Efficient Fluid Simulation Algorithms

Josh Slocum

April 2015

Abstract

The special effects industry requires photo-realistic fluid simulation techniques to produce visually pleasing images. This involves capturing small scale visual details such as thin films and rolling motions, requiring high levels of computational discretization. Adaptive data structures are optimal for this type of calculation because only a small subset of the fluid requires the full computational discretization. This research presents an adaptive multigrid solver for the Navier-Stokes equations on a k-d tree, compared against the previous standard for an adaptive multigrid solver, an octree.

1 Introduction

Fluid simulation is an incredibly important field of computation, with real-world applications in weather simulation, engineering aerodynamically efficient vehicles, modeling stars and galaxies, and modeling the fluids we interact with in everyday life. There are many branches of this field, each with specific applications. For example, computer graphics focuses on simulating or constructing visually realistic flows with high efficiency, whereas the aerodynamic simulations required to construct efficient airplanes require orders of magnitude more accuracy, without photorealistic rendering. As such, the mathematics and methods used depend largely on the goal of the simulation.

2 Mathematical Background

Many fluids are simulated with a set of differential equations known as the Navier-Stokes equations [15, 16]. These equations describe the fluid as a function of pressure, velocity, and time, and are shown below. In these equations, ρ is the density, v is the velocity, p is pressure, μ is the viscosity, and f is the sum of external forces.

$$\rho(\frac{\delta v}{\delta t} + v \cdot \nabla v) = -\nabla p + \mu \nabla^2 v + f \tag{1}$$

The above equation needs some additional constraints to limit the degrees of freedom, which vary on the type of the simulation. The two types of fluids dealt with in fluid simulations are compressible fluids and incompressible fluids. Compressible fluids are fluids with varying density and incompressible fluids are fluids with constant density. Assuming incompressibility simplifies the above equations, and the methods in which they are solved. There are no purely incompressible fluids in reality, but many fluids, such as water and oil, are basically incompressible at practical scales. Therefore, if the fluid in the desired application varies very little in density, it can be simulated more simply and efficiently, with a small loss in accuracy. As a result, compressible fluid simulations are often used in engineering and other practical applications, whereas most graphics applications tend to assume incompressibility.

Assuming incompressibility simplifies and restricts these equations by using the concept of the divergence of a velocity field. Divergence is a mathematical quantity representing the net value leaving or entering a point. The divergence of a velocity field v can be represented in vector notation as $\nabla \cdot v$, or as $\frac{\delta v_x}{\delta x} + \frac{\delta v_y}{\delta y}$ in two dimensions. Adding the divergence-free restriction to the system of equations allows us to simplify (1). As a result, if the fluid is homogeneous, it has a constant density, and one can remove the density from the equation. Similarly, since many fluids are not very viscous, one can assume an inviscid fluid and remove the viscous diffusion term from the equation. While many graphics applications do assume some viscosity since it smoothes out the computations, it is not necessary. These simplified equations are known as the Incompressible Navier-Stokes Equations, and are shown below.

$$\frac{\delta v}{\delta t} = -v \cdot \nabla v - \nabla p + f \tag{2}$$

$$\nabla \cdot v = 0 \tag{3}$$

The summary of the motion of a fluid can be described by the terms of the above equation. The first term is known as the advection term, and describes the propagation of the velocity field through time. The second term describes how the pressure acts as a counterbalance to the flow of the fluid to keep the fluid incompressible.

Since the approximate computation of these terms may introduce divergence into the velocity field, an extra computational step is needed to ensure that the final velocity field is divergence-free. This step, encapsulated in the second term of (2), is known as the projection step, because it defines a projection of the velocity field onto the space of divergence-free velocity fields.

To understand how this projection method works, one must use the mathematical property known as the Helmholtz-Hodge decomposition. It states that any vector field can be broken into the sum of a divergence-free vector field and a curl-free, or gradient, vector field (figure 1). In the incompressible fluids case, the velocity field v can be represented as

$$v = v^* + \nabla p \tag{4}$$

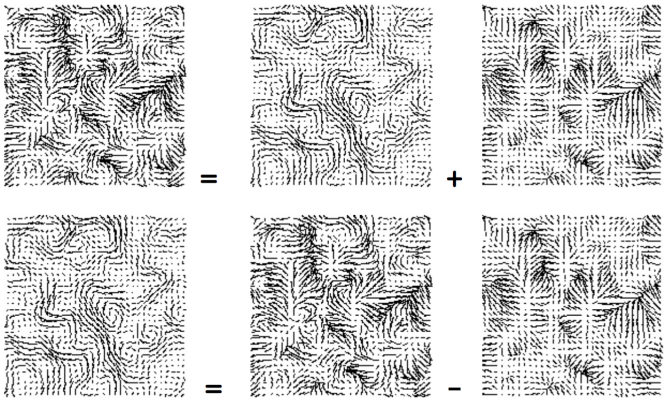

Figure 1: Helmholtz-Hodge Decomposition [16]

where v^* is the divergence free velocity field and ∇p is the curl-free, or gradient, vector field. Therefore, given the velocity v after applying the other terms of (2), we can solve for the final velocity field v^*. Taking the divergence of both sides of (4), we get

$$\nabla \cdot v = \nabla^2 p \tag{5}$$

meaning the divergence of the velocity must be equal to the Laplacian of the pressure. Using all of these equations, one can derive a set of steps for solving the motion of a fluid. [15, 12]

One additional concern is that simulating the fluid in an unbounded area is unreasonable. The simplest and most common method of imposing boundaries for the fluid is defining it to be inside some closed boundary function. In reality fluid cannot move through the walls in or out of the domain, so we must impose similar constraints upon the fluid in our simulation. The simple way to express this constraint in mathematical notation is to say that on the boundary d, with the normal of the boundary defined to be $n_d, v \cdot n_d = 0$ at all points. We must enforce these constraints everywhere in our simulation, regardless of how we choose to discretize it.

3 Approaches to Solving the Fluid Equations

There are two main methods of discretizing the computation of the Navier-Stokes equations: Eulerian method and Lagrangian methods. Eulerian methods discretize the fluid into a grid of cells, where each cell contains information about the pressure and velocity of the fluid within the cell. Lagrangian methods use particles to represent the fluid, with each particle containing a pressure and velocity value. In their simplest forms, both Eulerian methods and Lagrangian methods have problems, but lot of work has been done on improving them.

Jos Stam was the first to solve Eulerian methods on a grid in the computer graphics realm, introducing many methods from the scientific computing community into the graphics community [15]. While his work made large improvements on the previous graphics fluid algorithms, there was still much to be desired. For example, these implicit grid methods were inefficient at large resolutions, which are necessary to get realistic behavior. As such, more advanced methods were developed, from multigrid solvers that increase the rate of convergence, to adaptive quadtrees and octrees [12, 10]. These more advanced methods use multigrid algorithms while only discretizing the computation as much as is necessary in each location. Another constraint is that the Eulerian solvers only define the motion of the fluid at a given time, not the location of the fluid. This means they require additional logic to render a fluid moving through the simulation, such as particle or level set advection.

A huge improvement in Lagrangian fluid simulations arrived with the 1977 invention of Smoothed Particle Hydrodynamics, or SPH. Originally used to model stars in astrophysics [6], this method has since been extended to many other areas of physical simulation such as compressible and incompressible flow, and physical fractures [11]. The idea behind SPH is to consider the values for each particle not just at the point of the particle, but at some area surrounding the particle known as the kernel. Efficient and accurate interpolation schemes can be developed using spatial range queries to determine what particles affect a given point. Thus, a large portion of the efficiency comes from the optimization of these queries, which is usually done through spatial partitioning data structures. One benefit to SPH is that the fluid's location is explicitly determined by the particles, making simple rendering trivial. However, more complicated methods that require surface reconstruction become inaccurate due to the natural smoothing of the kernels.

Fundamentally, all of the improvements to each method have some similarities and tradeoffs. Both methods are slow on their own, and need some more intelligent, adaptive data structure to be practical at a large scale. In the case of Eulerian methods, the adaptive data structure defines the fluid simulation, whereas in the SPH methods the adaptive data structure is an optimization that partitions particles efficiently.

Given this, and my interest in the data structure side of the simulation, I have chosen to focus my work on the Eulerian data structures. A body of work optimizing different hierarchical structures for particle computations has existed for a long time [1, 7], but not as much work has been done on discretizing the Eulerian computation with interesting data structures. Besides multigrid methods, the main other data structure developed has been using "adaptive mesh" data structures that forego the fixed mesh of the trees and multigrids to build a mesh around the interface of the fluid. This approach makes representing the fluid much easier, but adds the very challenging problem of properly propagating the mesh through time while maintaining an efficient, evenly spaced, and conservative discretization [4]. I think there is more untapped potential in the fixed data structure area, which I aim to explore.

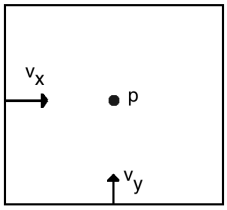

Figure 2: MAC grid representation

4 Eulerian Fluid Solvers

Eulerian solvers take the equations described earlier and discretize them on a grid. There are three main ways of storing values: at the cell centers, at the faces, and at the nodes. The most common method, called a MAC grid, does a mix of these [5]. MAC grids store velocity on the faces and pressure in the center (figure 2).

The discretization of the rest of the math depends on this choice. In my research, I went with the MAC approach, since it is the most common and the projection operator works best on this discretization. Additionally, using a non-staggered grid is known to cause spurious modes to arise over time [14].

Once the values are discretized, we can use the equations to derive an algorithm for solving the motion of the fluid. Given an initial state of the fluid, we can break the computation down into three steps, based off of the terms of the Incompressible Navier-Stokes equations:

1. Advect the velocity field forward in time
2. Solve for the pressure given the velocity field
3. Project the velocity field using the pressure gradient

When this algorithm is applied each timestep, the resulting velocity field properly defines the motion of the fluid, and any number of rendering algorithms can be applied.

4.1 Velocity Advection

The velocity advection part of the solver comes from the advection term $v \cdot \nabla v$. Some attempts have been made at approximating this term at the center of a grid cell [12]. This method stores the velocity at the center, constructs approximations to the new velocity at the cell faces, projects the approximate facial velocities to be divergence free, and finally averages the facial velocities to the cell center. This method is complicated, and faces a common issue among Eulerian solvers known as the "small cell problem", which is where the quantity in a cell could "overflow" and become inaccurate if the timestep is too large.

This is because these methods only use neighboring cells in the computation, which severely limits how quickly information can propagate through the fluid. Common workarounds have been developed, which include limiting the timestep to guarantee this does not happen, or combining smaller groups of cells [12, 5]. All of these methods are complicated and have their own drawbacks.

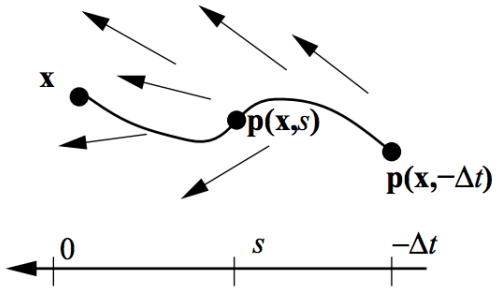

Figure 3: Semi-Lagrangian Advection [15]

Instead of computing the advection term and adding it to the velocity like the previously described algorithm, an alternative is to use the velocities from the last timestep to compute the velocity at the new timestep. If we assume that the external forces term is applied after the advection step, the velocities of the individual parcels of fluid have not changed between timesteps. Thus, one can move the old velocity values forward in time along themselves to get a set of velocity values at new points, at the new timestep. The problem with this method is that no matter how the new points are computed, the resulting points are likely not at the points where velocity is stored. This means that no matter how the new velocity is set from these new values, a large amount of error is introduced into the system.

To alleviate this problem, the velocity values need to be computed exactly at the point they are stored, at the new time step. Instead of tracing forward in time from the old velocities at the old positions, one can trace backward in time along the old velocities from the new positions. Starting from whatever point the velocity is stored, if a particle is traced along the velocity field from the original point to the previous timestep, the new location of that particle represents the location of the fluid from the original point at the previous timestep. Therefore, the velocity at the original point for the new timestep can be set to the velocity at the old point at the old timestep [15] (figure 3). To compute the velocity at the old point at the old timestep, one can use interpolation. If the velocity values at the corners of the cell at the old timestep are known, bilinear interpolation can be used to determine the velocity at the exact location of the particle (figure 4). This method is known as Semi-Lagrangian Advection, and is the method used in [15]. An additional benefit of this method is that using a more accurate particle tracing algorithm gives more accurate results.

Since the MAC discretization stores only the perpendicular velocity compo-

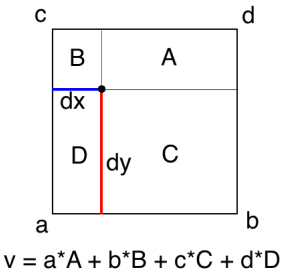

Figure 4: Bilinear Interpolation using the velocity values at the corners

nent at each face, the other component(s) must be calculated to trace backwards from the face. Additionally, the bilinear interpolation requires the velocity at the corners. To accomplish both, the corner, or "nodal" velocities can be computed, then averaged to the faces to calculate the other components.

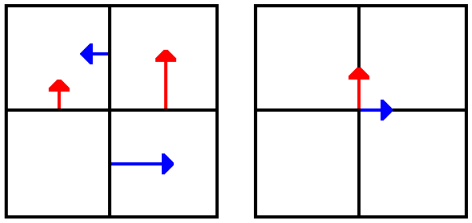

Figure 5: Nodal velocity calculation

This process is depicted in figure 5. In the simple grid, each component of the nodal velocities is simply the average of the two adjacent facial velocities. Since one facial component already exists on each face, we only have to average the other components of the nodal velocities to compute the total velocity at each face.

4.2 Solving for the Pressure

Given the advected velocity v, we can solve for the pressure in the cell c. We can take the integral of both sides of equation (5), getting

$$\int_c \nabla^2 p = \int_c \nabla \cdot v \tag{6}$$

We can use the divergence theorem to convert the left hand side of the equation to produce

$$\int_{\partial c} \nabla p \cdot n = \int_{c} \nabla \cdot v \tag{7}$$

where ∂c is the boundary of the cell c and n is the normal along that boundary. If one assumes the pressure gradient is constant across the face and the velocity divergence varies very little across the cell, this discretization over a cell can be represented as

$$h \sum_{d} \nabla p_d = h^2 \nabla \cdot v \tag{8}$$

where d is the direction, p_d is the pressure value at the neighboring cell in direction d, and h is the width of the cell. To get the discretized version of this equation on a uniform grid, the gradient at a face can be obtained by a first order approximation:

$$\nabla_d p = \frac{p_d - p}{h} \tag{9}$$

Plugging in this into (8), we get the final Laplacian discretization (figure 6)

$$\sum_{d} \frac{p_d - p}{h} = h \nabla \cdot v \tag{10}$$

Since each gradient is a linear combination of pressure values, each cell has a linear equation for its pressure value p. Since there are n linear equations and n unknown pressure values, this discretization forms a linear system of equations. Thus, it can be formulated as a matrix multiply $Ax = b$, where A is the coefficient matrix, b is the velocity divergence matrix and x is the pressure value matrix. There are two main methods that are designed for solving this problem: relaxation solvers and sparse linear solvers.

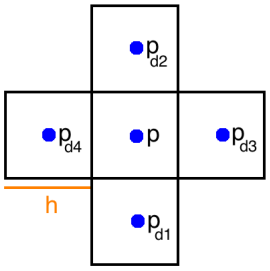

Figure 6: Discretized Laplacian

4.2.1 Relaxation solvers

Relaxation solvers are a class of methods that iteratively solve systems of equations. One widely applicable use of them is solving the Poisson equation $\nabla^2 p = f$. Poisson relaxation solvers work by solving the equation for each cell using values from the neighboring cells. Taking the earlier Laplacian equation (10) and solving for p produces

$$p = \frac{\Sigma_d p_d - h^2(\nabla \cdot v)}{4} \tag{11}$$

in the uniform case (figure 6). Using this equation, one can solve for p by applying this equation at each cell until convergence.

4.2.2 Sparse Linear Solvers

Relaxation is technically a sparse linear solver, but there are other methods that make certain assumptions about the properties of the matrix A that make the computation more efficient. For example, [10] made the tradeoff of being able to formulate A as a positive definite matrix for only using a first-order accurate gradient computation. They were then able to use a Conjugate Gradient solver, which is a more efficient solver than relaxation. Additionally, many non-adaptive frameworks use Sparse Cholesky decomposition. This method uses the Cholesky factorization to transform A, and solve for the pressure solution. The reason this is efficient for non-adaptive frameworks is because the factorization is the expensive part of the solver, so if the grid never changes, A never changes, and the factorization can be computed once and used at every step of the simulation. The reason relaxation and sparse linear solvers are treated separately is because relaxation is often implemented as an integral part of the fluid simulation [12], but the other sparse linear solvers are often called as a black box library to produce a result [15, 10].

I feel that in a comparative study such as my research, having control of the algorithm and having it be directly related to the topology of the data structure is very important. As such, I use the relaxation method in all of my implementations.

4.3 Projection

Using equation (4) to perform the projection, it is evident that the pressure gradients much be computed at the same place as the velocity. In a MAC discretization, the velocity is stored at the faces, so the pressure gradient must be computed at each face. Once this approximate pressure gradient is constructed, it is subtracted from the velocity at each face to obtain the projected velocity. The discretization of these equations is discussed in detail later.

4.4 Optimizations

The relaxation method and most sparse linear solvers are iterative methods, requiring many grid traversals to converge. Since advection and projection require a single traversal, the pressure solver is definitely the bottleneck of these computations. As such, the majority of the research in this area has focused on improving the performance of the poisson solver. Since Jos Stam solved the stable fluid equations, research has branched into two directions. Due to the uniformity of the grid discretization and the spatial locality of the computations, it is possible to attain significant performance improvements by parallelizing parts of the computation using multiple threads and/or the GPU. [3] was the first to attempt this on the GPU, implementing a multigrid version of [15].

The other option for optimization is using data structures that better discretize the computation and improve the convergence of the relaxation operators. These include adaptive multigrid solvers and adaptive trees. Again, since I am interested in the data structures involved, this is the branch of research I chose to follow.

5 Multigrid Methods

A multigrid consists of a series of grids, typically of size $2^n x 2^n$, where n is all integers in the range $l \leq n \leq h$. The purpose of a multigrid is to increase the rate of convergence of the given solver over what it would be for a uniform grid of size $2^n x 2^n$. Multigrids accomplish this in two ways. First, relaxation eliminates high-frequency errors in the grid. By changing the resolution of the grid, low frequency errors become higher frequency and thus can be eliminated by relaxation. Additionally, in the relaxation solver, each cell is populated with an arbitrary initial guess, and then relaxed to convergence. This means the choice of this guess greatly impacts convergence. If the equation with arbitrary inputs is solved on a the smaller grid, then its solution can be used as the initial guess on the larger grid. When this is done, the total time taken by the solver can be much lower, while still achieving the same solution. This transfer of

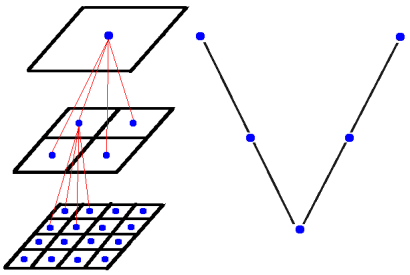

Figure 7: Multigrid V-Cycle

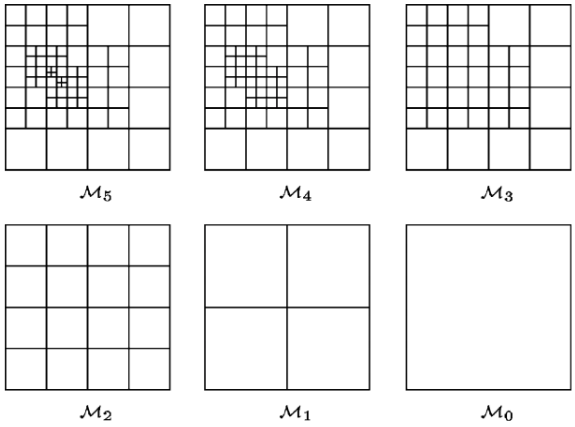

Figure 8: Example Multilevel Hierarchy [12]

relaxation solutions between the series of grids is called a V-cycle (figure 7) [17].

6 Uniform Adaptive Solvers

One problem with the multigrid algorithm is that even though it increases the rate of convergence, it still stores the entire possible discretized space, because it may have to do calculations on any part of it at any given time. Adaptive solvers improve on this by only discretizing the space where it is needed, and thus only discretizing the computation where it is needed. However, the increased complexity of this method requires abstracting multiple concepts.

6.1 Multilevel Hierarchy

In most multigrid solvers, the equation is solved at a particular "level", or grid, and then transferred to the next level. In a uniform multigrid, the level L is all cells on the grid of size $2^L x 2^L$. In a tree, which is structured similarly, the level L is all nodes that have a distance of L-1 from the root, with the root being on level 0, its children being on level 1, etc...

If not all of the cells on a given level exist, we need to use the cell on a smaller level that still occupies the same space. To do this, we define a concept of a "multilevel". A multilevel ML consists of all cells that are either on level L or all leaves that are on a level less than L. More formally,

$$level(L) = \{c | c.level = L\} \tag{12}$$

$$multilevel(ML) = \{c | c.level = ML\} \cup \{c | c.leaf \wedge c.level < ML\} \tag{13}$$

An example multilevel hierarchy is shown in figure 8. Additionally, since neighbors in the multilevel are likely not the same size, more complicated dis-

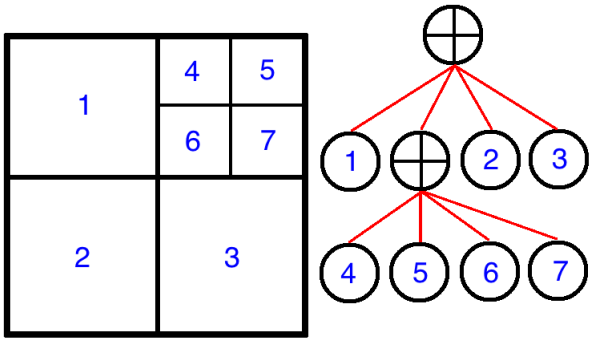

Figure 9: Quadtree

cretizations of the equations must be developed. In general, the computations are phrased in terms of the gradients of values at the faces of the cells, referred to hereafter as face gradients.

6.2 Adaptivity

The other half of the benefit from adaptive data structures is generating the proper hierarchy for the fluid flow. This is generally done by using some heuristic to either expand leaves, or contract cells whose children are all leaves. The choice of heuristic can vary according to preference and use case. Older multigrid methods have tried approximating how much error the result contains and using that to adapt [17]. More recently, [12] found that using the curl to compute a normalized vorticity of the velocity field is a good heuristic for their general fluid solver. They state that adapting when this is greater than some threshold limits the possible angular deviation of a particle traveling across the cell. [10] similarly adapted along vorticity, but adapted its liquid solver based on proximity to the boundary of the fluid.

[12] showed that using the same heuristic for both expanding and contracting can work well, and can limit any potential thrashing between a separate expanding and contracting function. This is done with a heuristic function f that takes in a cell and returns a boolean, which represents whether the given cell requires additional discretization or not. A leaf cell will be expanded when f(cell) is true, and a non-leaf cell will be contracted if all of its children are leaves and f(cell) is false. Additionally, [12] found that in the majority of use cases, the time stamp is small enough that very little, if any, adaptation occurs after one iteration of the adaptation function, so their adaptation algorithm runs once per timestep.

6.3 Case Study: Adaptive Quadtrees and Octrees

Adaptive quadtrees and octrees are the current standard, as far as general purpose, adaptive Eulerian solvers. These are data structures in which each node contains a region of space, and the regions of space in the node's children form an exact cover of the node's region of space. These regions are often axis-aligned squares or rectangles for simplicity. (figure 9). Two different adaptive quadtree and octree implementations are presented below.

6.3.1 Gerris

Gerris, the adaptive solver developed in [12], is credited as having the first implementation of the incompressible Navier-Stokes equations on a quadtree and octree. Gerris uses the fully-threaded tree structure [9] as an optimization, and the curl adaptivity function once per timestep to generate the proper hierarchy. The two key parts of their algorithm are their quadratic interpolation on the restricted tree structure, and their "half V-cycle" relaxation solver.

6.3.1.1 Tree Structure Gerris used quadratic interpolation to get second-order accuracy on all of their calculations. The problem with interpolating at T-junctions is that the center points of the cells are no longer colinear with the face gradient. This requires additional interpolations to be performed if accuracy is a goal. Gerris was able to limit the number of additional interpolations to 1 by restricting the heights of neighboring cells, trading off efficiency of calculation for better representing the underlying function.

6.3.1.2 Relaxation solver The typical multigrid V-Cycle starts at the smallest level, relaxing the solution and transferring it to the next level until it is at the finest level. It then coarsens the result back to the smallest level. Gerris took a slightly different approach, displayed in figure 10. In computing at the pressure correction, it computes f at each level, averaging up the tree, and then relaxes down the tree. This is not quite the same as a classical V-cycle, but is better suited for the hierarchical structure of the tree, and still efficiently finds the solution to the poisson equation. The math will be discussed in detail later.

6.3.2 Losasso's Octree

[10] state that they implemented two main improvements to the algorithm in [12]: They extend the functionality to liquids and remove the restriction on the neighbors in the tree. To simulate liquids, they use a level set interface to denote the presence of the fluid, and use the curvature of the zero level set as part of their surface tension equation. They also switched to the MAC discretization in order to make the projection more accurate.

To remove the neighbor restriction, [10] changed the way the pressure gradients are calculated. Instead of trying to closely approximate the gradients with

Compute R_L on $\mathscr{M}_L$
while $\|aR_L\|_\infty > \epsilon$
  for $l = L-1$ **to** 0
    Compute R_l using weighted average of R_{l+1}
  end for
  Apply relaxation operator $\mathscr{R}(\delta\phi, R_0)$ to $\mathscr{M}_0$ down to convergence
  for $l = 1$ **to** L
    Get initial guess for $\delta\phi$ in cells at level l using straight injection from level $l-1$
    Apply r times relaxations $\mathscr{R}(\delta\phi, R_l)$ to $\mathscr{M}_l$
  end for
  Correct ϕ on $\mathscr{M}_L$ using $\delta\phi$
  Compute R_L on $\mathscr{M}_L$
end while

Figure 10: Gerris' Relaxation Algorithm [12]

quadratic interpolation, previous results state that the approximation will still be convergent if you perturb the pressure from the center by a distance less than the size of the cell. Therefore, even though this method is technically less accurate, the solver will still be able to find a solution. This has two benefits. First, the gradients and Laplacian are much more efficient to calculate. Second, the simplicity of the equations means that unlike the system of equations generated by the quadratic interpolation, this system of equations can be converted into a positive-definite one, for which solvers exist that are much faster than solvers for more general matrices, such as the Preconditioned Conjugate Gradient method that [10] uses.

7 Potential of Non-uniformity

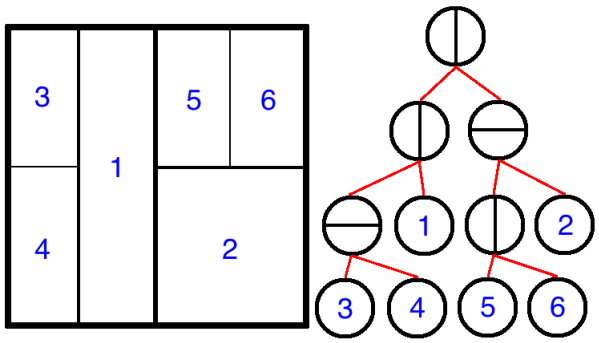

Figure 11: K-D tree

One interesting property of all of the above data structures is that they all discretize the fluid isotropically. For example, cells in quadtrees are axis aligned and have the same width and height. When they split, they split exactly down

the middle in both dimensions, to produce child cells that are still axis aligned and have the same width and height. While this does simplify the math and the calculations, fluids are inherently not uniform, and as such it possible to write frameworks that expose a better adaptive hierarchy.

One paper used this to their advantage. [8] implemented an interesting adaptive discretization. Since pressure deep under the surface of the water almost exactly varies linearly with depth, they represented the bulk of the water with single tall, thin cells that vary linearly in pressure. Then, near the surface, they discretized the fluid uniformly, and made their equations work at the boundary of these two types of cells. Because there was uniformity in the x and y dimensions, the computation could also be efficiently broken up into parallel components. While this is a very specific application for a fluid solver, it is clear [8] used the non-uniformity of the fluid between dimensions to greatly improve the discretization and performance.

7.1 K-d trees

In other fields, the non-uniform, adaptive data structure of choice is the k-d tree (figure 11). The original work describing k-d trees can be found in [2], which describes the structure and theoretical properties of the tree. K-d trees only split the domain in one dimension each time, and are not required to do so at the middle of the cell. As such, they are able to represent the hierarchy of the scene much better than octrees in many cases. Two of the most common applications of k-d trees are raytracing and collision detection, although they are not the only method used. In both areas, k-d trees are used to spatially subdivide the objects and polygons in the scene to decrease the number of intersection computations, whether those computations are objects intersecting with rays or other objects.

More recently, it has been shown that k-d trees can be applied to other physical simulation problems. [13] showed that in the field of radiative transfer, a k-d tree was a definite improvement over the previous standard of an octree.

8 Eulerian K-d Tree

I propose that an adaptive k-d tree is a good data structure for performing Eulerian fluid simulations. My hypothesis is that a k-d tree has the potential to be strictly better than the old standard of an octree, and at the very least the two will have interesting tradeoffs. These tradeoffs are the between the better adaptivity of the k-d tree versus the better tree compression and smaller pressure perturbation of the quadtree.

To perform this comparison, I implement both trees the same in every way possible, with well-defined abstractions so that the main parts that change between the implementations are the computation of the equation inputs. The advection, poisson solver, and projection code is exactly the same.

In terms of the actual algorithms used, I use what I believe to be the best of both worlds between the two previous tree implementations. I use the Semi-

Lagrangian advection scheme for its simplicity, the relaxation solver for its relation to the underlying topology, and the MAC discretizations from [10]. Since the previous examples suggest most of the potential for improvement is the non-uniformity between dimensions, the simple algorithm of splitting at the center of the face for the k-d tree is used. However, exploring non-median splits would be interesting and possibly worth further investigation, despite the increased complexity of the topology.

9 Implementation

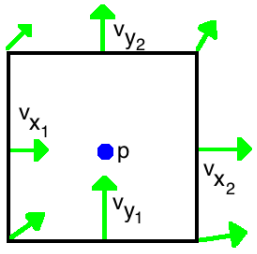

Figure 12: Quadtree/KDtree cell

9.1 Common elements

Since I have well-defined abstractions between the algorithmic layer and the tree layer, a large part of the code works exactly the same between the two implementations, but some of the details on the tree layer are different.

Similar to [10], I store the pressure at the center of the cell. I store the perpendicular velocity component on each face. I also store the nodal velocity at each corner for the interpolation required on adaptivity and advection (figure 12).

9.1.1 Mathematical Discretization

9.1.1.1 Advection Step To perform the Semi-Lagrangian advection, I use the same nodal velocity algorithm described earlier for the interpolation and other facial velocity component. However, this method becomes more complicated when the cell sizes are not uniform, and even worse when the cells themselves are not uniform. The simple case is when the two adjacent facial velocities are on faces of differing size. This means there are multiple valid adjacent face velocities to choose from on the other side. To reduce error, the smallest, and thus closest, face is used (figure 13, left). and a simple weighted average scheme suffices in this case. The much more complicated case is when

a node has only one adjacent facial velocity for a given component (figure 13, right).

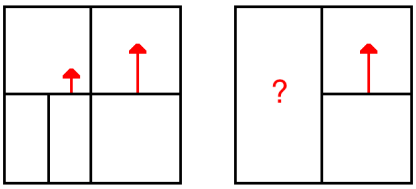

Figure 13: Problematic nodal velocity cases in a non-uniform grid.

Previously, [10] solved this problem by restricting the face velocity at T-junctions to be the average of the face's nodal velocities. However, this introduces error into the simulation and restricts the utility of the adaptation. In my approach, I interpolate the value of the face at the larger cell and average it with the existing value at the smaller cell. This decreases error because it does not ignore the facial velocity adjacent to the node. However, doing so introduces dependencies into the nodal velocity computation, because now nodal velocities are being used to compute nodal velocities. In quadtrees, this is not a problem because the nodal velocities at the T-junction can only depend on nodal velocities at a higher level in the tree, so there is a valid topological ordering of nodal velocities. For k-d trees, this is not the case, resulting in recursive dependencies in the nodal velocity calculation if interpolation is used.

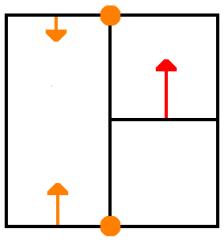

Figure 14: Mixed interpolation scheme in K-d trees

To alleviate this, I use the nodal velocities on the adjacent face and the two facial velocity values (figure 14). This does not have any recursive dependencies since these nodal velocities are on the parent of the smaller node, and the adjacent node, and thus cannot be at a T-junction.

9.1.1.2 Poisson Solver Step I use the same relaxation solver as [12], solving for a pressure correction, using (5). Since the pressure p* from the previous timestep is already known, we can compute a pressure correction dp in order to compute the new pressure p.

$$p = p^* + dp \tag{14}$$

Plugging this into (5) we get

$$L(p) = L(p^*) + L(dp) = \nabla \cdot v \tag{15}$$
$$L(dp) = \nabla \cdot v - L(p^*) = R \tag{16}$$

Given the residual to our system of equations R, we repeatedly relax the pressure correction until convergence, using the following equation

$$dp = \frac{R - A\beta_t}{\alpha_t} \tag{17}$$

In this equation, A is the area of the cell, β_t and α_t and are the sum of the β and α values over all of the cell's face gradients. Each gradient is defined to be

$$\nabla_d p = \beta - \alpha p \tag{18}$$

Where p is the pressure at the center of the cell. In the uniform case (9) β is $\frac{p_d}{h}$, and α is $-\frac{1}{h}$.

To compute the velocity divergence, the divergence theorem is used on the definition of velocity divergence to obtain

$$\int_c \nabla \cdot v = \oint_{\partial c} v \cdot n \tag{19}$$

where n is the face normal, c is the cell, and ∂c is the cell boundary. Thus, since I am storing the normal components of the velocity at the faces, the velocity divergence is just the sum of the velocity components weighted by the length of the face (figure 12). In the uniform case, this equation is simply

$$\nabla \cdot v = \frac{vx_2 + vy_2 - vx_1 - vy_1}{h} \tag{20}$$

9.1.1.3 Projection Step To compute the face gradient, I use the assertion from [10] that perturbing a value by $O(\Delta x)$ from the center of the cell still results in a convergent approximation. I compute the face gradients using a weighted average of the approximate gradients of each neighbor n in direction d, according to the area of their intersecting face (figure 15).

$$\nabla_d p = \frac{1}{width} \sum w_n \frac{p_n - p}{h_n} \tag{21}$$

Then, I use the gradient at each face and the velocity at each face to compute the projected velocity at each face.

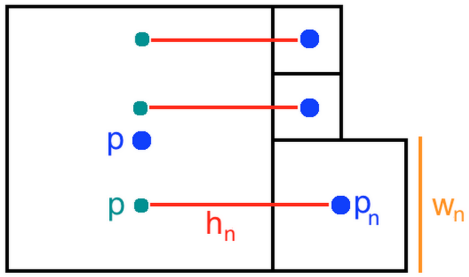

Figure 15: Face Gradient Discretization

9.1.2 Adaptivity

Following the assertions of [12], I run an iteration of the adaptivity algorithm once each timestep. The algorithm looks at each node. If the node is a leaf and f(node) is true, the node is expanded. If the node is not a leaf, all of its children are leaves, and f(node) is false, the node is contracted to be a leaf. Since the values within the newly expanded leaves are not defined, I use a similar interpolation scheme as the one during the advection step. The new nodal velocities are computed using the existing nodal and facial velocities, and then the new face velocities are computed by averaging nodal velocities. The pressure in the children is simply set to the parent's pressure, since the child pressures will be computed exactly in the poisson solver step afterwards. On contraction, I simply delete the children, since their nodal velocities are duplicated in the parent and the parents already have their facial velocities. Again, nothing has to be done with the pressure.

One problem with this approach is if the algorithm is reading and writing from the same tree, the adaptivity can clobber itself as it goes. In this case, reading is defined as computing whether to expand a node or not and computing the new values, and writing is defined as modifying the tree structure and setting the new values. To alleviate this problem, I have two copies of the tree, and compute the adaptivity function on the old one, compute the new values on the old one, then expand the new one and set those new values. I use a similar technique during the Semi-Lagrangian step to avoid the advection clobbering itself and setting incorrect values for the advected velocity.

9.2 Different Elements

The two main differences in mathematical implementation are the recursive discretizations of the gradient and Laplacian operators.

The main difference in the implementation of the above functions is keeping

track of the neighbors and levels for each cell. Since the level of the cell is used everywhere in the code for multilevel logic and to compute the width of the cell, it had to be well-defined in the case of the k-d tree. However, computing a single level is not enough, because the node differs in width in each dimension. Therefore, I introduce the concept of a level for each dimension, which is defined as the number of times the root has split in that dimension to produce the given node. In the two dimensional case, this means there is a separate level for x and y that is used when calculating cell width and height, but the overall level used in the multilevel calculation is still a single level. This overall level is the same as the quadtree, and is still defined to be the distance from the root minus one, which happens to be the sum of the dimension levels.

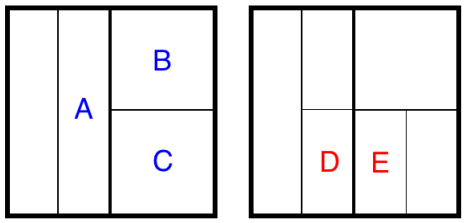

Figure 16: K-D tree bad neighbor situation

Additionally, the neighbor computation becomes more complex. Consider the k-d tree in figure 16. In a quadtree, it is guaranteed that every node with a spatial neighbor at the same level will be able to keep track of each other with neighbor pointers that take constant time to set. On the k-d tree, this is no longer true, due to two problematic cases. Case 1 happens at nodes A, B, and C. B and C nodes don't have a neighbor at the same x and y level in the negative x direction, and A doesn't have a neighbor at the same x and y level in the positive x direction. Case 2 occurs at nodes D and E. These two nodes are neighbors at the same x and y level. But, since their parents don't have neighbor pointers in the x direction, D and E cannot compute their neighbor in constant time. It is possible to still compute the neighbor pointer in logarithmic time using level coordinates stored in each cell. This is advantageous to do, since the neighbor pointers are set once and used multiple orders of magnitude more times in the relaxation steps.

The other important difference is in the adaptivity function. For expansion, the k-d node must determine the split dimension, along with whether or not to split. Additionally, the method of choosing the split dimension should be related to the method that determines general adaptivity. For the adaptivity functions that use velocity, I split along the dimension with the higher velocity gradient. One concern with the non-uniformity is that if the width and height of the cell become too different, it may introduce too much error into the calculation. Thus, I restrict the each node to have a maximum difference in x and y level, which can be specified by the user. As a result, when the adaptivity code is

called, if splitting a node in one dimension would cause this difference to be too large, it splits in the other one. The behavior of contraction is unchanged and identical to the quadtree.

9.3 Other Details and Obstacles

One additional difficulty in getting the poisson solver to work is the error in the discretization of the poisson function. In the continuous case, the integral over the domain of the Poisson equation $\nabla^2 p = f$ is guaranteed to be zero. A proof of this is using the divergence theorem is shown in equation (22), when combined with the zero-gradient boundary conditions. In the discrete case, the integral over the domain is the spatially weighted average of f. Therefore, if there is a non-zero weighted average of f, it is due to the discretization of the function. Certain flows can have much more error in this average than others, especially in multilevels with many T-junctions. This error then gets magnified during the relaxation process, and prohibits the solver from converging. Since the error should be 0 in the continuous case we are trying to model, the simplest fix is to remove the error in the average by taking the spatially weighted average of the residual in each cell, and subtracting that average from the residual in each cell before the relaxation process begins. This enables the relaxation process to converge more quickly and correctly.

$$\int_c \nabla^2 p = \int_c \nabla \cdot \nabla p = \int_{\partial c} \nabla p \cdot n. \tag{22}$$

10 Results

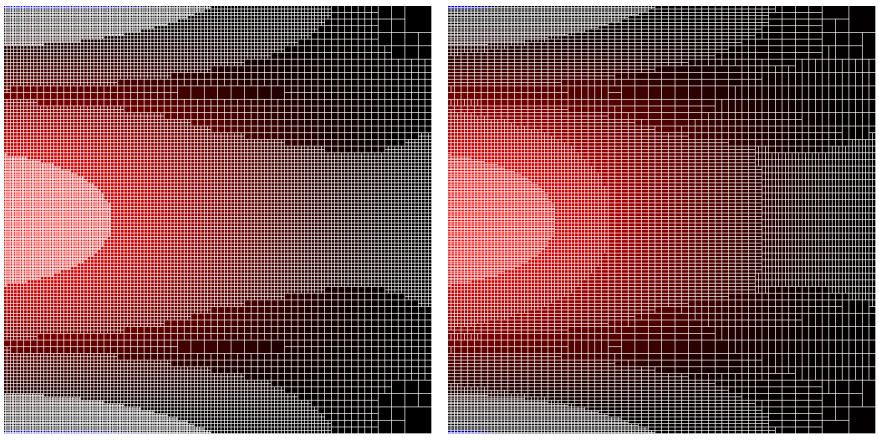

Figure 17: Quadtree and K-D tree discretizations of the same function

10.1 Adaptivity to a Function

Given the same adaptation scheme, it is fairly clear that the k-d tree discretizes better to problems which have non-uniformity between the dimensions. I performed multiple tests in which I input a velocity field and run adaptation on both a quadtree and a k-d tree until they both reach a stable state. This effectively forces both trees to represent the same topology. To perform the adaptivity, I use the norm of the velocity gradient as the adaptive criteria, and for the k-d tree I use the magnitude of the velocity gradient components to determine which dimension to split. For each test, I used multiple different thresholds for the adaptation function in order to produce topologies with a range of detail.

$$f(x, y) = (1 - cos(2\pi x))(1 - cos(2\pi y)) \tag{23}$$

The first test was inputting the velocity components as the x and y partial derivatives of function (23). The velocity field of this function effectively forms a sink at the center of the domain. The results of this test show that the k-d tree represents the function with 2 to 15 percent less nodes, and 32 to 43 percent less leaves than the quadtree.

The next two tests used the same method, but the function

$$f(x, y) = cos(\pi kx)cos(\pi ly) \tag{24}$$

where k and l are variable. This forms a function with multiple sources and sinks in the domain. I ran two tests with this function. The first test with function (24) is a uniform test with both k and l equal to 2, and the second a non-uniform test with k equal to 7 and l equal to 3. In the first test, the k-d tree has between 6 percent less and 32 percent more nodes than the quadtree, and 11 to 37 percent less leaves than the quadtree. In the second test, the k-d tree has 4 to 32 percent less nodes and 35 to 55 percent less leaves than the quadtree.

The final test used the function

$$f(x, y) = (2x^3 - 3x^2 + 1)((2y - 1)^4 - 2(2y - 1)^2 + 1) \tag{25}$$

This function was chosen to be a non-uniform polynomial function between both dimensions. Given the zero-gradient boundary conditions, I fitted a cubic and quartic function to the domain [0,1] such that the derivative for each function at both 0 and 1 was 0. With one function for each dimension, the composite function approximately models a fluid flowing toward the middle left side of the domain. The results of this test show that the k-d tree represents this function with 5 to 33 percent less nodes, and 36 to 55 percent less leaves than the quadtree.

10.2 Poisson Solver Performance and Error

To test the performance and accuracy of the poisson solver, I use the same functions as the adaptivity tests, and similarly set the velocity field to the partial

derivatives of the function. Since the Laplacian of a function is the divergence of its gradient, if the velocity is set to the gradient of f, then the solution of the poisson equation should be exactly f. Thus, we can test the accuracy of the poisson solver by comparing the final result at each cell center to the true value. In theory, if the k-d tree is in fact able to better discretize the function, it should achieve less error with faster performance.

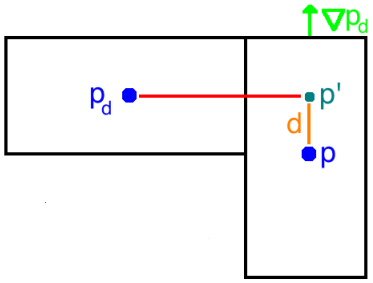

Figure 18: K-D tree T-junction

In many cases, especially (23) it is clear the k-d tree plateaus off before the quadtree, due to the increased error of the perturbing discretization. In figure 19 we compare the result of discretizing to (25) vs (23). Since (25) is not uniform, the k-d tree is able to discretize to it well. However, not only does the k-d tree do better on non-uniform grids, it actually does strictly worse on uniform grids. In figure 18 we can see that a large perturbation in pressure is required at non-uniform T-junctions, compared to the width of the cells. While this approximation is still convergent, it introduces a lot of error into the simulation. One of these poor-performing T-junctions is visible at the center of the middle image in figure 19, whereas none of these exist on the left, and many can be seen in the image on the right. This supports the hypothesis that the poor discretization of the Laplacian on these bad T-junctions is the cause of the error. On top of that, the tests show that when the k-d tree does not plateau off due to the discretization, it performs much faster, but when the k-d tree has more error it performs slower. This is likely due to the fact that increased error results in decreased convergence of the relaxation solver, and thus decreased performance.

To attempt to alleviate the error problem, I introduced a pressure interpolation scheme into both the k-d tree and the quadtree (figure 18). I use a first order approximation computing the interpolated pressure p' with

$$p' = p + d\nabla p_d \tag{26}$$

This method is still convergent because it uses the same Laplacian computation as before, with the only difference being the input pressure values. The

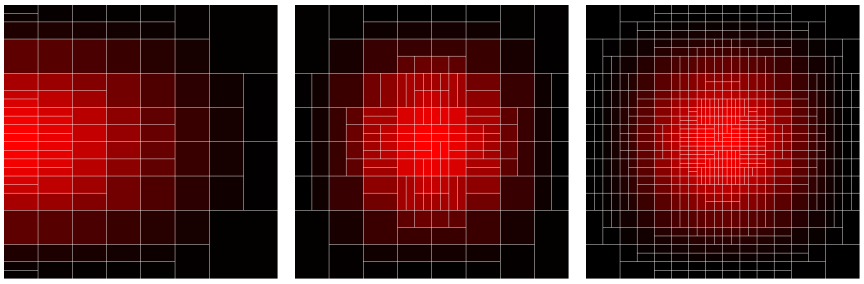

Figure 19: Comparison of bad T-junctions in K-D tree discretizations

interpolation method runs on face gradient and Laplacian computations when face is at a T-junction. One problem with the k-d tree version of the interpolation is that a cyclic dependency can be formed when the gradient calculation in an interpolation needs an interpolation istelf. To solve this, I cap the number of interpolations that occur before returning to the old perturbation discretization.

Similar to the adaptivity tests, we can see that the k-d tree's poisson solver performs much better on functions that are not uniform in the x and y dimensions.

One interesting trend is that enabling pressure interpolation is almost always worse on the quadtree, and enabling pressure interpolation on the k-d tree had variable results. It is clear that the k-d tree performs much worse on the uniform functions cos and cos22, even with pressure interpolation. On the functions without uniformity between the dimensions, results are inconclusive, and more tests will be needed to identify any trends in this area. It is already clear by looking at the graphs that the k-d tree has more variance in its performance than the quadtree. A couple interesting points are that on the xy function, the k-d tree appears on pace to beat the quadtree, but then hits the limits of its discretization at around an error of .001, whereas the quadtree can still get less error with more discretization. However, on the cos73 test, the k-d tree never hits a discretization plateau and is able to achieve the same error as the quadtree with around 2 to 4x less computation on high discretizations.

These results suggest that on larger non-uniform flows the k-d tree may be able to greatly outperform the quadtree with a better discretization, but the important result would be to determine if the k-d tree can keep pace with the quadtree on uniform flows, especially one such as the cos test where it hits a high error plateau.

10.3 Test Improvements

While the number of nodes and leaves given a threshold is a decent indicator of either tree's ability to adapt to a function, the measurement that really matters is how much error there is in the actual discretization of the function f. The discretization of f at a point is computed with the quadratic interpolation of

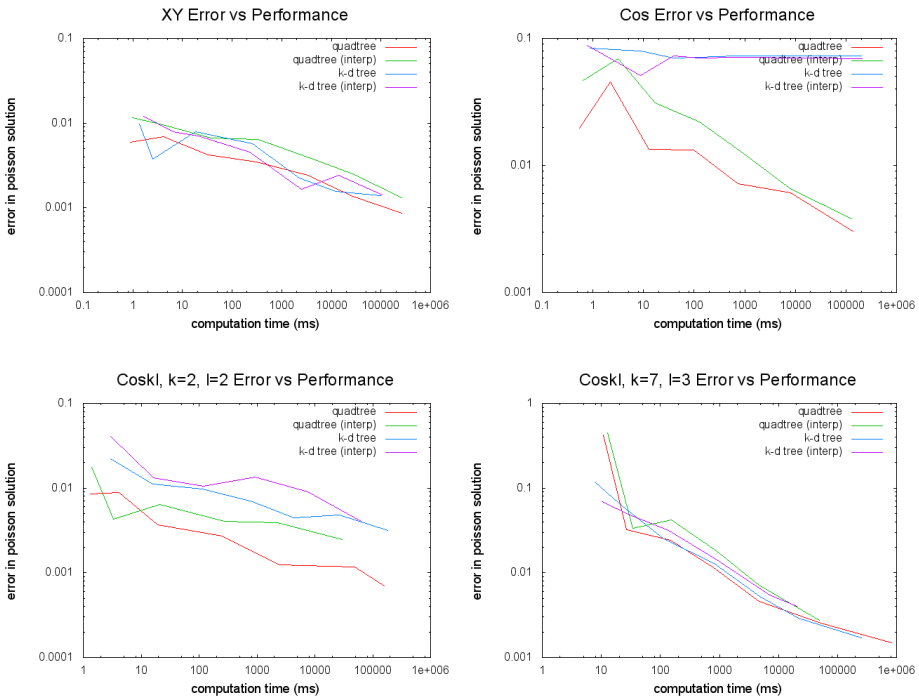

Figure 20: Poisson Solver Performance

the nodal velocities. Thus, to compute the error in the total discretization of the function, one could set the nodal velocities to the exact partial derivatives of f at that point, and take the integral of the difference between f and the discretization of f, over the whole domain.

This equation becomes

$$\Sigma_c \int_c \| < \frac{df}{dx}, \frac{df}{dy} > - < I_x(x,y), I_y(x,y) > \| \tag{27}$$

Where $I_x(x,y)$ and $I_y(x,y)$ are the interpolated values of the x and y velocity. The total error over the quadtree and k-d tree for the same threshold could be computed. The end result of this would be that the comparison of number of nodes and/or leaves to error would be much more useful in determining which tree is better at discretizting than a comparison of number of nodes and/or leaves to threshold.

More functions could be used for both tests to provide a larger sample of results. However, since the results are very clearly correlated with the uniformity between the dimensions, this may not be necessary. Additionally, since the two functions I tested were polynomial and periodic, and any function can be represented with a Taylor decomposition or a Fourier decomposition, I would

expect to see similar results, although the adaptivity thresholds would likely need to be different.

11 Conclusion and Future Direction

Fluids are non-uniform between the dimensions, and [8] was able to leverage this to improve the simulation capabilities of Eulerian solvers. The more general version of non-uniform tree solvers is a k-d tree, and as such the k-d tree should be able to discretize the fluid's space much better than the old standard of an octree. Building on the quadtree and octree work of [12, 10], I implemented an Eulerian k-d tree solver and tested it against an Eulerian quadtree solver.

While k-d trees are able to discretize non-uniform fluids better than quadtrees, they suffer from increased error in the pressure discretization I used, and as a result perform worse on fluids that are close to uniform between the dimensions. K-d trees have the potential to outperform quadtrees given a better discretization, but they are much more variable and the quadtree is likely the safer choice.

The largest improvement to this method would be to develop a pressure discretization method that utilizes the non-uniformity of the k-d tree to decrease error and improve performance. If k-d trees cannot keep pace with quadtrees on uniform flows, they will likely not be used, even if they can outperform quadtrees on non-uniform flows.

The next modification would be to use a level set method similar to [10] to perform liquid simulation. I would expect multiple interesting results to emerge. The first of these is that the k-d tree will likely discretize very well around the interface of the fluid, since the velocity is very much not uniform in that region. The second would be on deep water simulations, similar to [8]. If the water's interface is not the adaptive criteria used, I would expect the pattern of taller, thinner cells to emerge naturally.

Since the tradeoff between tree compression and increased non-uniformity would be exaggerated in an octree vs a three dimensional k-d tree, it would be interesting to use these same equations on both of these trees and compare the results there.

Also, [10, 12] included math to collide the fluid with solid objects, which is only a slight abstraction up from the math presented in this paper. I did not see this as a necessary feature, but I do think it is probable that a k-d tree would discretize better around this interface than an octree.

As mentioned earlier, it would be interesting to see if the increased potential for following the flow by splitting at points other than the median would be worth the increased complexity of the discretization and increased computational time for the adaptivity. Regarding the increase in computation time, the adaptivity already takes much less time than the poisson solver, so since it is far from the bottleneck, the increase in computation time at this part might not be significant, but may compensate by increasing the convergence of the poisson solver.

Even though I chose the relaxation operator for the poisson solver, my discretization would likely work with the sparse matrix solvers. It would be interesting to compare the different performance characteristics of the quadtree and k-d tree given the same sparse linear solver library.

12 Acknowledgements

This work was done as the undergraduate honors thesis requirement for the Turing Scholars Honors Program at The University of Texas at Austin.

I would not have been close to completing this project without my second reader Dr. Vouga helping me to better understand the problem I was trying to solve, or my advisor Dr. Fussell getting me started early.

References

[1] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

[4] Pascal Clausen, Martin Wicke, Jonathan R Shewchuk, and James F O'brien. Simulating liquids and solid-liquid interactions with lagrangian meshes. *ACM Transactions on Graphics (TOG)*, 32(2):17, 2013.

[5] David Cline, David Cardon, and Parris K Egbert. Fluid flow for the rest of us: Tutorial of the marker and cell method in computer graphics.

[6] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.

[7] Lars Hernquist and Neal Katz. Treesph-a unification of sph with the hierarchical tree method. *The Astrophysical Journal Supplement Series*, 70:419–446, 1989.

[8] Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 805–811. ACM, 2006.

[9] Alexei M Khokhlov. Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations. *Journal of Computational Physics*, 143(2):519–543, 1998.

[10] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics (TOG)*, 23(3):457–462, 2004.

[11] Joe J Monaghan. Smoothed particle hydrodynamics. *Reports on progress in physics*, 68(8):1703, 2005.

[12] Stéphane Popinet. Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries. *Journal of Computational Physics*, 190(2):572–600, 2003.

[13] Waad Saftly, Maarten Baes, and Peter Camps. Hierarchical octree and kd tree grids for 3d radiative transfer simulations. *Astronomy & Astrophysics*, 561:A77, 2014.

[14] Mark R Schumack, William W Schultz, and John P Boyd. Spectral method solution of the stokes equations on nonstaggered grids. *Journal of Computational Physics*, 94(1):30–58, 1991.

[15] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[16] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[17] MC Thompson and Joel H Ferziger. An adaptive multigrid technique for the incompressible navier-stokes equations. *Journal of computational Physics*, 82(1):94–121, 1989.