Adaptive Fluid Simulation Using a Linear Octree Structure

Sean Flynn
Brigham Young University
ecthelisean@gmail.com

Parris Egbert
Brigham Young University
egbert@cs.byu.edu

Seth Holladay
Brigham Young University
seth_holladay@byu.edu

Jeremy Oborn
Brigham Young University
jeremy.oborn@gmail.com

ABSTRACT

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

ACM Reference Format:
Sean Flynn, Parris Egbert, Seth Holladay, and Jeremy Oborn. 2018. Adaptive Fluid Simulation Using a Linear Octree Structure. In *CGI 2018: Computer Graphics International 2018, June 11–14, 2018, Bintan Island, Indonesia.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3208159.3208196

1 INTRODUCTION

Physically-based approaches to fluid simulation have drastically increased the realism in modern visual entertainment. Because the underlying algorithms are computationally intensive, efficient data structures are crucial to support the ever-increasing demand for new simulations.

CGI 2018, June 11–14, 2018, Bintan Island, Indonesia

ACM ISBN 978-1-4503-6401-0/18/06…$15.00
https://doi.org/10.1145/3208159.3208196

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.

The uniform Eulerian simulation grid has remained the most popular data structure for fluid simulation due to its cache-friendly data layout and straightforward implementation. With this type of grid, the user must predetermine and set a fixed global resolution for the grid that will capture the finest anticipated details such as droplets or thin sheets of water. This is problematic because the fixed resolution of the grid will be either unnecessarily high in regions where little detail is needed, or too low in areas that require fine detail. Therefore, a grid that can dynamically adapt its resolution only where detail is needed is desirable.

Previous approaches to solving the adaptivity problem include the use of octrees, multiple grids of varying resolutions, and a variety of other custom data structures. While each technique has alleviated some aspect of the need for adaptive simulation, each also has its respective drawbacks. These issues include an 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 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 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.

1.1 Related Work

The foundational computer graphics work in fluid simulation with realistic fluid surface computation is based on the level set method [4, 7] which builds upon Stam's seminal fluid simulation work [19]. Our simulation algorithm is specifically based on the Fluid-Implicit Particle (FLIP) method [21] which is a hybrid semi-Lagrangian technique that uses particles to store and advect fluid velocity resulting in a more free-flowing fluid behavior. All of these techniques rely on a uniform Eulerian simulation grid.

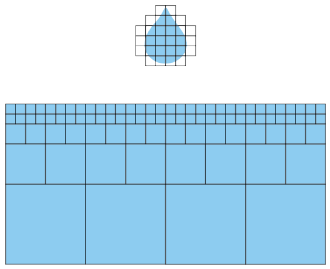

Figure 1: A 2D droplet is shown falling into a body of resting water. Our linear octree structure locally adapts grid resolution where it is needed.

Octrees were presented as an alternative to uniform grids in order to allow local adaptivity while maintaining a stable fluid simulation method [12, 15, 16, 18] with varying levels of success. The FLIP fluid method was later adapted to these octree approaches [9]. 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 a variety of custom data structures used for adaptive fluid simulation. VDB [13] is a robust hierarchical data structure that provides spatial adaptivity. Far-field grids [20] preserve a uniform grid data layout but only natively support a single rectangle-shaped region of higher resolution. A grid coarsening step performed only for the pressure solve provided a speedup for large-scale fluid simulations [11]. Chimera grids [3] use multiple overlapping Cartesian grids to provide adaptivity and structure with an emphasis on allowing parallelization. SP Grid [1, 17] uses an efficient pyramid of custom octree-like sparse paged grids to allow adaptivity and uniform grid-like performance. Recently, a linear adaptive tile tree approach was presented to provide efficient adaptivity for Maya's Bifrost simulator [14]. A narrow band FLIP method which represented the fluid near the surface as particles only while using a uniform grid inside the surface also provided a decent speedup [5]. Hexahedral grids and tetrahedral meshes have also been used for adaptive fluid simulation [2, 6].

1.2 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 [9, 12, 18]

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.

2 LINEAR OCTREE STRUCTURE

2.1 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.

Our approach uses a linear octree structure stored contiguously in memory. This structure reduces the indirection to the simulation data and avoids the need 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 2.

2.2 Data Layout

The memory for our grid is allocated as a simple array of nodes. Note that allocating this memory contiguously requires that we specify the minimum and maximum cell sizes upfront. As a result, the sizes of individual cells can dynamically change at runtime, but the global resolution of the grid cannot. In practice, this constraint is not a concern because the min and max sizes can be set far enough apart that any reasonably sized simulation can fit in memory. If additional resolution were needed, the simulation would be too large to simulate in a reasonable amount of time. Having an unbounded resolution can lead to memory overflow issues, so bounds are typically enforced in traditional octrees as well. Our structure also uses more memory than traditional octrees. However, our grid will never use more memory than a uniform grid, and in modern machines, in all but the most extreme cases, speed is a much greater concern than memory.

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. [12]. A diagram of a node in our grid is shown in Figure 3.

2.3 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

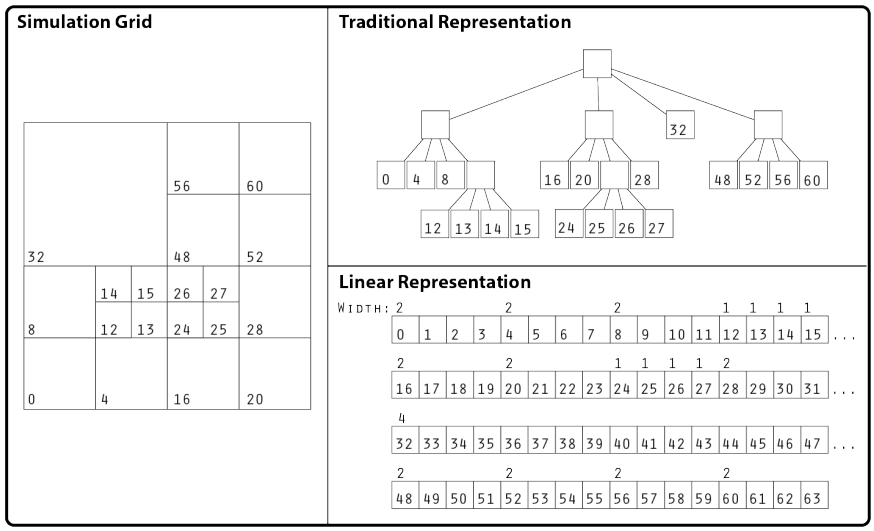

Figure 2: Shown in 2D for clarity, a simple adaptive fluid simulation grid is shown on the left. On the top right the traditional octree representation of the grid is shown. The bottom right shows our storage of the data. Our approach greatly reduces the time required to access data over the traditional octree structure.

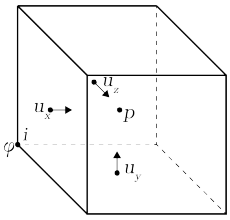

Figure 3: A node in our linear octree grid is shown above. The components of velocity u_x, u_y, and u_z, are stored on the minimal cell face centers. Signed distance ϕ is stored at the corner of each cell to simplify interpolations. Pressure p is stored at the center of the cell.

of this cell's width. Velocity and the 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.

Our criteria for local grid refinement is based on the signed distance field to the fluid-air interface. If the zero level set is contained in a cell, it will be subdivided to the smallest possible size, 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.

2.4 Traditional Octree Implementation

To verify the improvement using our linear octree approach, we also implemented a traditional octree data structure. The simulation logic described in section 3 is shared between the two data structures. The only difference between simulations running on our linear structure and the traditional structure is 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. Implementing both structures and sharing the simulation code allowed us to provide a fair comparison between the data structures alone rather than between simulation techniques.

In our traditional octree implementation, each node, starting with the root node, 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.

3 SIMULATION

The main contribution of this paper is not a new fluid simulation technique. Rather, we are showing that octree fluid simulation methods can be used with our data structure to significantly improve the runtime performance. We do give some detail into our implementation to fill in some of the gaps that the cited techniques lack.

Our fluid simulation algorithm is based on the semi-Lagrangian FLIP method [21]. Our fluid surface representation is done using the level set method [7] with the modifications mentioned in the referenced FLIP paper. For surface tension effects we use the ghost fluid method [8, 10] with mean curvature computed using the Laplacian of our signed distance field.

3.1 Hierarchical Pressure Projection

The pressure projection is the process by which we enforce non-divergence in our fluid. Our technique is based on a previous octree-based fluid simulation method [18] except we adapt it to liquids rather than smoke. This technique performs the pressure projection as a series of uniform solves starting with the largest size in the grid and progresses down to the smallest, enforcing non-divergence at each iteration. 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 3.2)
4: **for** `each size` s_i `in` *sizes* $< s_{max}$ **do**
5: Compute the pressure at s_i (section 3.2)
6: Store the pressure at each cell in the grid at size s_i
7: Subtract the pressure gradient (section 3.3)

3.2 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 1).

$$p = \gamma\kappa \tag{1}$$

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 2). The location where $\phi = 0$ is at a distance θ from the center of the cell being considered (Equation 3).

$$\kappa = \nabla \cdot \nabla\phi \tag{2}$$

$$\theta = \frac{\phi_{i,j,k}}{\phi_{i,j,k} - \phi_{i+s,j,k}} \tag{3}$$

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

Each solve in our hierarchical algorithm is a simple uniform pressure projection that enforces non-divergence. Because each solve uses the explicit boundary conditions from the previous solve, we ensure that fluid remains divergence free through all steps of the algorithm.

3.3 Pressure gradient

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.

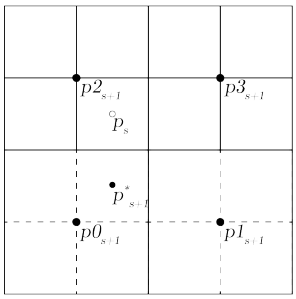

Figure 4: Shown in 2D for clarity, an example of solving for pressure is shown in this diagram. In this example, we are currently solving for the pressure at size s. The pressure for size $s+1$ has been computed in the previous iteration of the hierarchy and is shown stored at the locations $p0_{s+1}$, $p1_{s+1}$, $p2_{s+1}$, 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.

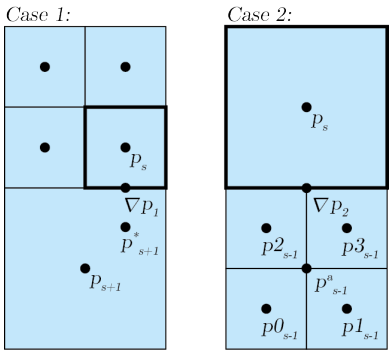

Figure 5: Shown in 2D for clarity, the cases for computing the non-uniform pressure gradient are shown. The equations for computing ∇p_i are shown in Equations 4-7.

For velocity components 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).

$$u(x,y,z) = u(x,y,z) - \frac{\Delta t}{s_i}\nabla p(x,y,z) \tag{4}$$

At t-junction boundaries, special consideration is necessary. Because the fluid-air interface has a buffer of uniform cells around it, we need only consider non-uniform cases at fluid-fluid boundaries. These cases are shown in Figure 5.

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 5).

$$\nabla p_1 = p_s - p^*_{s+1} \tag{5}$$

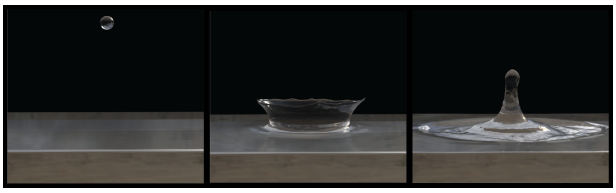

Figure 6: An example of a crown splash simulated using our linear octree structure.

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 6 and 7).

$$\nabla p_2 = p_s - {p^a}_{s-1} \tag{6}$$

$${p^a}_{s-1} = \frac{p0_{s-1} + p1_{s-1} + p2_{s-1} + p3_{s-1}}{4} \tag{7}$$

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. This ensures that our fluid is divergence free.

4 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 in section 2. To verify the difference in speed between the two we used three simulation setups: resting water, crown splashes, and a container filling up with liquid. Table 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.

We were first able to verify the stability of our implementations by observing that resting water simulations remained at rest for various grid configurations.

Next, we tested several different simulation configurations for crown splashes, as shown in Figures 6 and 7. Crown splashes are an ideal candidate for benefiting 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. They are also a good test to verify that our FLIP implementation with surface tension is correct. 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. As seen in Figure 7, our adaptive simulations are nearly identical to simulations running on a uniform grid. This means that for these cases, auto-refinement does not cause any noticeable artifacts.

Finally, also shown in Figure 7, 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.

5 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 adaptive simulation methods 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. Using our method 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.

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.

REFERENCES

[1] Mridul Aanjaneya, Ming Gao, Haixiang Liu, Christopher Batty, and Eftychios Sifakis. 2017. Power Diagrams and Sparse Paged Grids for High Resolution Adaptive Liquids. *ACM Trans. Graph.* 36, 4, Article 140 (July 2017), 12 pages. https://doi.org/10.1145/3072959.3073625

[2] Ryoichi Ando, Nils Thuerey, and Chris Wojtan. 2013. Highly Adaptive Liquid Simulations on Tetrahedral Meshes. *ACM Transactions on Graphics (SIGGRAPH)* 32 (4) (August 2013), 10.

[3] R. Elliot English, Linhai Qiu, Yue Yu, and Ronald Fedkiw. 2013. Chimera Grids for Water Simulation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '13)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/2485895.2485897

[4] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. 2002. Animation and Rendering of Complex Water Surfaces. *ACM Trans. Graph.* 21, 3 (July 2002), 736–744. https://doi.org/10.1145/566654.566645

[5] Florian Ferstl, Ryoichi Ando, Chris Wojtan, Rüdiger Westermann, and Nils Thuerey. 2016. Narrow Band FLIP for Liquid Simulations. *Comput. Graph. Forum* 35, 2 (May 2016), 225–232. https://doi.org/10.1111/cgf.12825

[6] F. Ferstl, R. Westermann, and C. Dick. 2014. Large-Scale Liquid Simulation on Adaptive Hexahedral Grids. *Visualization and Computer Graphics, IEEE Transactions on* 20, 10 (Oct 2014), 1405–1417. https://doi.org/10.1109/TVCG.2014.2307873

[7] Nick Foster and Ronald Fedkiw. 2001. Practical Animation of Liquids. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. ACM, New York, NY, USA, 23–30. https://doi.org/10.1145/383259.383261

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 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.

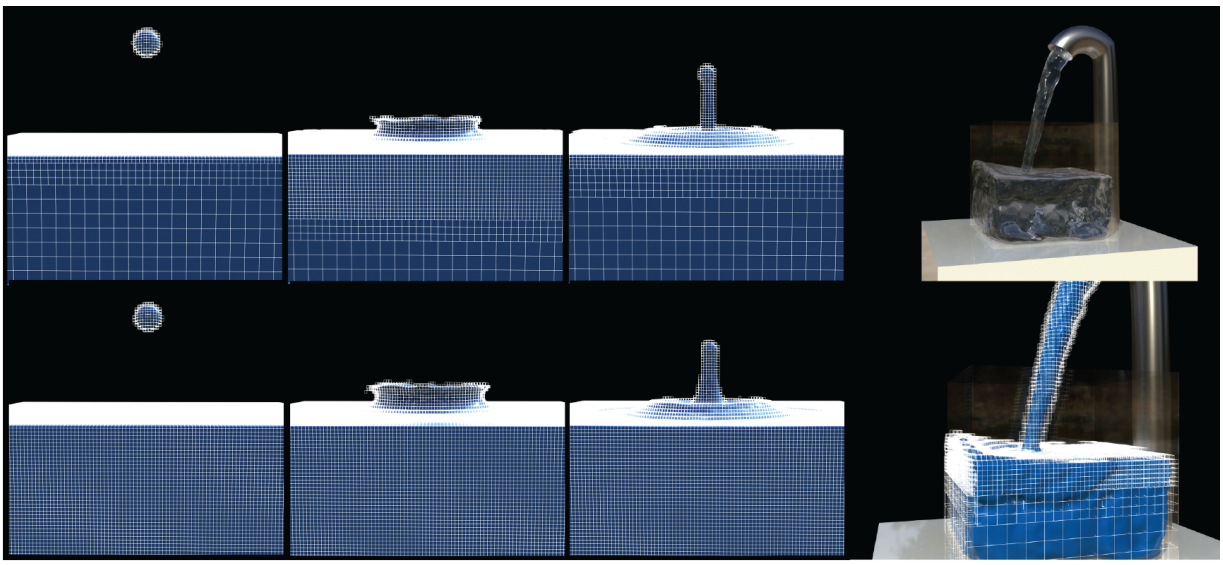

Figure 7: On the left side of this figure is a crown splash being simulated with auto-refinement (top left), and the same splash running on a uniformly subdivided grid (bottom left). Notice that while the top splash took significantly less time to simulate, both splashes look nearly identical. On the right side of the figure is a auto-refined simulation of a container filling with liquid with its grid drawn (bottom right) and without the grid (top right).

[8] Frederic Gibou, Ronald P. Fedkiw, Li-Tien Cheng, and Myungjoo Kang. 2002. A Second-order-accurate Symmetric Discretization of the Poisson Equation on Irregular Domains. *J. Comput. Phys.* 176, 1 (Feb. 2002), 205–227. https://doi.org/10.1006/jcph.2001.6977

[9] Woo-Suck Hong. 2009. *An Adaptive Sampling Approach to Incompressible Particle-based Fluid.* Ph.D. Dissertation. College Station, TX, USA. Advisor(s) House, Donald H. and Keyser, John. AAI3370710.

[10] Myungjoo Kang, Ronald P. Fedkiw, and Xu-Dong Liu. 2000. A Boundary Condition Capturing Method for Multiphase Incompressible Flow. *J. Sci. Comput.* 15, 3 (Sept. 2000), 323–360. https://doi.org/10.1023/A:1011178417620

[11] Michael Lentine, Wen Zheng, and Ronald Fedkiw. 2010. A Novel Algorithm for Incompressible Flow Using Only a Coarse Grid Projection. *ACM Trans. Graph.* 29, 4, Article 114 (July 2010), 9 pages. https://doi.org/10.1145/1778765.1778851

[12] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating Water and Smoke with an Octree Data Structure. In *ACM SIGGRAPH 2004 Papers (SIGGRAPH '04)*. ACM, New York, NY, USA, 457–462. https://doi.org/10.1145/1186562.1015745

[13] Ken Museth. 2013. VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.* 32, 3, Article 27 (July 2013), 22 pages. https://doi.org/10.1145/2487228.2487235

[14] Michael B. Nielsen and Robert Bridson. 2016. Spatially Adaptive FLIP Fluid Simulations in Bifrost. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 41, 2 pages. https://doi.org/10.1145/2897839.2927399

[15] Maxim A. Olshanskii, Kirill M. Terekhov, and Yuri V. Vassilevski. 2013. An octree-based solver for the incompressible Navier-Stokes equations with enhanced stability and low dissipation. (2013).

[16] Stéphane Popinet. 2003. Gerris: A Tree-based Adaptive Solver for the Incompressible Euler Equations in Complex Geometries. *J. Comput. Phys.* 190, 2 (Sept. 2003), 572–600. https://doi.org/10.1016/S0021-9991(03)00298-5

[17] Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation. *ACM Trans. Graph.* 33, 6, Article 205 (Nov. 2014), 12 pages. https://doi.org/10.1145/2661229.2661269

[18] Lin Shi and Yizhou Yu. 2002. *Visual Smoke Simulation with Adaptive Octree Refinement.* Technical Report. Champaign, IL, USA.

[19] Jos Stam. 1999. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 121–128. https://doi.org/10.1145/311535.311548

[20] Bo Zhu, Wenlong Lu, Matthew Cong, Byungmoon Kim, and Ronald Fedkiw. 2013. A New Grid Structure for Domain Extension. *ACM Trans. Graph.* 32, 4, Article 63 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461999

[21] Yongning Zhu and Robert Bridson. 2005. Animating Sand As a Fluid. *ACM Trans. Graph.* 24, 3 (July 2005), 965–972. https://doi.org/10.1145/1073204.1073298