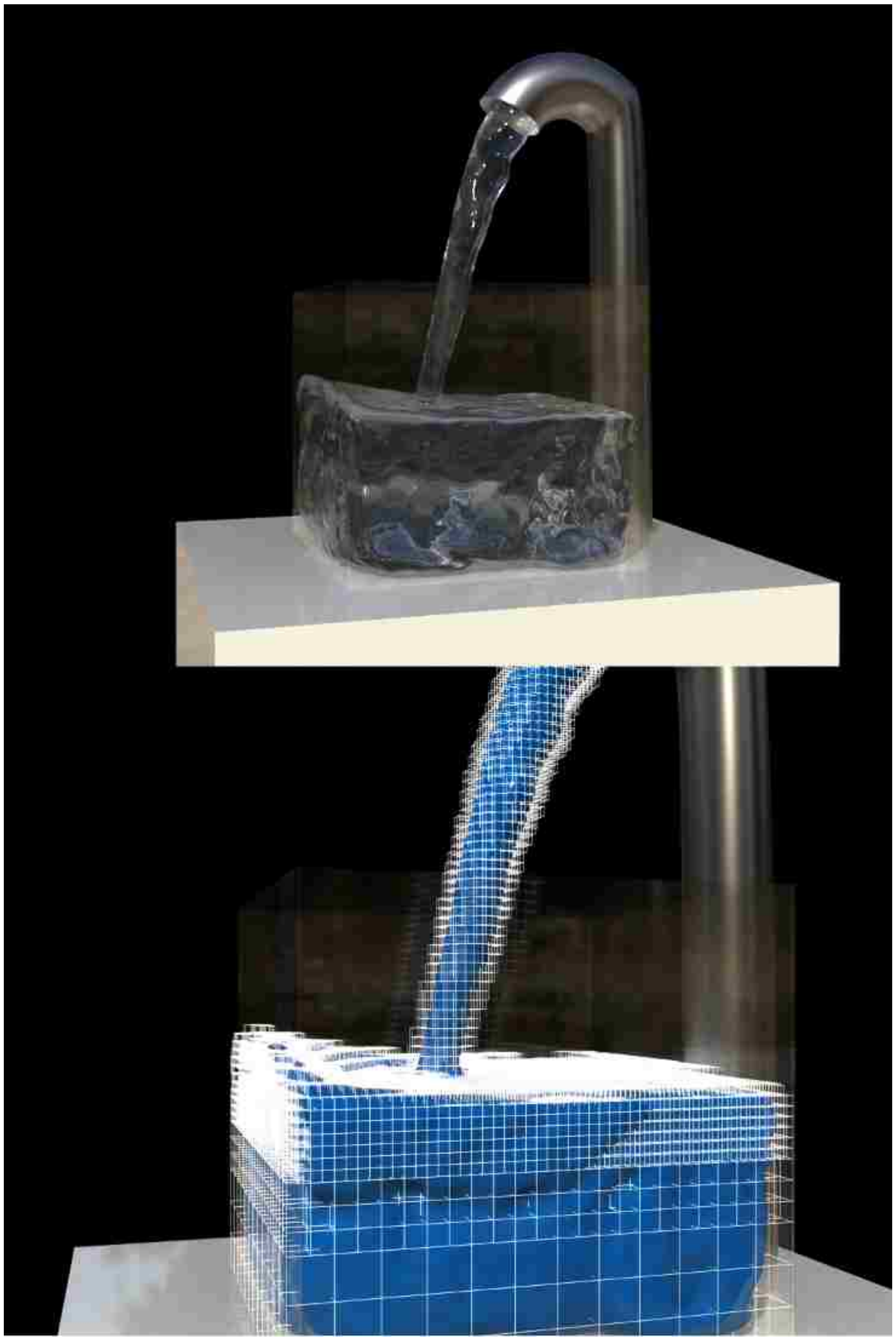

Figure 5.5: An example of a container filling with liquid using our linear octree structure. As the container fills up, the resolution of the grid at the bottom is reduced.

Chapter 6

Conclusion and Future Work

We have presented a method for adaptive fluid simulation that uses an efficient linear octree grid structure. The current preferred data structure for performing fluid simulation is a uniform grid because of its cache-friendly data layout and its simplicity. However, fluid simulations frequently require simultaneously capturing both large and small-scale details making the uniform grid inaccurate at low resolutions or inefficient at high resolutions. Existing methods for providing adaptivity include octrees, far-field grids, narrow band techniques, and a variety of custom grid data structures. These methods often suffer from high computational or implementation overhead and complexity. Our method provides the common interface of an octree with its adaptivity, but avoids the data indirection inherent in a recursive pointer-based tree structure. While our approach uses more memory, we observed a significant runtime speedup when compared to a traditional octree implementation. The run-times on our structure were around 1.5 to 5 times faster than the run-times on the traditional octree grid depending on the simulation configurations.

Intuitively, this speedup can be primarily attributed to our data structure having faster data access than the traditional octree structure. Rather than needing to use a depth-first search to find a cell in our grid and then access its data, we use simple index arithmetic to lookup a cell based on the position. At each time step in the simulation, we iterate over the grid several times and for many of the iterations we must look up the neighbors for each cell. Getting the neighbors in our approach is a constant-time index arithmetic operation, whereas on the traditional octree, the tree must be traversed up to a common parent and down to each neighbor. Additionally, while iterating over our grid at the finest resolution, because of the contiguous linear locality of the data, we observe improved cache-performance over the fragmented memory layout of the traditional octree.

We plan to explore a few areas of improvement for our linear octree method. Because we store the entire grid as if it were of the highest resolution throughout the simulation, there may be ways to better utilize it. To simplify or even speedup velocity interpolations, we could potentially cache the results of interpolations as they occur on the grid, and then when interpolating points near those that are cached, use simple trilinear interpolation at the highest resolution to get the velocity without the need for special cases at coarse-to-fine boundaries. It may also be worth exploring using our structure as it if it were a uniform grid of the highest resolution for all steps of the simulation except the pressure projection. We are also interested in exploring better criteria for local refinement beyond always refining only at the fluid-air interface as described in Section 3.3.

References

[1] Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Trans. Graph.*, 36(4):140:1–140:12, July 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073625.

[2] Ryoichi Ando, Nils Thürey, and Chris Wojtan. Highly adaptive liquid simulations on tetrahedral meshes. *ACM Trans. Graph.*, 32(4):103:1–103:10, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461982. URL `http://doi.acm.org/10.1145/2461912.2461982`.

[3] J U Brackbill and H M Ruppel. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *J. Comput. Phys.*, 65(2):314–343, August 1986. ISSN 0021-9991. doi: 10.1016/0021-9991(86)90211-1. URL `http://dx.doi.org/10.1016/0021-9991(86)90211-1`.

[4] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*, pages 61–76, New York, NY, USA, 1996. Springer-Verlag New York, Inc. ISBN 3-211-82885-0. URL `http://dl.acm.org/citation.cfm?id=274976.274981`.

[5] R. Elliot English, Linhai Qiu, Yue Yu, and Ronald Fedkiw. Chimera grids for water simulation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '13, pages 85–94, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2132-7. doi: 10.1145/2485895.2485897.

[6] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21(3):736–744, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566645. URL `http://doi.acm.org/10.1145/566654.566645`.

[7] F. Ferstl, R. Westermann, and C. Dick. Large-scale liquid simulation on adaptive hexahedral grids. *Visualization and Computer Graphics, IEEE Transactions on*, 20(10):1405–1417, Oct 2014. ISSN 1077-2626. doi: 10.1109/TVCG.2014.2307873.

[8] Florian Ferstl, Ryoichi Ando, Chris Wojtan, Rüdiger Westermann, and Nils Thürey. Narrow band FLIP for liquid simulations. *Comput. Graph. Forum*, 35(2):225–232, May 2016. ISSN 0167-7055. doi: 10.1111/cgf.12825. URL `https://doi.org/10.1111/cgf.12825`.

[9] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 23–30, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: 10.1145/383259.383261. URL `http://doi.acm.org/10.1145/383259.383261`.

[10] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, September 1996. ISSN 1077-3169. doi: 10.1006/gmip.1996.0039. URL `http://dx.doi.org/10.1006/gmip.1996.0039`.

[11] Frederic Gibou, Ronald P. Fedkiw, Li-Tien Cheng, and Myungjoo Kang. A second-order-accurate symmetric discretization of the Poisson equation on irregular domains. *J. Comput. Phys.*, 176(1):205–227, February 2002. ISSN 0021-9991. doi: 10.1006/jcph.2001.6977. URL `http://dx.doi.org/10.1006/jcph.2001.6977`.

[12] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[13] F. H. Harlow and J. E. Welch. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8:2182–2189, December 1965. doi: 10.1063/1.1761178.

[14] Francis H. Harlow and J. Eddie Welch. Numerical calculation of timedependent viscous incompressible flow of fluid with free surface. *Physics of Fluids*, 8(12):2182–2189, 1965. doi: http://dx.doi.org/10.1063/1.1761178.

[15] Woo-Suck Hong. *An Adaptive Sampling Approach to Incompressible Particle-based Fluid.* PhD thesis, College Station, TX, USA, 2009. AAI3370710.

[16] Myungjoo Kang, Ronald P. Fedkiw, and Xu-Dong Liu. A boundary condition capturing method for multiphase incompressible flow. *J. Sci. Comput.*, 15(3):323–360, September 2000. ISSN 0885-7474. doi: 10.1023/A:1011178417620. URL `https://doi.org/10.1023/A:1011178417620`.

[17] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 457–462, New York, NY, USA, 2004. ACM. doi: 10.1145/1186562.1015745.

[18] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-659-5. URL `http://dl.acm.org/citation.cfm?id=846276.846298`.

[19] Michael B. Nielsen and Robert Bridson. Spatially adaptive FLIP fluid simulations in Bifrost. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, pages 41:1–41:2, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4282-7. doi: 10.1145/2897839.2927399.

[20] Maxim A. Olshanskii, Kirill M. Terekhov, and Yuri V. Vassilevski. An octree-based solver for the incompressible Navier-Stokes equations with enhanced stability and low dissipation. *Computers & Fluids*, 84:231 – 246, 2013. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2013.04.027. URL `http://www.sciencedirect.com/science/article/pii/S0045793013001771`.

[21] Stéphane Popinet. Gerris: A tree-based adaptive solver for the incompressible Euler equations in complex geometries. *J. Comput. Phys.*, 190(2):572–600, September 2003. ISSN 0021-9991. doi: 10.1016/S0021-9991(03)00298-5.

[22] Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Trans. Graph.*, 33(6):205:1–205:12, November 2014. ISSN 0730-0301. doi: 10.1145/2661229.2661269.

[23] Lin Shi and Yizhou Yu. Visual smoke simulation with adaptive octree refinement. Technical report, Champaign, IL, USA, 2002.

[24] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-48560-5. doi: 10.1145/311535.311548. URL `http://dx.doi.org/10.1145/311535.311548`.

[25] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk. A multiscale approach to mesh-based surface tension flows. *ACM Trans. Graph.*, 29(4):48:1–48:10, July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778785. URL `http://doi.acm.org/10.1145/1778765.1778785`.

[26] Chris Wojtan, Nils Thürey, Markus Gross, and Greg Turk. Deforming meshes that split and merge. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, pages 76:1–76:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-726-4. doi: 10.1145/1576246.1531382.

[27] Hongkai Zhao. A fast sweeping method for eikonal equations. *Math. Comput.*, 74(250): 603–627, 2005.

[28] Bo Zhu, Wenlong Lu, Matthew Cong, Byungmoon Kim, and Ronald Fedkiw. A new grid structure for domain extension. *ACM Trans. Graph.*, 32(4):63:1–63:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461999.

[29] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073298. URL `http://doi.acm.org/10.1145/1073204.1073298`.