4-25-2005

A Benchmark and analysis of spatial data structures for physical simulations

Domingo Lara
Trinity University

A Benchmark and Analysis of Spatial Data Structures for Physical Simulations

Domingo Lara

A departmental thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for Graduation.

April 15, 2005

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

A Benchmark and Analysis of Spatial Data Structures for Physical Simulations

Domingo Lara

Abstract

Collision detection is an issue in physical simulations; without it simulations are inaccurate. Unfortunately, effective collision detection can require a significant amount of computational power. To reduce the number of computations and make the problem more tractable, computer scientists have used data structures to partition the system. This removes the need to have every single particle check for possible collisions with every other particle in the system; however, generic data structures typically do not work as well as specialized data structures, so this has led to the creation of multiple spatial data structures. Some spatial data structures and algorithms were customized and created to optimize memory usage while others have been made to increase speed. This project seeks to compare spatial data structures in systems with uniformly and non-uniformly distributed particles, while varying the number of particles and the filling factor. The results of this project should provide useful information to those doing general collisional simulations, such as physicists and engineers.

Acknowledgments

First and foremost, I would like to thank my parents who morally and financially supported me throughout my education. To my friends, both in Trinity and outside Trinity, who believed I could eventually finish this thesis. To Inga Munsinger, who proofread much of my thesis out of kindness. To Dr. Berna Massingill and Dr. John Howland who took on the extraordinary task of proofreading the whole thesis and tackling various grammatical atrocities. Furthermore, without their help this thesis would have been written in a lesser typesetting program. I thank Dr. Gerald Pitts, who initially inspired me to do research. Finally I thank Dr. Mark Lewis, who suggested this project, patiently provided help and advice throughout this project, and aided in proofreading.

A Benchmark and Analysis of Spatial Data Structures for Physical Simulations

Domingo Lara

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer simulations face a fundamental trade-off between time and accuracy. Scientists who want a very detailed simulation will need to be patient for the results. Scientists who want a fast simulation may need to be content with a less elaborate simulation. Trade-offs are not new to computer science and have motivated the study and formation of new algorithms.

Collision detection and response is an issue in spatial and physical simulations that affects scientists and engineers. It is crucial to know if two objects will collide with each other in a given time-step. Unfortunately, determining if two particles will collide is not a computationally easy task. In fact, collision detection is often seen as the computational bottleneck for many simulations and is a topic of research for many people.

1.1 Collision Detection

To fully appreciate the utility of non-exhaustive collision detection methods it is helpful have some understanding of the computations needed for detection.

First, we assume the particles are perfect spheres and particles can only travel in straight lines between timesteps. First calculate the sum of the first particle's radius and the second particle's radius and call the value r. Next calculate the distance between the two particles and call this value d. If $r > d$, then the particles have collided with each other.

The described method is regarded as a rudimentary and poor collision detection algorithm for anything other than a system of spherical objects. If this type of collision detection were used in a spatial simulation, the collisions would appear grossly inaccurate. There are better algorithms for determining if a particle and a plane intersect, if a particle collides with a triangle, and so on. Each of these methods is more complicated than sphere-to-sphere collision detection and borrows heavily from vector calculus. To better detect collisions, the programmer should have some understanding of normal vectors, dot products, and some trigonometry.

The problem becomes even worse if the examined polyhedra are not convex. For a realistic animation or virtual reality simulation, it is sometimes helpful to have good collision response, which requires the system to know exactly which face the particle is colliding with and where in relation to the center of the object the collision is taking place.

Due to the computational nature of collision detection, we want to execute such expensive methods only when a collision is possible. If two particles are near each other and are moving toward each other, then it is reasonable to to check the particles. If two particles are not near each other or are not moving toward each other, then it may not be necessary to check them. This intuitive notion of spatial locality motivated the creation of spatial data structures.

1.2 Purpose

This project seeks to do a rigorous analysis of spatial data structures for sheared boundary physical simulations. It compares the performance of various data structures when particle distribution, filling factors, and numbers of particles are varied. For this project, five data structures were investigated: fixed grids, variable grids, quadtrees, KD-Trees, and PSKD-Trees. This group provides an overview of traditional and newer data structures. Obviously, not all data structures can be benchmarked and analyzed as there are significantly more variations on several of the aforementioned structures.

Of the seven data structures, fixed grids are the most popular followed by quadtrees, and KD-trees are also popular but not as much as fixed grids. Variable grids and the PSKD-tree are less known. These lesser-known data structures were preferred over other lesser known data structures [4, 6] because of the ease of finding literature and information on them, and the direct applicability to simulations of interest.

This project is significant because it can help those who are designing simulations to make better decisions given their problem domain. Naturally, this research would have a more direct application to those who are doing simulations on systems that have large numbers of particles that collide, since this research on spatial data structures is based on how efficient they are when their data evolves over time. Virtual reality, complicated particle simulations, planetary formation, and granular flow simulations are possible domains that might be affected by this research.

Some may argue that this research will become less significant as parallel computing becomes more prominent. They may claim should a serial simulation take too long to finish, simply build a parallel version of the program. However, this assumes all serial simulations can easily be converted into parallel programs and neglects the fact that this would mean

redesigning many simulations that are currently in use. Furthermore, parallelizing these legacy systems may mean greatly redesigning several classes and modifying lines of code. While changing a spatial data structure may require coding, the required time to code a data structure should be much less than the required time to redesign a serial simulation.

Others may claim that as hardware speeds increase, having a fast spatial data structure may matter less. But the is no guarantee that hardware speeds will increase at the same rate as the complexity of simulations. Furthermore, a minimal speed boost in some cases may erase hours (or more) from simulation run times. Also, if a simulation does not need to be finished at an earlier deadline, the increased speed allows for greater complexity and detail. To many researchers, there can never be enough detail.

1.3 Review of Literature

This project was suggested by Dr. Mark Lewis who felt there was a need for an analysis of spatial data structures. A review of the literature later showed he was correct. There are plenty of articles on parallel data structures, parallel simulations [13], and benchmarks on certain kinds of data structures [12], but none on data structures specifically for simulations. The few remotely relevant data structure articles dealt with database and time-evolving data. The literature does not address PSKD Tree, PR methods, or variable grid methods in regards to particle distribution, filling factor, or population size. According to popular opinion, fixed grids should outperform all the other methods assuming a uniform distribution of particles, and trees may do well in non-uniform. In theory, grids will dominate in an uniform system because they have constant lookup and retrieval time and can be built quickly. It is also believed that since grids are poor at adapting, trees would be superior to grids in a non-uniformly distributed, constantly changing system. We wish to test these

popular ideas and to see what happens with different number of particles and filling factors.

Chapter 2

Simulation Background

2.1 Simulation Theory

There are several paradigms for modeling a given system, and there are times when certain models or combinations of models are easier to work with. For example, if the goal of a system deals with the state of an object, then declarative modeling might be an optimal modeling strategy; or, a system that deals with changes and events might be easier to model with a functional approach.

The simulation that was used to benchmark these data structures was a blend of several modeling paradigms. It has the strengths and ease of both constraint-based modeling and entity-based spatial modeling.

2.1.1 Constraint-Based Modeling

Unlike other modeling approaches, constraint-based modeling emphasizes the rules, constraints, or equations that a system must obey. With a declarative or functional model these requirements may not be obvious or difficult to maintain. Furthermore, declarative

and functional models are driven either by the notions of direction or state, but a constraint-based model are driven by the need to maintain equality and balance.

Constraint-based models are popular with physical systems, some biological systems, or any domain where the events of a system can be expressed as a difference or differential equation.

2.1.2 Spatial Modeling

Spatial modeling is similar to constraint-based modeling except that the location of objects matters. There are two types of spatial modeling: entity-based and space-based. The former cares about the locations of individual particles and objects; the latter cares about groups and regions of objects. For collision detection, an entity-based approach is used because interactions among individual particles is important. This means that the state vector for at least one object in the simulation must have location coordinates and/or a velocity vector. This approach is also popular for physical systems.

2.1.3 Multi-Models

This simulation is a multi-model; it has constraints and needs to contain spatial information. Each object is associated with a state vector containing the object's x, y, z coordinates and its velocity vector. Spatial information is obviously important for keeping track of how the system evolves and for collisions.

In this system, momentum needed to be reasonably conserved. To maintain our physical constraint, a sheared boundary is integrated into the simulation system. This boundary deals with wrap around issues with particle trajectories and increases the velocity of certain particles.

However, the introduction of spatial and constraint-based modeling leads to the question

of how to keep of track of time as the system evolves. There are two conventional methods for doing this: time-slicing and event scheduling.

2.1.4 Time-Slicing

To represent time, some simulations have a time variable, and simulate the passage of time with a loop increments the time variable. Time-slicing divides time into n equal segments and increments the simulation clock by n at each iteration. However, if events that change the system state do not typically happen at each timestep, then this method wastes time on computations that are not worthwhile.

2.1.5 Event Scheduling

Event-based simulations attempt to remove the non-worthwhile computations that a time-sliced simulation might have. If only events that change the state of a system are worthwhile and it is known when these events will occur, then there is no need to spend time performing computations for the timesteps in between. For example, in queue simulations if the arrival times of the individuals and the service time of the processors is known, an event-based simulation implemented with a priority queue would reduce needless computation and overhead.

2.2 Methodology

To determine which data structure was optimal, various simulations and runs were executed. In each simulation, either the filling factor, number of particles, particle distribution, or the spatial structure was varied. Simulations were run on Linux machines equipped with Intel Pentium 4 at 2.4 GHz and 512 MB of RAM and running Fedora Core 2. The programs were

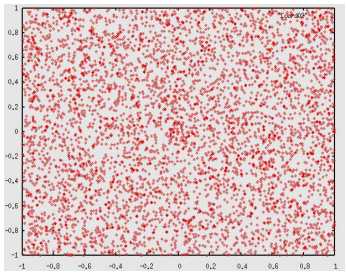


Figure 2.1: Uniform

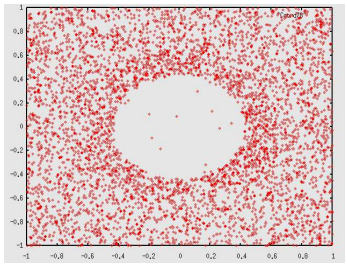


Figure 2.2: NonUniform

compiled with "g++ -Wall -pedantic -O7" and debugged with gdb; due to the computational nature of the simulations the optimization level was set to maximum.

Each data structure was tested in a uniform system with 100, 300, 1,000, 3,000, 10,000, 30,000, and 100,000 particles. In non-uniform systems these data structures were tested with 100, 300, 1,000, 3,000, 10,000, and 30,000 particles; in non-uniform tests 100,000 particle runs were omitted because they took significantly longer than uniform simulations and it was not feasible to keep the simulation running long enough to get results.

To confirm that these systems had either an uniform or non-uniform distribution for the duration of the simulation, the coordinates of the particles were mapped with gnuplot and the number of collisions was examined.

To vary the distribution of particles a repelling force was added into the system. Determining the amount of force this repelling force needed to produce was not as trivial as initially thought, since sheer force and extreme velocities kept reproducing an uniform distribution.

In order to reach a conclusion, these numbers were then entered into a table and various graphs. All of these simulations used the existing numerical framework that was written by Dr. Mark Lewis for his astrophysics research.

2.2.1 Filling Factors

Deciding on which data structure is optimal for a given system may depend on how concentrated the particles are in a system. For example, a sparse system may encounter few collisions, whereas a densely populated system may experience a large number of collisions. Filling factor referred to how much of simulated region's volume was occupied by the particles.

2.2.2 Number of Particles

In addition to varying the filling factor of the systems, the project sought to find out how the data structures scaled with different population sizes. The number moving particles in a closed system is directly related to the number of collisions. For this research, tests with 100,000, 30,000, 10,000, 3,000, 1,000, 300, and 100 particles were used. While it is unfortunate that not the 100,000 particle simulation runs could not be done for non-uniform systems, it is still possible to draw conclusions with the data that was gathered.

2.2.3 Particle Distribution

To better understand how the data structures could adapt and deal with different systems, this project tested data structures on systems that had uniformly and non-uniformly distributed particles. These two distributions should provide useful comparisons to scientists and engineers.

2.3 Existing Simulation Framework

In the physical simulation that these data structures were geared for, the data structure indexes particles from a *Population* class. The data structure uses a template of the *Popu-*

lation class to call methods. To determine if two particles are having a collision the given structure invokes a method from the *Collision Handler* class. This *Collision Handler* then performs tests on the two particles.

These simulations were run in a previously written simulation system, which was written in C++. This system simulates the motion of particles in a closed space. In the main program, first the number of particles and filling factor is gathered from the command line, then various classes are initialized. The following equation illustrates the relationship between the particle radius and filling and determines the radius size:

$$radius = \sqrt[3]{\frac{size_x \times size_y \times fillingFactor}{numBodies \times \frac{4}{3} \times \pi}}$$

Next, the main program sets up the type of spatial data structure and forcing. Forcing refers to all the forces in the system. In the non-uniform tests attracting and repelling forces were introduced to force the particles not to be uniform. The function that initializes the repelling force takes two parameters: the strength of the repulsion and the repelling radius.

Initially, there was only one repelling force to make the system non-uniform, but as the timesteps progressed the system became more uniform. Then an attracting force was introduced instead. However, this forced all the particles to clump up in the middle of the system; even though this was non-uniform, it was not satisfying. Finally, to make this system non-uniform two forces were introduced, one that repelled and another that attracted. Both forces act from the same spot; however, the attracting force has a larger radius but is weak, while the repelling force has a small radius but is strong. This combination created a torus of particles that would be non-uniform and create multiple collisions.

Once the forcing is initialized, the type of population must be configured. In this simulation, population referred to spherical particles that move in straight lines. However, the design is flexible enough to allow for various types of particles and populations.

Next, the type of particle distribution is selected. Initially both uniform and non-uniform tests start with a uniform distribution of particles. Even though the simulation could have given the particles a non-uniform distribution, without a repelling or attracting force the particles would with time revert back to an uniform distribution.

After the initializations, the simulation executes. Using time-slicing it runs for about 2000 timesteps; in each timestep the particles move, collide with each other, and are acted on by various forces. To ensure that no particles escape the area, any particle that leaves the general boundary will wrap around and return on the other side. The simulations were run for 2000 timesteps because this is roughly how long it took for the system to reach equilibrium.

2.4 Implementation Details

The *System* class is primarily responsible for providing an easy way to evolve the system. In main, a for loop repeatedly calls *System*'s advance function to evolve the system. Figures 2.3 and 2.4 are UML sequence diagrams for simulations with grid and tree methods respectively.

System has two private attributes, an instance of *Forcing* and an instance of *Population*. *Forcinig* is responsible for resolving issues with potential collisions and applying certain forces to particles. There are two types of collisional forcing: *CollisionForcing* and *TreeCollisionForcing*, the primary difference being that the former works with gird- or hash-based structures and the latter with trees.

To initialize the *BasicPopulation* class, *Boundary* and *Output* classes had to be defined.

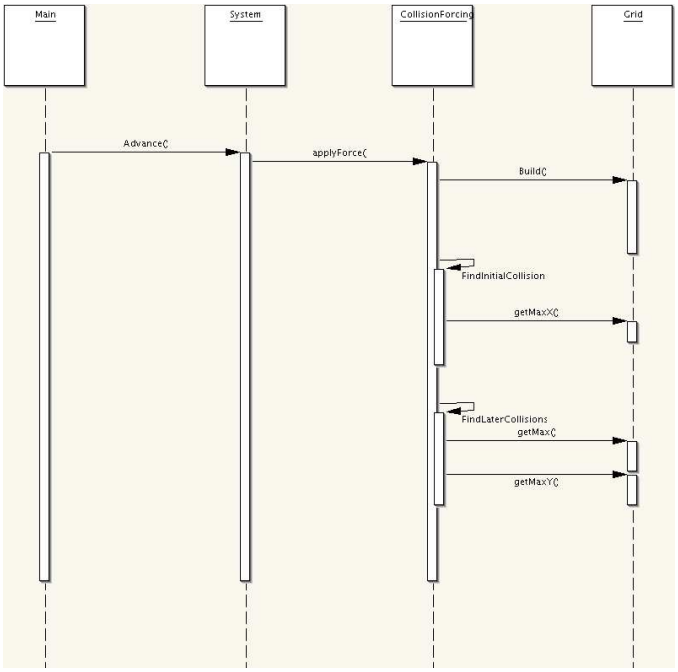


Figure 2.3: Sequence Diagram for Grid Methods

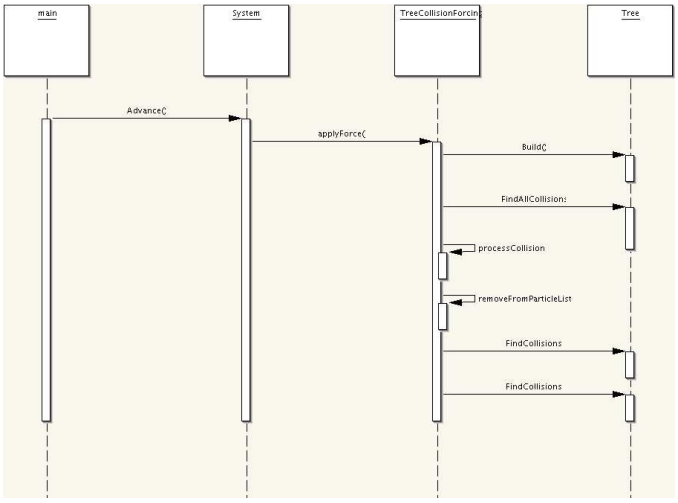


Figure 2.4: Sequence Diagram for Tree Methods

These two classes define the space where the particles are and how to display them. If a particle leaves, a method in the *Forcing* class is called by the *Boundary* sub-type. To aid the *Boundary* class, a *Coordinate* class was created.

Most of this code was made to be polymorphic and flexible. This simulation code can accommodate different population types and different distributions.

2.5 Data Structure Inter-workings

When the data structure is first declared, a few member variables are defined, but most of the structure is built by the Build method which constructs an appropriate tree or grid for the particles represented in the *Population* class. Most of the data structures create themselves by repeatedly inserting a particle one at a time until all particles are indexed.

There are two methods the framework uses for testing for collisions, RecurseForOne and RecurseForAll. The RecurseForOne method searches for all the particles within a

certain neighborhood of a particle, once a particle is determined to be roughly within a neighborhood of a requested particle, an index of both particles and the current time is passed to a method in the *CollisionHandler* class. The radius of this neighborhood is determined by the following formula:

$$radius \quad = \quad 2(\max radius_{particle}) + (3 * RMSVelocity)(TimeStep)$$

RMSVelocity is the relative velocity of nearby particle pairs; this velocity information is needed to determine how big of a search radius is needed. The faster the particles are moving, the larger the search radius. However, since time and collisions affect the particles' velocity, this value is usually recalculated after a certain number of timesteps.

When a pair of particles is passed to the *CollisionForcing*, the *CollsionHandler* performs more expensive testing to determine whether or not these particles are colliding with each other.

RecurseForAll is a more generalized method that applies RecurseForOne to all particles or can be a more optimized method.

2.6 Notes on Optimizations

To compare data structures as accurately and as quickly as possible, the code was optimized as follows. In the data structures, whenever a coordinate needed to be processed it was processed as an array of doubles rather than a vector of doubles since at the time it was believed the repeated creation and resizing of vectors was masking a call to the `new` operator. Additionally, the call to `new` would unfairly slow down certain data structures. Vectors were only used in instances where any possible loss of speed would be outweighed by the safety

and ease of use they provide.

To quickly determine if two particles were colliding, all particles were made spherical. Deciding if two particles represented as spheres have collided is regarded as computationally easier to determine than determining whether two triangles, etc. have collided. Even though the simulation could have been altered to accommodate multiple shapes, it would have taken even more time to run these experiments and would have reached roughly the same conclusions.

Moreover, these programs were all optimized by the GNU C++ compiler with the "-O7" flag, where the code is further optimized in assembly. Attention was paid to not excessively burdening any data structure with extra variables and assignments relative to the others.

Chapter 3

Types of Spatial Data Structures

3.1 Review of Literature

In the simulation literature, many authors discuss the applications of various data structures to various problems, such as Cheng and Lee [2],and Eppstein and Erickson [3]. Furthermore, there are a number of papers on parallel implementations of these data structures such as Warren and Salmon [13], and Barrett [1]. Barrett provides good background information on several popular data structures, as well as some algorithm analysis, and also details why linear quadtrees should be applied to astronomical databases.

This project primarily relied on the writings of Samet [10, 11]. In his book [11], Samet surveys various spatial data structures; he does an analysis and provides pseudo-code. Moreover, he references several books and articles regarding spatial data structures, making it easy to locate more detailed facts, such as the publication by GoodChild and Grandfield [5]. Samet provided a large amount of information on point and region methods.

This project required having a strong understanding of various data structures and to this end Samet's book was useful; to further supplement my knowledge of these data

structures an extract of Andrew Moore's PhD thesis [9] was used. The experiment also used some unconventional data structures that were popularized by Mark Lewis (specifically, the variable-size grid [8] and the PSKD-Tree [7]). So his writings were consulted.

3.2 Data Structures

3.2.1 Grids

Fixed Grids

Fixed grids are a popular collision detection method that are $O(n)$ for many systems. Fixed grids divide the system into several squares of equal area, as shown in Figure 3.1; this method is also called the cell method.

This method essentially places particles into a collection of buckets; it can be implemented as a two-dimensional array of linked lists. Whenever a particle's center is in the cell's area, then it is inserted into the corresponding linked list.

Collision detection works by checking the neighbors of a given cell for nearby particle-containing squares. If there are no squares that contain particles near it, then the next square is checked and so on; if a square does contain particles more computationally intensive algorithms are run.

For example, in Figure 3.1 a sample collision detection algorithm would first examine the upper right cell and then slowly move to the right. Assume the algorithm wants to determine if there are any potential collisions with the shaded gray particle. First, the algorithm would check all adjacent cells for any cells containing particles; in this case only the cell directly above it and to the left contain particles. Next the algorithm would check to see if there is a potential collision between the particles in shaded cell and its neighbor's

particle. If there is a potential collision, then a more intensive collision detection algorithm is applied.

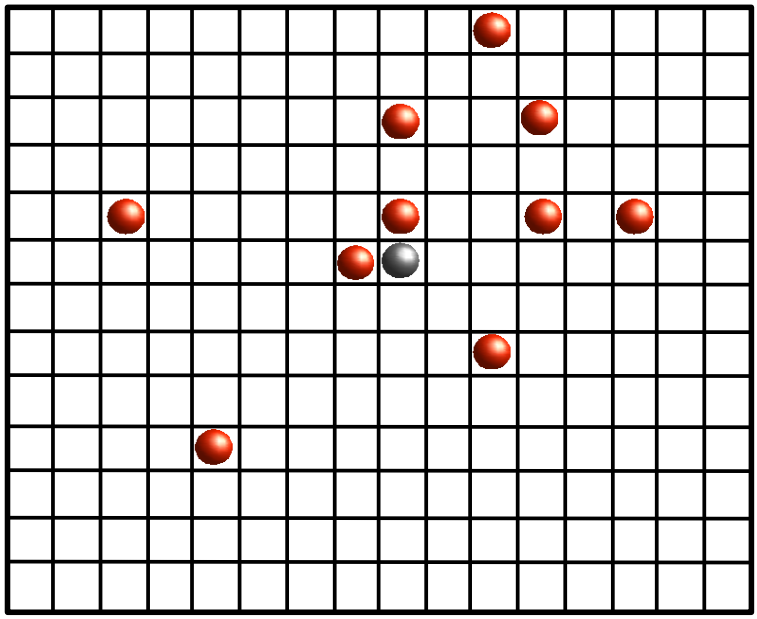


Figure 3.1: Fixed Grid

Now we will elaborate on what we mean by the next square. According to Goodchild and Grandfield [5], there are several ways to determine which is the next square in a grid. For example, an algorithm can run through all squares in a row, left to right, and then wrap around; this is called row order. This is the ordering that was chosen for this project, due to its simplicity and ease of writing. The other methods are row-prime order, Morton order, Peano-Hilbert order, Cantor-diagonal order, and spiral order; depending on the application and the users of the system, different orderings may be optimal. Some orderings may save

cache, some orderings might be more intuitive for a given system, etc.

Variable Grids

Fixed grids, however, are wasteful when the system has a non-uniform distribution of particles. There is no need to check hundreds of empty cells when there are only a few dozen cells in a highly populated area. In this situation, it is desirable to spend time traversing in detail highly populated areas and to superficially inspect those near-empty regions; variable grids attempt to provide this solution without going into the full complexity of a tree.

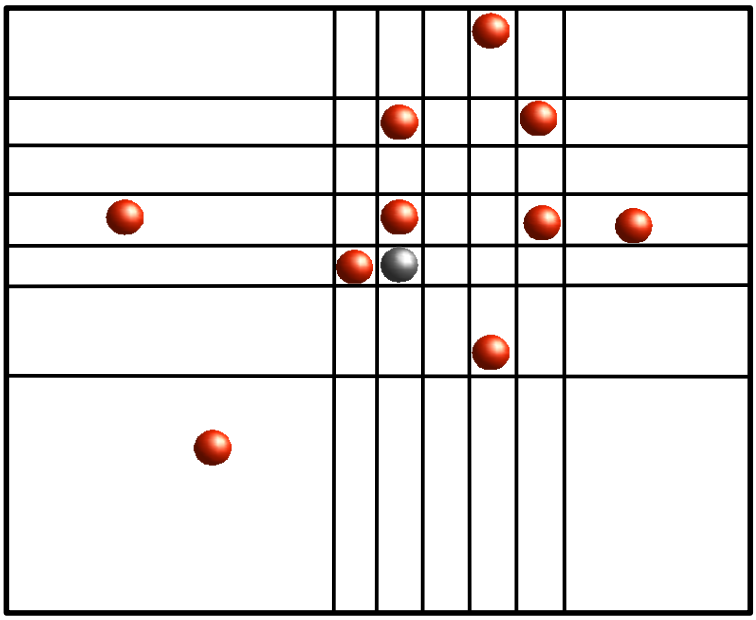


Figure 3.2: Variable Grid

Variable grids are similar to fixed grids, with some slight differences. Unlike fixed grids,

the dimensions and size of the cells are not fixed and uniform; some areas with many particles may have hundreds of cells, whereas areas with just a few particles may contain just a dozen large cells. The reasoning is that clusters of particles should have smaller squares, and vast regions with no particles should be considered just one big square. This way when the detection algorithm traverses the grid time is not wasted on small empty squares. Variable grids have a lookup and retrieval time of $O(1)$.

3.2.2 Trees

Grids look up and retrieve particles in $O(1)$ and are not hard to program, but few can adapt to systems. Those that can adapt to different distributions of particles are currently a minority. These reasons motivate some simulations to use tree structures for their collision detection and particle management.

Trees have a lengthy history and have been well-studied. They can be dynamic and adapt as the system evolves. In theory, trees deal with simulations of non-uniformly distributed particles better than non-dynamic methods, such as fixed grids.

Quadtrees

Quadtrees are trees that recursively divide the system into four similar regions. Each interior node will have 2^d descendants, where d is the dimension of the space. Varying quadtrees will have differing rules for resolution and how they handle empty space; however, all divide regions into four smaller areas. Figure 3.3 shows a 2 dimensional system that has been divided by a tree and the corresponding tree. Octrees are the 3 dimensional generalization of quadtrees; unfortunately quadtrees do not scale well in terms of memory into higher dimensions.

For example, suppose a module tries to add a given element into the tree. Assuming a

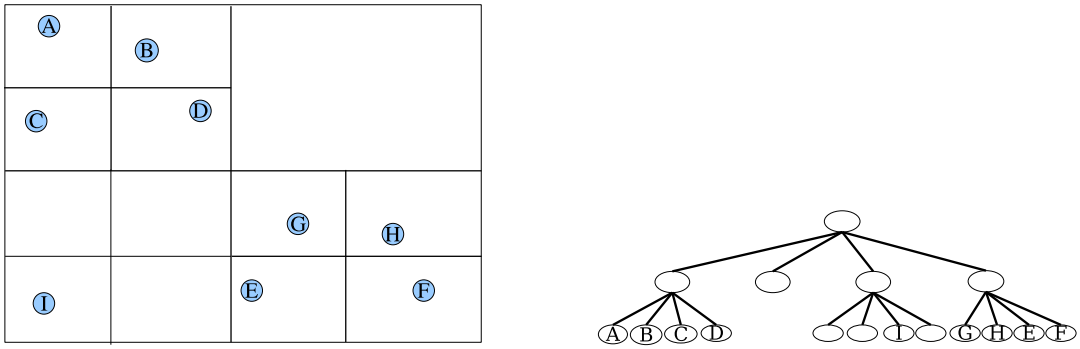


Figure 3.3: Quadtree

standard quadtree, first a function decides where, in relation to a given point an element can belong: northwest, northeast, southwest, or southeast quadrant. After a decision is made, the function determines in which of the selected quadrant's quadrants to insert the element. This process continues until the leaves of the tree are reached. The following is pseudo-code for the insert method:

```
void insert(Particle &p)
{
    Particle i = root;
    Particle parent = root;
    Direction D;
    while(i.validParticle)
    {
        D = i.findRegion(p);
        parent = i;
        i = i.getChild(D);
    }

    i.assignChild(p,D);
}
```

Since this is a tree structure, inserting and traversing the tree takes logarithmic time.

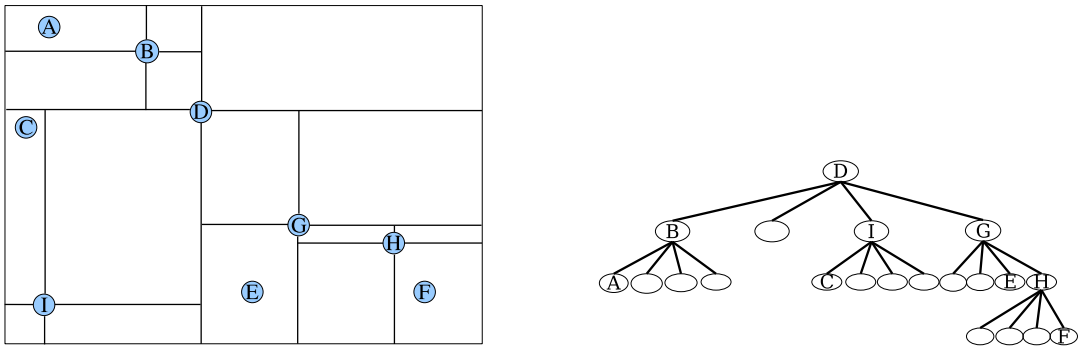


Figure 3.4: Point Quadtree

Additionally, depending how it is constructed and whether it is point or region-based, it can use less memory than grids. Unfortunately, quadtrees are significantly more complex to code and require a certain number of particles in order for it to be considered efficient.

3.2.3 Point vs. Region-Based Data Structures

Additionally, quadtrees have changed and spawned several variants since their creation. The quadtree that was just described was a region quadtree for an $n \times n$ region. For the sake of completeness and a better understanding of quadtrees it may be useful to describe a slightly different approach to quadtrees. Figure 3.4 shows a quadtree variation and the region it decomposed. Here the region is divided into rectangles where the points are the corners of the rectangles and particles can be stored throughout the tree, not just the leaves.

Notice that the shape of point trees is based on the order in which particles are entered into the tree. It is possible to inadvertently create an incredibly skewed tree if the root represents a particle with an extreme x or y value, whereas region-based structures will have the same shape regardless of the order in which particles were added.

In addition to region and point quadtrees, there are other spatial data structures such as

MX-quadtrees, PMR quadtrees, linear quadtrees, and more. Most of these data structures were made to more adequately deal with the type of data given and to process the data in a more specialized way. For example, some data structures deal better with ranges and regions rather than specific points; some data structures were geared for curvilinear data and others are more suited for area data. This project will not go into these variants for the sake of brevity, but can be the topic of future research.

Recursive Decomposition

Quadtrees operate on the principle of recursive decomposition. Recursive decomposition is the process of dividing a given system repeatedly, until a termination criterion is reached. According to Samet, et al. there are three ways quadtrees can differ:

1. Resolution
2. The type of data they can represent, i.e. homogeneous or heterogeneous
3. The division algorithm

Resolution refers to how much detail the quadtree decomposes its system down to; some quadtrees will only subdivide until they reach a region of a certain size and others will divide all the way down to a given particle. Resolution can be constant or variable.

Moreover, not all trees necessarily contain homogeneous data. Some popular quadtree designs allow for empty or non-particle regions to be represented in the tree.

Finally, different designs have different decomposition algorithms. These refer to which regions get mapped to which node in the subtree.

KD-Trees

KD-Trees are trees with k dimensions, where each level has a different key or discriminator than adjacent levels. With respect to simulations, KD-trees are binary trees that divide the system by the x or y axis. The division algorithm first divides the area by either its x or y axis, and on the next level in the tree the divided area is divided once more by the other axis. This alternation process is not necessarily true for all KD-Trees, but informally speaking all nodes in a tree could divide by a different dimension. The specifics of this data structure vary based on whether or not this is a standard or point KD-Tree. The following is pseudo-code of an insert method for a KD-Tree:

```
void insert(Particle &P)
{
    Particle i = tree[root];
    Particle.parent = root;
    int splitDim;
    double splitVal;
    while(i.validParticle)
    {
            Direction D;
            splitDim = i.getSplitDim();
            splitVal = i.getCoordinate(splitDim);
            if(splitVal > P.getCoordinate(splitDim))
                    D = LEFT;
            else
                    D = RIGHT;
            parent = i;
            i = i.getChild(D);
     }
     i.assignChild(p,D);
}
```

Like quadtrees, their operations are logarithmic and can save memory. A KD-Tree is be a binary tree which has roughly $O(\log N)$ depth. While KD-Trees are more complex

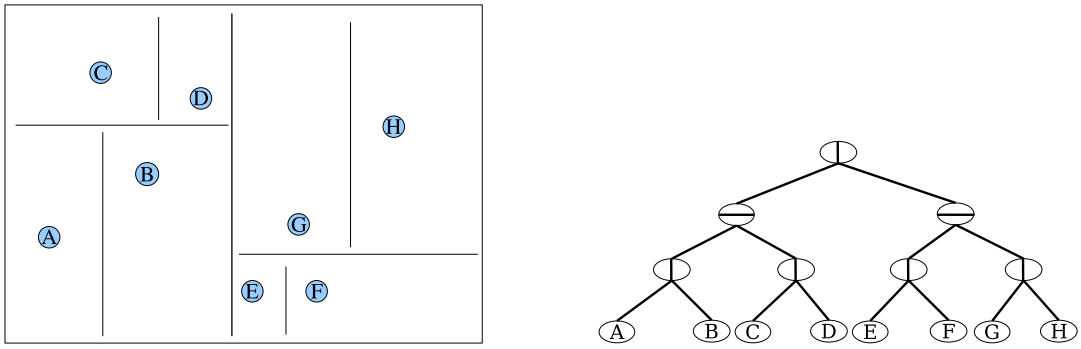


Figure 3.5: KD-Tree

than grids, they are slightly easier to code than quadtrees because of lower branching factor and less logic. Unlike region quadtrees, each node should be less expensive since each node only needs to keep track of two pointers instead of four. Furthermore, should the need to represent more than two dimensions arise, KD-Trees scale better than quadtrees.

Notice that the definition of KD-Trees does not prohibit the creation of point-based KD-Trees. The line that divides a region could be based a point.

PSKD-Trees

PSKD-Trees are a derivative of KD-Trees: they are binary trees, that instead of dividing a system strictly on spatial coordinates, divide based on spatial and velocity coordinates. According to the literature, they divide the particles based on phase space – space that is divided into location and velocity. Every particle has four variables: x, y, v_x, and v_y.

By dividing based on location and on velocity, PSKD-Trees allow the user to increase efficiency and accuracy. Sometimes it is possible for a particle to be far away from any other particle, but to be traveling exceptionally fast; if this particle is traveling fast enough, it can go through another particle in next time step. PSKD-Trees help prevent this. They

also guard against checking particles that are next to each other but moving in opposite directions.

Because of the complexity of collision detection, the programmer wants to keep the number of particles that need to be tested to as small as a number as possible. However, should there be a system where half of the particles are moving extremely quickly and the other half are moving very slowly, then the RMS Velocity will be inadequate for either group. PSKD-Trees solve this problem, by considering more variables and incorporating them into the search radius method.

Chapter 4

Results

The simulations were run for multiple filling factors and numbers of particles. The following are the results of the runs.

4.1 Uniformly Distributed Particles

In the simulations, the particles were uniformly distributed as shown in Figures 4.1, 4.2, 4.3, and 4.4. They were run for the filling factors 0.1, 0.03, 0.01, 0.003 and for number of particles 300, 1,000, 3,000 10,000, 30,000, and 100,000. Figures **??**, **??**, **??**, and **??** show the average number of times the data structure invoked the CollisionTime method to test if the particle pair was a potential collision.

It was not surprising to see that the grid methods were faster than the tree methods. While the trees could adapt better than the grids on average; the trees' startup cost was too great to overcome the grids' low startup time.What was surprising was that the variable grid in certain filling factors overcame the fixed grid.

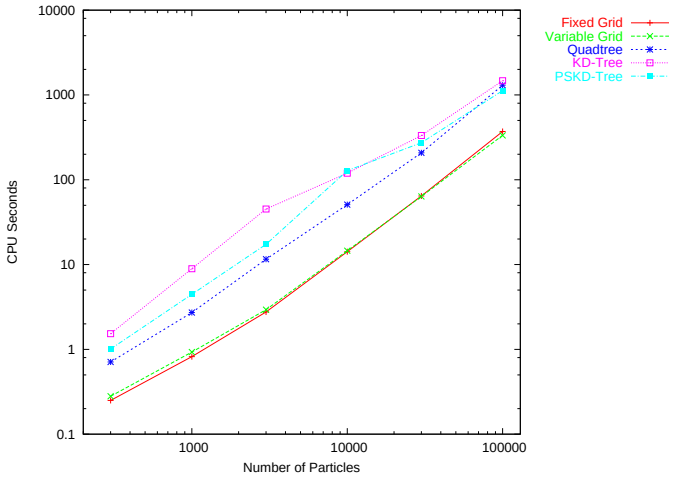


Figure 4.1: Uniform Distribution with a Filling Factor of 0.1

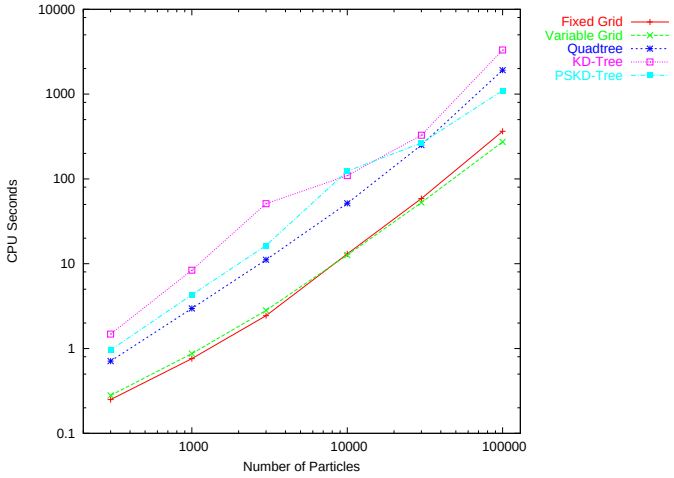


Figure 4.2: Uniform Distribution with a Filling Factor of 0.03

4.1.1 Fixed and Variable Grid

Oddly, the variable grid outperforms the fixed grid on certain filling factors and large populations, see Fig 4.4. This result is interesting; traditionally fixed grids are regarded

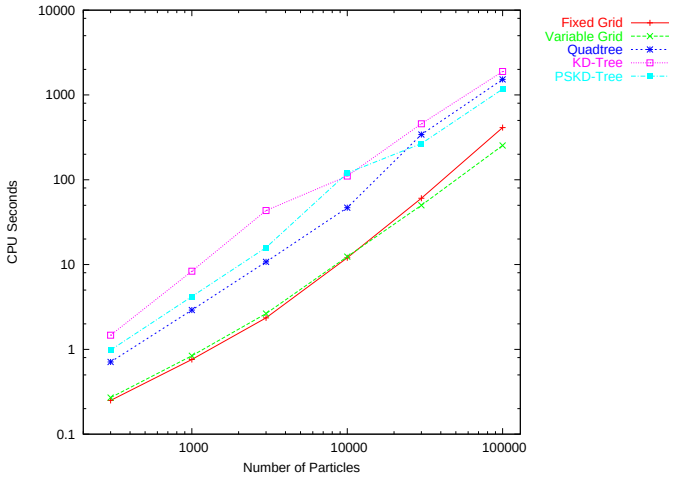


Figure 4.3: Uniform Distribution with a Filling Factor of 0.01

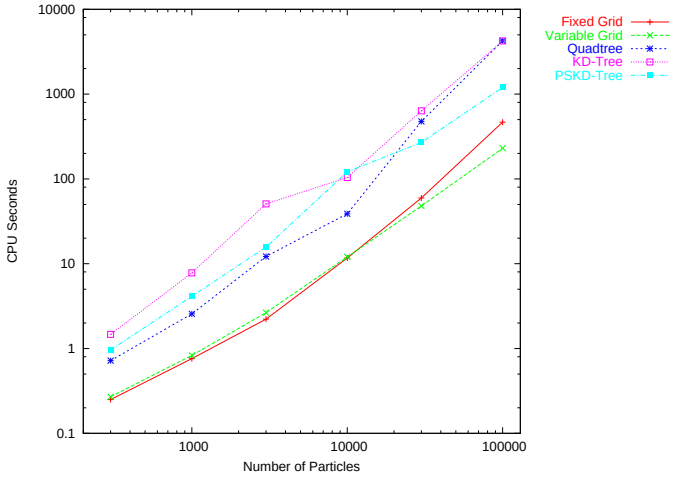


Figure 4.4: Uniform Distribution with a Filling Factor of 0.003

as the best method for an uniformly distributed system. Yet this notion does not agree with the data. Fixed and variable grid behave similarly; when there are small uniform populations in high filling factors, fixed grid outperforms variable. In uniformly distributed

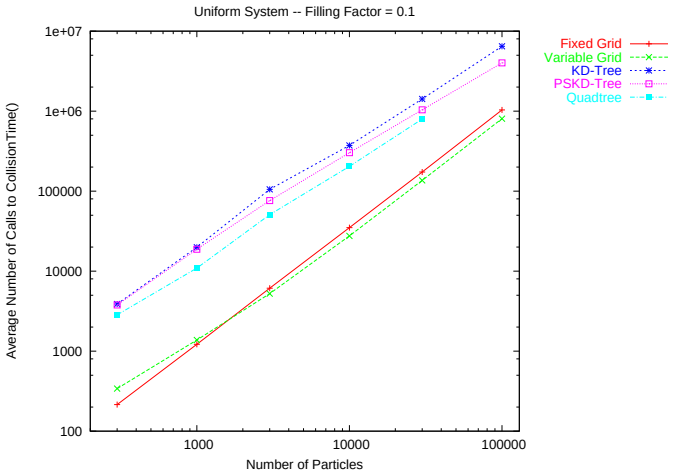


Figure 4.5: Uniform Distribution with a Filling Factor of 0.1

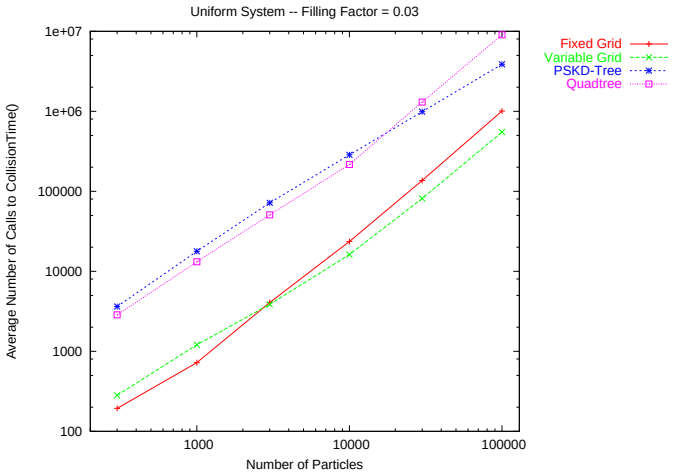


Figure 4.6: Uniform Distribution with a Filling Factor of 0.03

systems with low filling factors, the distance between the variable grid's data points and the fixed's appeared to increase and it was believed it would continue to do so as the filling factors decreased.

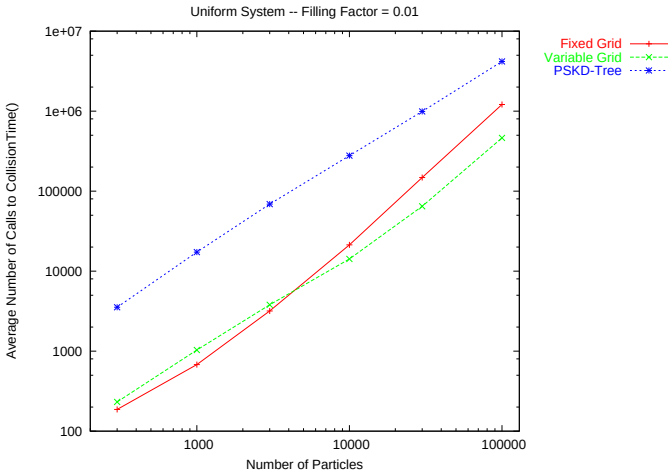


Figure 4.7: Uniform Distribution with a Filling Factor of 0.01

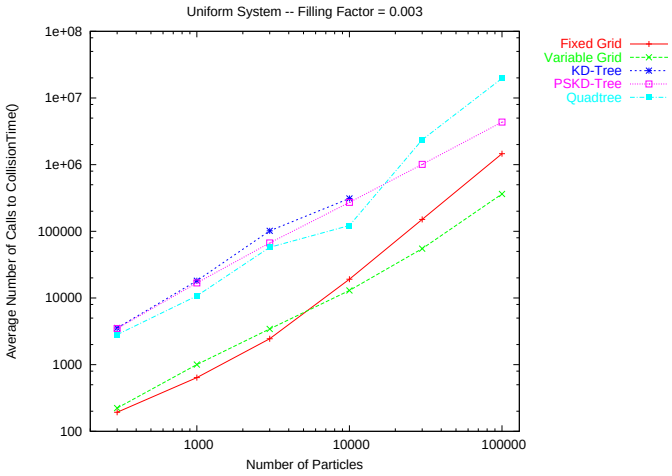


Figure 4.8: Uniform Distribution with a Filling Factor of 0.003

To test this notion more simulations with differing filling factors and 100,000 particles were run. The results are in Table 4.1.1. The results indicate that the gap between the two functions does tend to increase as filling factor decreases but it then starts to decrease. This

Filling Factor	Variable Grid	Fixed Grid	(Fixed Grid - Variable Grid)
0.5	675.19	662.40	-12.790
0.3	521.22	513.80	-7.420
0.1	333.46	370.90	37.440
0.03	273.09	363.99	90.900
0.01	253.40	412.06	158.66
0.003	229.93	466.81	236.88
0.001	233.38	466.17	232.79
0.0003	222.76	451.51	228.75
0.0001	210.44	450.59	240.15

Table 4.1: Fixed Grid's and Variable Grid's Time

inconsistency leaves us with no generalizations about the gap between the two. However, from the results it is clear that variable grid will outperform the fixed grid method after a certain filling factor.

We speculate that as filling factor increases so does the particle radius. This increases the required cell size, thus decreasing the number of cells in the fixed grid, which leaves fewer empty squares. As the filling factor decreases the size of the cells decreases, creating more cells in an empty region. Traversing these empty squares takes a speed hit, and may possibly slow down fixed grid.

In general, the grids did best in terms of speed. The only other method that came close to outperforming the grids was the KD-Tree. Its smaller dimensionality and relatively small start-up cost helped it outperform the PSKD-Tree.

4.2 Non-Uniformly Distributed Particles

In the simulations, the particles were non-uniformly distributed as shown in Figures 4.9, 4.10, 4.11, and 4.12. They were run for the filling factors 0.1, 0.03, 0.01, and 0.003 and for the numbers of particles 300, 1,000, 3,000, 10,000, and 30,000. Because of the repelling and

attracting forces, the number of collisions increased significantly faster than the uniformly distributed system; figure 4.13(a) is a plot of collisions per timestep. Where the uniformly distributed system experienced roughly a constant number of collisions, Figure 4.13(b), the number of collisions grew linearly in the non-uniformly distributed system.

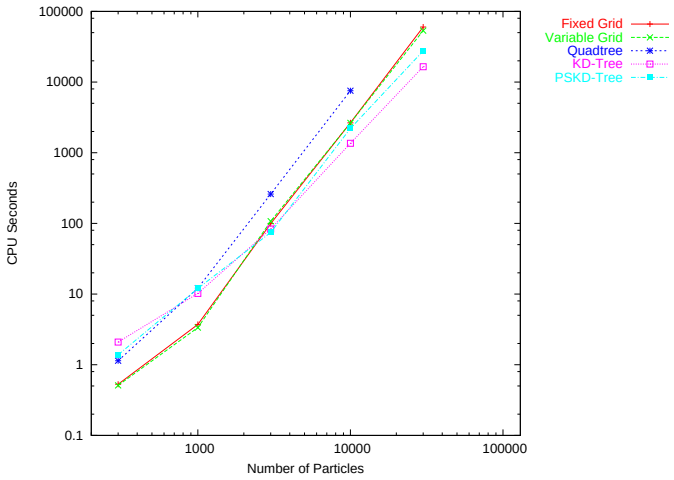


Figure 4.9: Non-Uniform Distribution with a Filling Factor of 0.1

In the graphs and charts for the uniformly distributed particle systems, the grid methods tended to be faster than the tree method, especially when the filling factor and number of particles became high. In the non-uniformly distributed particle simulations, the trees tended to dominate the grid methods, especially when the filling factor and number of particles became high. However, the variable grid again surprisingly outperformed the other methods when the low filling factors 0.01 and 0.03 were used, in one case performing 261.79 CPU seconds faster than the KD-Tree.

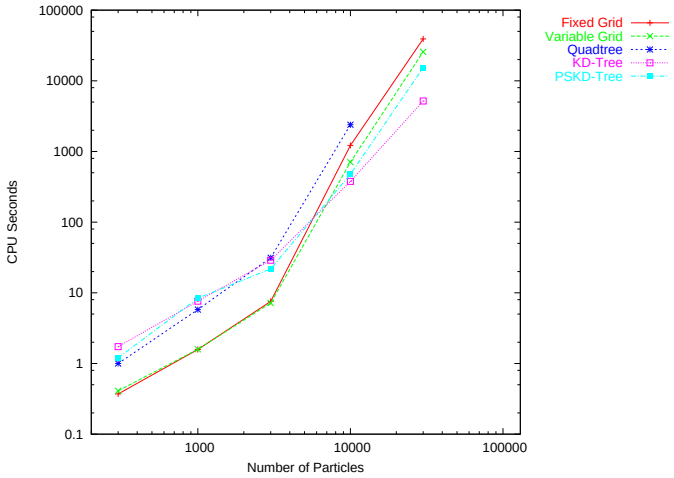


Figure 4.10: Non-Uniform Distribution with a Filling Factor of 0.03

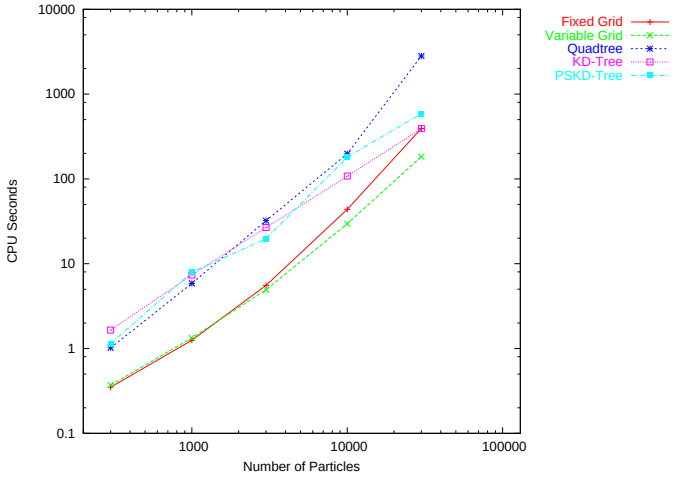


Figure 4.11: Non-Uniform Distribution with a Filling Factor of 0.01

4.2.1 Variable Grid and KD-Tree

It was odd to have any grid method do better than the tree methods. By popular opinion, grids should not outperform trees. To confirm these results, we examined the number of

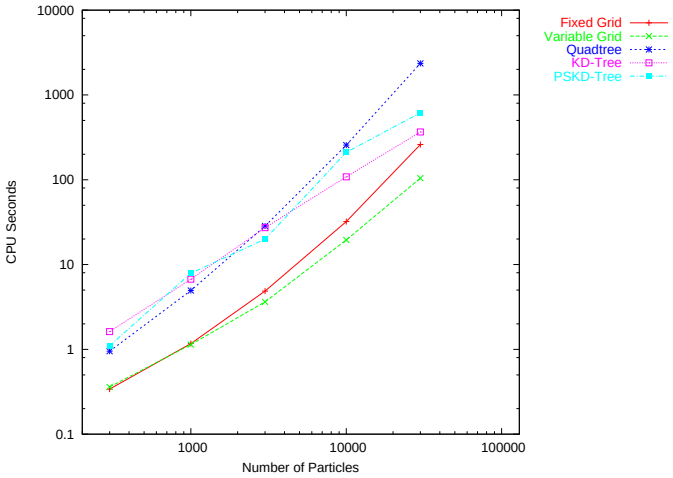
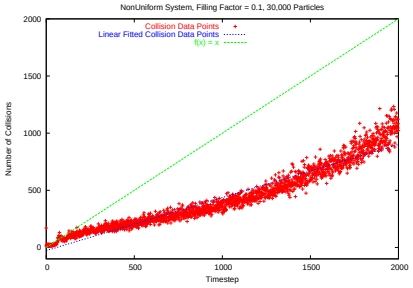
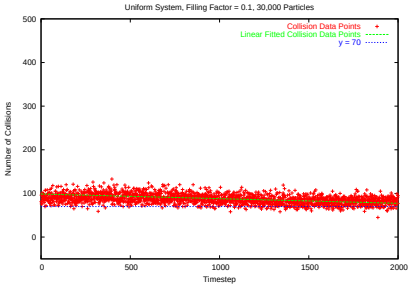


Figure 4.12: Non-Uniform Distribution with a Filling Factor of 0.003



(a) Non-Uniform Distribution with a Filling Factor of 0.1 and 30,000 particles



(b) Uniform Distribution with a Filling Factor of 0.1 and 30,000 particles

potential collisions and the number of collisions for each data structure. The two methods consistently had about the same number of collisions and potential collisions. Then multiple simulations on the same two data structures at a filling factor of 0.003 with 100,000 particles were run and the variable grid and the results were consistent.

Further testing was done and the variable grid's speed only increased further as filling

factor decreased with non-uniformity and population size were held constant. From Figure **??**, it appears that variable grid will continue to beat KD-tree for lower filling factors.

4.2.2 The Quadtree and the Non-Uniform Tests

For this section the quadtree was able to be fully tested for the population size 30,000 at filling factors 0.1 and 0.03. These two simulations tended to last the longest for all data structures, but the quadtree did not finish its simulation before the this project ended. Due to time constraints all that can be said is that the quadtree ran for over 40 hours on these final simulations, and from other graphs appeared to be slower than other methods in a non-uniform environment. Figure 4.13 and figure 4.14 may help to provide some insight as to why the quadtree performed so poorly relative to others in non-uniform tests. Regretfully other figures could not be generated due to the large amount of time of it takes to generate these figures.

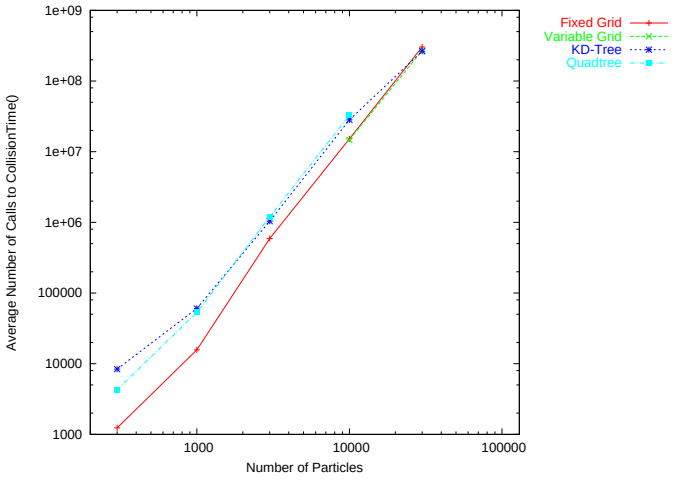


Figure 4.13: Non-Uniform Distribution with a Filling Factor of 0.1

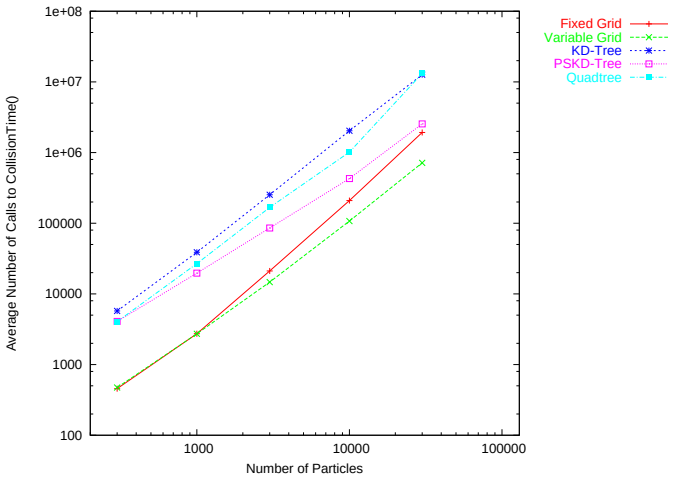


Figure 4.14: Non-Uniform Distribution with a Filling Factor of 0.01

Chapter 5

Conclusion

5.1 Final Remarks

To sum up the results, the fixed grid was the optimal method in uniformly distributed systems with relatively high filling factors. However, with lower filling factors and high population sizes in the uniform system, the variable grid outperformed the fixed grid. In a non-uniformly distributed system the KD-Tree outperformed most methods when the population size was high; when filling factors start to decrease and become lower the variable grid performed better for large population sizes.

5.2 Possible Weaknesses

Many projects usually have one weakness or assumption in them and this project was no exception. These results assumed that all data structures were equally optimized and that coding styles did not different significantly among the data structures. Naturally as the project progressed programming skill and expertise increased and the most recent

structures might have been better coded than earlier data structures. To counteract this possible source of error, revisions to old code was made periodically. Usually the changes were syntactically, but occasionally they were algorithmic.

Furthermore, it is possible that these data structures do not necessarily reflect the most optimized and efficient instances of these abstract data types. Perhaps a data structure would have performed better if it could store more particles in a given node. This opens the possibility to further research but does not take away from the accuracy of the current results. These considerations should be kept in the mind of those who may use this information. The tree methods for this research allowed for a maximum of five particles in the leaf nodes.

Additionally, the gathered data did contain some noise. Due to the way the operating system handles switching between processes, these data structures would occasionally have different times. Due to noise, this data should be interpreted as having a five percent standard deviation. In some cases, the test program was executed multiple times and the times averaged to minimize the effect of noise. Not all of the data was averaged due to time constraints. The noise was not enough significant to cause benefit or hurt a data structure enough so that it would cause it to outperform another

5.3 Further Research

5.3.1 Other Data Structures

While this work attempted to benchmark some of the more popular spatial data structures for physical simulations, there are still many spatial data structures which could be analyzed such as MX Quadtrees and pyramids.

Furthermore, while it is now known how these data structures scale with differing dis-

tributed particles and population sizes, there are no articles that mention how these data structures react to different time-step sizes.

5.3.2 Time-Step Size

This is significant because with certain applications it is more accurate to have a small time-step. For example, in the collision simulation during the non-uniform simulation runs with 100,000 particles there were timesteps with over 500,000 collisions. Clearly, in this time step there were some particles that had more than one collision.

However, if we include more collisional forces in the simulation the overhead of each timestep grows. This makes it desirable to take larger timesteps but doing so complicates our simulation further. For example, with more forces and larger timesteps we can no longer assume the particles will always travel in straight lines. However, this is tricky for spatial data structures because the number and location of collisions will affect how the system evolves and where particles are.

These are only a few ideas; there are many more possible ideas and there will be a demand for any research that could potentially speed up a simulation because in some cases it can make the difference between a hour and several hours.

Bibliography

[1] P. Barrett. Application of the Linear Quadtree to Astronomical Databases. In *Astronomical Data Analysis Software and Systems IV, ASP Conference Series*, volume 77, 1995.

[2] Siu-Wing Cheng and Jacky Kam-Hing Lee. Quadtree Decomposition, Steiner Triangulation, and Ray Shooting. Technical Report HKUST-TCSC-98-09, Hong Kong Univ. of Science & Tech., Dept. of Computer Science, Clear Water Bay, Hong Kong, 1998.

[3] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 58–67. ACM Press, 1998.

[4] M. Franquesa-Niubo and P. Brunet. Collision prediction using MKtrees. In *Proceedings of the WSCG 2004, The 12 International Conf. in Central Europe on Comp. Graphics, Visualization and Comp. Vision 2004*, volume 1, February 2004.

[5] Micheal Goodchild and Andrew Grandfield. Optimizing raster storage: An examination of four alternatives. In *Proceedings of the 6th International Symposium on Automated Cartography*, pages 400–407, October 1983.

[6] Taosong He. Fast collision detection using QuOSPO trees. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 55–62. ACM Press, 1999.

[7] Mark Lewis. Efficient collision detection optimized for long timesteps. In *Proceedings of the IASTED International Conference, Modeling and Simulation*, 2005.

[8] Mark Lewis and G. R. Stewart. A new methodology for granular flow simulations of planetary rings – collision handing. In *Proceedings of the IASTED International Conference, Modeling and Simulation*, pages 292 – 297, 2003.

[9] Andrew Moore. An introductory tutorial on KD-Trees. Technical Report Technical Report No. 209, Computer Laboratory, University of Cambridge, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1991.

[10] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[11] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[12] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Benchmarking access methods for time-evolving regional data. *Data Knowl. Eng.*, 49(3):243–286, 2004.

[13] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM Press, 1993.

Appendix A

Source Code

A.1 FixedGridCollisionHash.h

```
// FixedGridCollisionHash.h
// This class represents a standard method of creating a set of bins to
// quickly find pairs of particles that need to be tested for collisions.
// The grid size on it is a fixed grid where all bins are the same size.

#ifndef FIXED_GRID_COLLISION_HASH
#define FIXED_GRID_COLLISION_HASH

#include<vector>
#include<math.h>

class FixedGridCollisionHash {
        public:
                FixedGridCollisionHash() {
                        buildsSinceLastAdjust=10000;
                        first.resize(0);
                        next.resize(0);
                }

                template<class Population>
                void build(Population &pop) {
                        minx=pop.getx(0);
                        maxx=pop.getx(0);
                        miny=pop.gety(0);
                        maxy=pop.gety(0);
                        for(int i=1; i<pop.getNumBodies(); ++i) {
                                if(pop.getx(i)<minx) minx=pop.getx(i);
                                if(pop.getx(i)>maxx) maxx=pop.getx(i);
                                if(pop.gety(i)<miny) miny=pop.gety(i);
                                if(pop.gety(i)>maxy) maxy=pop.gety(i);
                        }
                        if(buildsSinceLastAdjust>100) {
                                adjust(pop);
                        }
                        // At this point we should have a valid minSpacing value.
                        // So we can use that to build the hash with.  Add 1 just
                        // to be safe.
                        factor=1.0/minSpacing;
                        int numx=(int)((maxx-minx)*factor)+1;
                        int numy=(int)((maxy-miny)*factor)+1;
                        first.resize(numx);
                        for(int i=0; i<numx; ++i) {
                                first[i].resize(numy);
                                for(int j=0; j<numy; ++j) first[i][j]=-1;
                        }
                        printf("Hash resized to %d by %d for %d bodies\n",numx,numy,pop.getNumBodies());
                        next.resize(pop.getNumBodies());
                        for(int i=0; i<pop.getNumBodies(); ++i) {
```

```
                        int binx=getBinX(pop,i);
                        int biny=getBinY(pop,i);
                        if(binx<0) {
                                binx=0;
                        }
                        if(binx>=(int)first.size()) {
                                binx=first.size()-1;
                        }
                        if(biny<0) {
                                biny=0;
                        }
                        if(biny>=(int)first[0].size()) {
                                biny=first[0].size()-1;
                        }
                        next[i]=first[binx][biny];
                        first[binx][biny]=i;
                }

                ++buildsSinceLastAdjust;
        }

        int getMaxX() {
                return first.size();
        }

        int getMaxY() {
                return first[0].size();
        }

        double getMinSpacing() {
                return minSpacing;
        }

        int getFirst(int x,int y) {
                return first[x][y];
        }

        int getNext(int part) {
                return next[part];
        }

        template<class Population>
        int getBinX(const Population &pop,int i) {
                return (int)((pop.getx(i)-minx)*factor);
        }

        template<class Population>
        int getBinY(Population &pop,int i) {
                return (int)((pop.gety(i)-miny)*factor);
        }
private:
        // This function assigns a safe value to minSpacing for the
        // hash to use.
        template<class Population>
        void adjust(Population &pop) {
                buildsSinceLastAdjust=0;
                if(first.size()==0) {
                        // This is the first time so build a hash in such a way as to
                        // get roughly 10 particles per hash cell.  Then use the regular
                        // algorithm to get a good size after that.
                        double area=(maxx-minx)*(maxy-miny);
                        minSpacing=sqrt(10.0*area/pop.getNumBodies());
                        printf("area=%e, minSpacing=%e %e\n",area,minSpacing,10.0*area/pop.getNumBodies());
                        build(pop);
                }
                 if(pop.getNumBodies()<100) return;
                // if(pop.getNumBodies()<1000) return;
                // Run through calculating a velocity dispersion.  I normally
                // use just RMS, but it might be safer to use something that
                // is more sensative to the higher relative velocities.
                double sum=0.0,cnt=0.0;
                while(cnt<pop.getNumBodies()/100.0) {
                        int x=lrand48()%first.size();
                        int y=lrand48()%first[0].size();
                        int f=first[x][y];
                        if(f>-1) {
```

```
                                        int n=next[f];
                                        while(n>-1) {
                                                // Increment sum.
                                                double dvx=pop.getvx(f)-pop.getvx(n);
                                                double dvy=pop.getvy(f)-pop.getvy(n);
                                                double dvz=pop.getvz(f)-pop.getvz(n);
                                                sum+=dvx*dvx+dvy*dvy+dvz*dvz;
                                                cnt+=1.0;
                                                n=next[n];
                                        }
                                        int x2,y2;
                                        n=f;
                                        while(n==f) {
                                                x2=x+(lrand48()%3)-1;
                                                y2=y+(lrand48()%3)-1;
                                                if(x2>0 && x2<(int)first.size() && y2>0 && y2<(int)first.size())
                                                        n=first[x2][y2];
                                                else n=-1;
                                        }
                                        while(n>-1) {
                                                // Increment sum.
                                                double dvx=pop.getvx(f)-pop.getvx(n);
                                                double dvy=pop.getvy(f)-pop.getvy(n);
                                                double dvz=pop.getvz(f)-pop.getvz(n);
                                                sum+=dvx*dvx+dvy*dvy+dvz*dvz;
                                                cnt+=1.0;
                                                n=next[n];
                                        }
                                }
                        }
                        minSpacing=2.0*pop.getMaxParticleRadius()+5.0*sqrt(sum/cnt)*pop.getTimeStep();
                        printf("minSpacing1=%e\n",minSpacing);
                        double area=(maxx-minx)*(maxy-miny);
                        if(area/(minSpacing*minSpacing)>maxRatio*pop.getNumBodies()) {
                                minSpacing=sqrt(area/(maxRatio*pop.getNumBodies()));
                        }
                        printf("minSpacing2=%e\n",minSpacing);
                }

                std::vector<std::vector<int> > first;
                std::vector<int> next;
                int buildsSinceLastAdjust;
                double minSpacing;
                double factor;
                double minx,maxx,miny,maxy;
                const static double maxRatio;
};

const double FixedGridCollisionHash::maxRatio=7.0;

#endif
```

A.2 VariableGridCollisionHash.h

```
// VariableGridCollisionHash.h
// This class represents a method of binning up particles to search for
// collisions that is adaptive yet still provides O(1) placement and retrievals.
// The idea is that there is a simple mapping for a very high resolution 1-D
// array in both x and y to the cells of the more course grained 2-D grid that
// is used to search for collisions.  This allows the total number of grid cells
// to stay

#ifndef VARIABLE_GRID_COLLISION_HASH
#define VARIABLE_GRID_COLLISION_HASH

#include<vector>
#include<math.h>

class VariableGridCollisionHash {
        public:
                VariableGridCollisionHash() {
                        buildsSinceLastAdjust=10000;
                        first.resize(0);
                        next.resize(0);
                }

                template<class Population>
                void build(Population &pop) {

                        minx=pop.getx(0);
                        maxx=pop.getx(0);
                        miny=pop.gety(0);
                        maxy=pop.gety(0);
                        for(int i=1; i<pop.getNumBodies(); ++i) {
                                if(pop.getx(i)<minx) minx=pop.getx(i);
                                if(pop.getx(i)>maxx) maxx=pop.getx(i);
                                if(pop.gety(i)<miny) miny=pop.gety(i);
                                if(pop.gety(i)>maxy) maxy=pop.gety(i);
                        }

                        if(buildsSinceLastAdjust>1000) {
                                adjust(pop);
                        }
                        // At this point we should have a valid minSpacing and factor value.
                        // So we can use that to build the hash with.  Add 1 just
                        // to be safe.  First resize all the vector.
                        int numx=(int)((maxx-minx)*factor)+1;
                        int numy=(int)((maxy-miny)*factor)+1;
                        xMap.resize(numx);
                        for(int i=0; i<numx; ++i) xMap[i]=0;
                        yMap.resize(numy);
                        for(int i=0; i<numy; ++i) yMap[i]=0;
                        int sy=(int)(sqrt(maxRatio*pop.getNumBodies()*(maxy-miny)/(maxx-minx)));
                        int sx=(int)(maxRatio*pop.getNumBodies()/sy);
                        first.resize(sx+1);
                        for(unsigned int i=0; i<first.size(); ++i) {
                                first[i].resize(sy+1);
                                for(unsigned int j=0; j<first[i].size(); ++j) first[i][j]=-1;
                        }
                        next.resize(pop.getNumBodies());

                        // Build mappings.
                        xBin.resize(pop.getNumBodies());
                        yBin.resize(pop.getNumBodies());
                        for(int i=0; i<pop.getNumBodies(); ++i) {
                                int binx=(int)((pop.getx(i)-minx)*factor);
                                xMap[binx]++;
                                xBin[i]=binx;
                                int biny=(int)((pop.gety(i)-miny)*factor);
                                yMap[biny]++;
                                yBin[i]=biny;
                        }

                        int pTot=pop.getNumBodies()/sx;
                        int pCnt=0;
                        int binCnt=0;
                        for(int i=0; i<numx; ++i) {
                                pCnt+=xMap[i];
```

```
                        xMap[i]=binCnt;
                        if(pCnt>=pTot) {
                                binCnt++;
                                pCnt=0;
                        }
                }
                pTot=pop.getNumBodies()/sy;
                pCnt=0;
                binCnt=0;
                for(int i=0; i<numy; ++i) {
                        pCnt+=yMap[i];
                        yMap[i]=binCnt;
                        if(pCnt>=pTot) {
                                binCnt++;
                                pCnt=0;
                        }
                }

                // Now build the actual hash with the variable sizes given by the
                // maps.
                for(int i=0; i<pop.getNumBodies(); ++i) {
                        int binx=xBin[i];
                        int biny=yBin[i];
                        next[i]=first[xMap[binx]][yMap[biny]];
                        first[xMap[binx]][yMap[biny]]=i;
                }

                ++buildsSinceLastAdjust;
        }

        int getMaxX() {
                return first.size();
        }

        int getMaxY() {
                return first[0].size();
        }

        double getMinSpacing() {
                return minSpacing;
        }

        int getFirst(int x,int y) {
                return first[x][y];
        }

        int getNext(int part) {
                return next[part];
        }

        template<class Population>
        int getBinX(const Population &pop,int i) {
                int binx=(int)((pop.getx(i)-minx)*factor);
                return xMap[binx];
        }

        template<class Population>
        int getBinY(Population &pop,int i) {
                int biny=(int)((pop.gety(i)-miny)*factor);
                return yMap[biny];
        }
private:
        // This function assigns a safe value to minSpacing for the
        // hash to use.
        template<class Population>
        void adjust(Population &pop) {
                buildsSinceLastAdjust=0;
                if(first.size()==0) {
                        // This is the first time so build a hash in such a way as to
                        // get roughly 10 particles per hash cell.  Then use the regular
                        // algorithm to get a good size after that.
                        double area=(maxx-minx)*(maxy-miny);
                        minSpacing=sqrt(10.0*area/pop.getNumBodies());
                        factor=1.0/minSpacing;
                        build(pop);
                }
```

```
                        // Run through calculating a velocity dispersion.  I normally
                        // use just RMS, but it might be safer to use something that
                        // is more sensative to the higher relative velocities.
                        double sum=0.0,cnt=0.0;
                        while(cnt<pop.getNumBodies()/100.0) {
                                int x=lrand48()%first.size();
                                int y=lrand48()%first[0].size();
                                int f=first[x][y];
                                if(f>-1) {
                                        int n=next[f];
                                        while(n>-1) {
                                                // Increment sum.
                                                double dvx=pop.getvx(f)-pop.getvx(n);
                                                double dvy=pop.getvy(f)-pop.getvy(n);
                                                double dvz=pop.getvz(f)-pop.getvz(n);
                                                sum+=dvx*dvx+dvy*dvy+dvz*dvz;
                                                cnt+=1.0;
                                                n=next[n];
                                        }
                                        int x2,y2;
                                        n=f;
                                        while(n==f) {
                                                x2=x+(lrand48()%3)-1;
                                                y2=y+(lrand48()%3)-1;
                                                if(x2>0 && x2<(int)first.size() && y2>0 && y2<(int)first.size())
                                                        n=first[x2][y2];
                                                else n=-1;
                                        }
                                        while(n>-1) {
                                                // Increment sum.
                                                double dvx=pop.getvx(f)-pop.getvx(n);
                                                double dvy=pop.getvy(f)-pop.getvy(n);
                                                double dvz=pop.getvz(f)-pop.getvz(n);
                                                sum+=dvx*dvx+dvy*dvy+dvz*dvz;
                                                cnt+=1.0;
                                                n=next[n];
                                        }
                                }
                        }
                        minSpacing=2.0*pop.getMaxParticleRadius()+3.0*sqrt(sum/cnt)*pop.getTimeStep();
                        factor=1.0/minSpacing;
                }

                std::vector<int> xMap;
                std::vector<int> yMap;
                std::vector<int> xBin;
                std::vector<int> yBin;
                std::vector<std::vector<int> > first;
                std::vector<int> next;
                int buildsSinceLastAdjust;
                double minSpacing;
                double factor;
                double minx,maxx,miny,maxy;
                const static double maxRatio;
};

const double VariableGridCollisionHash::maxRatio=7.0;

#endif
```

A.3 KDTree.h

```
#include <vector>

#define MAX_NUM 5
#define DIM 3

struct KDTreeNode {
        int particle[MAX_NUM];
        int numParts;
        int splitDim;
        double splitValue;
        int left,right;
};

struct KDBounds {
        double min[DIM],max[DIM];
};

class KDTree {
        public:
                template<class Population>
                void build(Population &pop) {
                        pool.resize(pop.getNumBodies()*2);
                        root=0;
                        pool[root].particle[0]=0;
                        pool[root].numParts=1;
                        firstFree=1;
                        for(int i=0; i<DIM; ++i) {
                                totalBounds.min[i]=1e100;
                                totalBounds.max[i]=-1e100;
                        }
                        for(int i=1; i<pop.getNumBodies(); ++i) {
                                add(pop,i);
                                for(int j=0; j<DIM; ++j) {
                                        totalBounds.min[j]<?=pop.get(i,j);
                                        totalBounds.max[j]>?=pop.get(i,j);
                                }
                        }
                }

                template<class Population,class CollisionManager>
                void findAllCollisions(Population &pop,CollisionManager &cm) {
                        KDBounds b1,b2;
                        for(int i=0; i<DIM; ++i) {
                                b1.min[i]=totalBounds.min[i];
                                b1.max[i]=totalBounds.max[i];
                                b2.min[i]=totalBounds.min[i];
                                b2.max[i]=totalBounds.max[i];
                        }
                        cnt=0.0;
                        searchRadius=0.0;
                        setSearchRadius(pop,root);
                        searchRadius=2.0*pop.getMaxParticleRadius()+3.0*sqrt(searchRadius/cnt);
                        recurseForAll(pop,cm,root,b1,root,b2);
                }

                template<class Population,class CollisionManager>
                void findCollisions(Population &pop,CollisionManager &cm,int p,int ignore) {
                        KDBounds b;
                        for(int i=0; i<DIM; ++i) {
                                b.min[i]=totalBounds.min[i];
                                b.max[i]=totalBounds.max[i];
                        }
                        recurseForOne(pop,cm,p,ignore,b,root);
                }
        private:
                template<class Population>
                void add(Population &pop,int p) {
                        int node=root;
                        while(pool[node].numParts==0) {
                                double val=pop.get(p,pool[node].splitDim);
                                if(val<pool[node].splitValue) {
                                        node=pool[node].left;
                                } else {
                                        node=pool[node].right;
```

```
                }
        }
        if(pool[node].numParts>=MAX_NUM) {
                double min[DIM];
                double max[DIM];
                for(int i=0; i<DIM; ++i) {
                        min[i]=pop.get(p,i);
                        max[i]=min[i];
                }
                for(int i=0; i<MAX_NUM; ++i) {
                        for(int j=0; j<DIM; ++j) {
                                double val=pop.get(pool[node].particle[i],j);
                                if(val<min[j]) min[j]=val;
                                if(val>max[j]) max[j]=val;
                        }
                }
                int sd=0;
                double sv;
                for(int i=1; i<DIM; ++i) {
                        if(max[i]-min[i]>max[sd]-min[sd]) sd=i;
                }
                sv=0.5*(max[sd]+min[sd]);
                pool[node].splitDim=sd;
                pool[node].splitValue=sv;
                pool[node].left=firstFree;
                pool[node].right=firstFree+1;
                pool[firstFree].numParts=0;
                pool[firstFree+1].numParts=0;
                for(int i=0; i<MAX_NUM; ++i) {
                        double val=pop.get(pool[node].particle[i],sd);
                        int dest=firstFree;
                        if(val>sv) dest=firstFree+1;
                        pool[dest].particle[pool[dest].numParts]=
                                pool[node].particle[i];
                        pool[dest].numParts++;
                }
                pool[node].numParts=0;
                double val=pop.get(p,sd);
                node=firstFree;
                if(val>sv) node=firstFree+1;
                firstFree+=2;
        }
        pool[node].particle[pool[node].numParts]=p;
        pool[node].numParts++;
}

template<class Population,class CollisionManager>
void recurseForAll(Population &pop,CollisionManager &cm,int node1,KDBounds &b1,int node2,KDBounds &b2) {
        if(pool[node1].numParts>0 && pool[node2].numParts>0) {
                for(int i=0; i<pool[node1].numParts; ++i) {
                        int j;
                        if(node1==node2) j=i+1;
                                else j=0;
                        for(; j<pool[node2].numParts; ++j) {
                                double t=pop.collisionTime(pool[node1].particle[i],pool[node2].particle[j]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(pool[node1].particle[i],pool[node2].particle[j],t);
                                }
                        }
                }
        } else if(pool[node1].numParts>0) {
                recurseForAll(pop,cm,node2,b2,node1,b1);
        } else if(node1==node2) {
                // This is a special case that prevents me from
                // doing extra work. In this situation I know they
                // all overlap and I only need to recurse in 3 different
                // combination, not 4.
                int sd=pool[node1].splitDim;
                double sv=pool[node1].splitValue;
                double maxTmp=b1.max[sd];
                double minTmp=b1.min[sd];

                b1.max[sd]=b2.max[sd]=sv;
                recurseForAll(pop,cm,pool[node1].left,b1,pool[node2].left,b2);
                b1.max[sd]=b2.max[sd]=maxTmp;
                b1.min[sd]=b2.min[sd]=sv;
```

```
                recurseForAll(pop,cm,pool[node1].right,b1,pool[node2].right,b2);
                b1.min[sd]=minTmp;
                b1.max[sd]=sv;
                recurseForAll(pop,cm,pool[node1].left,b1,pool[node2].right,b2);
                b1.max[sd]=maxTmp;
                b2.min[sd]=minTmp;
        } else {
                int sd=pool[node1].splitDim;
                double sv=pool[node1].splitValue;
                double tmp;
                if(b2.min[sd]-searchRadius<sv) {
                        tmp=b1.max[sd];
                        b1.max[sd]=sv;
                        recurseForAll(pop,cm,node2,b2,pool[node1].left,b1);
                        b1.max[sd]=tmp;
                }
                if(b2.max[sd]+searchRadius>sv) {
                        tmp=b1.min[sd];
                        b1.min[sd]=sv;
                        recurseForAll(pop,cm,node2,b2,pool[node1].right,b1);
                        b1.min[sd]=tmp;
                }
        }
}

template<class Population,class CollisionManager>
void recurseForOne(Population &pop,CollisionManager &cm,int p,int ignore,KDBounds &b,int node) {
        if(pool[node].numParts>0) {
                for(int i=0; i<pool[node].numParts; ++i) {
                        if(pool[node].particle[i]!=p && pool[node].particle[i]!=ignore) {
                                double t=pop.collisionTime(p,pool[node].particle[i]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(p,pool[node].particle[i],t);
                                }
                        }
                }
        } else {
                int sd=pool[node].splitDim;
                double sv=pool[node].splitValue;
                double val=pop.get(p,sd);
                if(val-searchRadius<sv) {
                        double tmp=b.max[sd];
                        b.max[sd]=sv;
                        recurseForOne(pop,cm,p,ignore,b,pool[node].left);
                        b.max[sd]=tmp;
                }
                if(val+searchRadius>sv){
                        double tmp=b.min[sd];
                        b.min[sd]=sv;
                        recurseForOne(pop,cm,p,ignore,b,pool[node].right);
                        b.min[sd]=tmp;
                }
        }
}

template<class Population>
void setSearchRadius(Population &pop,int node) {
        if(pool[node].numParts>0) {
                for(int i=0; i<pool[node].numParts; ++i) {
                        for(int j=i+1; j<pool[node].numParts; ++j) {
                                double dx=pop.getvx(i)-pop.getvx(j);
                                double dy=pop.getvy(i)-pop.getvy(j);
                                double dz=pop.getvz(i)-pop.getvz(j);
                                searchRadius+=dx*dx+dy*dy+dz*dz;
                                cnt+=1.0;
                        }
                }
        } else {
                setSearchRadius(pop,pool[node].left);
                setSearchRadius(pop,pool[node].right);
        }
}

template<class Population>
void printTree(Population &pop,int node) {
        if(pool[node].numParts==0) {
```

```
                        printf("Node %d %d %e\n",node,pool[node].splitDim,pool[node].splitValue);
                        printf("left\n");
                        printTree(pop,pool[node].left);
                        printf("right\n");
                        printTree(pop,pool[node].right);
                        printf("done with %d\n",node);
                } else {
                        printf("Particles in %d\n",node);
                        for(int i=0; i<pool[node].numParts; ++i) {
                                int p=pool[node].particle[i];
                                printf("%d %e %e %e %e %e\n",p,pop.get(p,0),pop.get(p,1)
                                        ,pop.get(p,2),pop.get(p,3),pop.get(p,4),pop.get(p,5));
                        }
                        printf("Done with %d\n",node);
                }
        }

        std::vector<KDTreeNode> pool;
        int firstFree;
        int root;
        KDBounds totalBounds;
        double searchRadius;
        double cnt;
};
```

A.4 PSKDTree.h

```
/**
 * PSKDTree.h
 * This file defines a templated phase space KD-tree that can be used to
 * find potential collisions between particles in a population.
**/

#include <vector>

#define MAX_NUM 5
#define DIM 3

struct TreeNode {
        int particle[MAX_NUM];
        int numParts;
        int splitDim;
        double splitValue;
        int left,right;
};

struct Bounds {
        double min[2*DIM],max[2*DIM];
};

class PSKDTree {
        public:
                template<class Population>
                void build(Population &pop) {
                        pool.resize(pop.getNumBodies()*2);
                        root=0;
                        pool[root].particle[0]=0;
                        pool[root].numParts=1;
                        firstFree=1;
                        for(int i=0; i<2*DIM; ++i) {
                                totalBounds.min[i]=1e100;
                                totalBounds.max[i]=-1e100;
                        }
                        for(int i=1; i<pop.getNumBodies(); ++i) {
                                add(pop,i);
                                for(int j=0; j<2*DIM; ++j) {
                                        totalBounds.min[j]<?=pop.get(i,j);
                                        totalBounds.max[j]>?=pop.get(i,j);
                                }
                        }
                }

                template<class Population,class CollisionManager>
                void findAllCollisions(Population &pop,CollisionManager &cm) {
                        Bounds b1,b2;
                        for(int i=0; i<2*DIM; ++i) {
                                b1.min[i]=totalBounds.min[i];
                                b1.max[i]=totalBounds.max[i];
                                b2.min[i]=totalBounds.min[i];
                                b2.max[i]=totalBounds.max[i];
                        }
                        recurseForAll(pop,cm,root,b1,root,b2);
                }

                template<class Population,class CollisionManager>
                void findCollisions(Population &pop,CollisionManager &cm,int p,int ignore) {
                        Bounds b;
                        for(int i=0; i<2*DIM; ++i) {
                                b.min[i]=totalBounds.min[i];
                                b.max[i]=totalBounds.max[i];
                        }
                        recurseForOne(pop,cm,p,ignore,b,root);
                }
        private:
                template<class Population>
                void add(Population &pop,int p) {
                        int node=root;
                        while(pool[node].numParts==0) {
                                double val=pop.get(p,pool[node].splitDim);
                                if(val<pool[node].splitValue) {
                                        node=pool[node].left;
```

```
                } else {
                        node=pool[node].right;
                }
        }
        if(pool[node].numParts>=MAX_NUM) {
                double min[2*DIM];
                double max[2*DIM];
                for(int i=0; i<2*DIM; ++i) {
                        min[i]=pop.get(p,i);
                        max[i]=min[i];
                }
                for(int i=0; i<MAX_NUM; ++i) {
                        for(int j=0; j<2*DIM; ++j) {
                                double val=pop.get(pool[node].particle[i],j);
                                if(val<min[j]) min[j]=val;
                                if(val>max[j]) max[j]=val;
                        }
                }
                int sd=0;
                double sv;
                for(int i=1; i<2*DIM; ++i) {
                        if(max[i]-min[i]>max[sd]-min[sd]) sd=i;
                }
                sv=0.5*(max[sd]+min[sd]);
                pool[node].splitDim=sd;
                pool[node].splitValue=sv;
                pool[node].left=firstFree;
                pool[node].right=firstFree+1;
                pool[firstFree].numParts=0;
                pool[firstFree+1].numParts=0;
                for(int i=0; i<MAX_NUM; ++i) {
                        double val=pop.get(pool[node].particle[i],sd);
                        int dest=firstFree;
                        if(val>sv) dest=firstFree+1;
                        pool[dest].particle[pool[dest].numParts]=
                                pool[node].particle[i];
                        pool[dest].numParts++;
                }
                pool[node].numParts=0;
                double val=pop.get(p,sd);
                node=firstFree;
                if(val>sv) node=firstFree+1;
                firstFree+=2;
        }
        pool[node].particle[pool[node].numParts]=p;
        pool[node].numParts++;
}

template<class Population,class CollisionManager>
void recurseForAll(Population &pop,CollisionManager &cm,int node1,Bounds &b1,int node2,Bounds &b2) {
        if(pool[node1].numParts>0 && pool[node2].numParts>0) {
                for(int i=0; i<pool[node1].numParts; ++i) {
                        int j;
                        if(node1==node2) j=i+1;
                                else j=0;
                        for(; j<pool[node2].numParts; ++j) {
                                double t=pop.collisionTime(pool[node1].particle[i],pool[node2].particle[j]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(pool[node1].particle[i],pool[node2].particle[j],t);
                                }
                        }
                }
        } else if(pool[node1].numParts>0) {
                recurseForAll(pop,cm,node2,b2,node1,b1);
        } else if(node1==node2) {
                // This is a special case that prevents me from
                // doing extra work. In this situation I know they
                // all overlap and I only need to recurse in 3 different
                // combination, not 4.
                int sd=pool[node1].splitDim;
                double sv=pool[node1].splitValue;
                double maxTmp=b1.max[sd];
                double minTmp=b1.min[sd];

                b1.max[sd]=b2.max[sd]=sv;
                recurseForAll(pop,cm,pool[node1].left,b1,pool[node2].left,b2);
```

```
                b1.max[sd]=b2.max[sd]=maxTmp;
                b1.min[sd]=b2.min[sd]=sv;
                recurseForAll(pop,cm,pool[node1].right,b1,pool[node2].right,b2);
                b1.min[sd]=minTmp;
                b1.max[sd]=sv;
                recurseForAll(pop,cm,pool[node1].left,b1,pool[node2].right,b2);
                b1.max[sd]=maxTmp;
                b2.min[sd]=minTmp;
        } else {
                int sd=pool[node1].splitDim;
                double sv=pool[node1].splitValue;
                double dt=pop.getTimeStep();
                double maxRad=pop.getMaxParticleRadius();
                int dim=(sd<DIM)?sd:sd-DIM;
                double tmp=b1.max[sd];
                b1.max[sd]=sv;
                if(checkOverlap(b1,b2,dt,maxRad,dim)) recurseForAll(pop,cm,node2,b2,pool[node1].left,b1);
                b1.max[sd]=tmp;
                tmp=b1.min[sd];
                b1.min[sd]=sv;
                if(checkOverlap(b1,b2,dt,maxRad,dim)) recurseForAll(pop,cm,node2,b2,pool[node1].right,b1);
                b1.min[sd]=tmp;
        }
}

template<class Population,class CollisionManager>
void recurseForOne(Population &pop,CollisionManager &cm,int p,int ignore,Bounds &b,int node) {
        if(pool[node].numParts>0) {
                for(int i=0; i<pool[node].numParts; ++i) {
                        if(pool[node].particle[i]!=p && pool[node].particle[i]!=ignore) {
                                double t=pop.collisionTime(p,pool[node].particle[i]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(p,pool[node].particle[i],t);
                                }
                        }
                }
        } else {
                int sd=pool[node].splitDim;
                double sv=pool[node].splitValue;
                bool below=false;
                bool above=false;
                double val=pop.get(p,sd);
                double dt=pop.getTimeStep();
                double maxRad=pop.getMaxParticleRadius();
                if(sd<DIM) {
                        if(val<sv+2.0*maxRad) {
                                below=true;
                                if(val+pop.get(p,sd+DIM)*dt>sv+dt*b.min[sd+DIM]-2.0*maxRad) above=true;
                        }
                        if(val>sv-2.0*maxRad){
                                above=true;
                                if(val+pop.get(p,sd+DIM)*dt<sv+dt*b.max[sd+DIM]+2.0*maxRad) below=true;
                        }
                } else {
                        if(val<sv+2.0*maxRad/dt) {
                                below=true;
                                if(pop.get(p,sd-DIM)>=b.min[sd-DIM]-2.0*maxRad) above=true;
                        }
                        if(val>sv-2.0*maxRad/dt) {
                                above=true;
                                if(pop.get(p,sd-DIM)<=b.max[sd-DIM]+2.0*maxRad) below=true;
                        }
                }
                if(below) {
                        double tmp=b.max[sd];
                        b.max[sd]=sv;
                        recurseForOne(pop,cm,p,ignore,b,pool[node].left);
                        b.max[sd]=tmp;
                }
                if(above) {
                        double tmp=b.min[sd];
                        b.min[sd]=sv;
                        recurseForOne(pop,cm,p,ignore,b,pool[node].right);
                        b.min[sd]=tmp;
                }
        }
```

```
	}

	/**
	 * Determines if particles in two different bounds regions could possibly
	 * overlap.
	 */
	bool checkOverlap(Bounds &b1,Bounds &b2,double dt,double maxRad,int i) {
		return !(((b1.max[i]<b2.min[i]-2.0*maxRad) &&
			(b1.max[i]+dt*b1.max[i+DIM]<b2.min[i]+dt*b2.min[i+DIM]-2.0*maxRad)) ||
			((b1.min[i]>b2.max[i]+2.0*maxRad) &&
			(b1.min[i]+dt*b1.min[i+DIM]>b2.max[i]+dt*b2.max[i+DIM]+2.0*maxRad)));
	}

	template<class Population>
	bool checkOverlap(Bounds &b,Population &pop,int p,double dt,double maxRad) {
		bool ret=true;
		for(int i=0; ret && i<DIM; ++i) {
			if(((b.min[i]>pop.get(p,i)+2.0*maxRad) &&
				(b.min[i]+dt*b.min[i+DIM]>pop.get(p,i)+dt*pop.get(p,i+DIM)+2.0*maxRad)) ||
				((b.max[i]<pop.get(p,i)-2.0*maxRad) &&
				(b.max[i]+dt*b.max[i+DIM]<pop.get(p,i)+dt*pop.get(p,i+DIM)-2.0*maxRad))) {
					ret=false;
			}
		}
		return ret;
	}

	template<class Population>
	void printTree(Population &pop,int node) {
		if(pool[node].numParts==0) {
			printf("Node %d %d %e\n",node,pool[node].splitDim,pool[node].splitValue);
			printf("left\n");
			printTree(pop,pool[node].left);
			printf("right\n");
			printTree(pop,pool[node].right);
			printf("done with %d\n",node);
		} else {
			printf("Particles in %d\n",node);
			for(int i=0; i<pool[node].numParts; ++i) {
				int p=pool[node].particle[i];
				printf("%d %e %e %e %e %e %e\n",p,pop.get(p,0),pop.get(p,1)
					,pop.get(p,2),pop.get(p,3),pop.get(p,4),pop.get(p,5));
			}
			printf("Done with %d\n",node);
		}
	}

	std::vector<TreeNode> pool;
	int firstFree;
	int root;
	Bounds totalBounds;
};
```

A.5 QuadTree.h

```
// QuadTree.h

#include <vector>

#define MAX_NUM 5

struct QuadTreeNode {
        int particle[MAX_NUM];
        int numParts;
        double cx,cy;
        int child[4];
        double size;
};

class QuadTree {
        public:
                template<class Population>
                void build(Population &pop) {
                        pool.resize(pop.getNumBodies()*2);
                        root=0;
                        pool[root].particle[0]=0;
                        pool[root].numParts=1;
                        pool[root].child[0]=-1;
                        pool[root].cx=(pop.getMaxx()+pop.getMinx())*0.5;
                        pool[root].cy=(pop.getMaxy()+pop.getMiny())*0.5;
                        pool[root].size=max(pop.getMaxx()-pop.getMinx(),pop.getMaxy()-pop.getMiny());
                        firstFree=1;
                        for(int i=1; i<pop.getNumBodies(); ++i) {
                                add(pop,i);
                        }
                }

                template<class Population,class CollisionManager>
                void findAllCollisions(Population &pop,CollisionManager &cm) {
                        cnt=0.0;
                        searchRadius=0.0;
                        setSearchRadius(pop,root);
                        searchRadius=2.0*pop.getMaxParticleRadius()+3.0*sqrt(searchRadius/cnt);
                        recurseForAll(pop,cm,root,root);
                }

                template<class Population,class CollisionManager>
                void findCollisions(Population &pop,CollisionManager &cm,int p,int ignore) {
                        recurseForOne(pop,cm,p,ignore,root);
                }
        private:
                int child(double x,double y,double cx,double cy) {
                        return ((x<cx)?0:1)|((y<cy)?0:2);
                }

                template<class Population>
                void add(Population &pop,int p) {
                        double x=pop.getx(p);
                        double y=pop.gety(p);
                        addRecur(pop,p,x,y,root);
                }

                template<class Population>
                void addRecur(Population &pop,int p,double x,double y,int node) {
                        if(pool[node].child[0]>-1) {
                                addRecur(pop,p,x,y,pool[node].child[child(x,y,pool[node].cx,pool[node].cy)]);
                                return;
                        }
                        if(pool[node].numParts<MAX_NUM) {
                                pool[node].particle[pool[node].numParts]=p;
                                pool[node].numParts++;
                        } else {
                                pool[node].child[0]=firstFree;
                                pool[node].child[1]=firstFree+1;
                                pool[node].child[2]=firstFree+2;
                                pool[node].child[3]=firstFree+3;
                                pool[firstFree].numParts=0;
                                pool[firstFree+1].numParts=0;
                                pool[firstFree+2].numParts=0;
```

```
                pool[firstFree+3].numParts=0;
                pool[firstFree].child[0]=-1;
                pool[firstFree+1].child[0]=-1;
                pool[firstFree+2].child[0]=-1;
                pool[firstFree+3].child[0]=-1;
                double s=pool[node].size*0.5;
                double hs=0.5*s;
                pool[firstFree].cx=pool[node].cx-hs;
                pool[firstFree].cy=pool[node].cy-hs;
                pool[firstFree].size=s;
                pool[firstFree+1].cx=pool[node].cx+hs;
                pool[firstFree+1].cy=pool[node].cy-hs;
                pool[firstFree+1].size=s;
                pool[firstFree+2].cx=pool[node].cx-hs;
                pool[firstFree+2].cy=pool[node].cy+hs;
                pool[firstFree+2].size=s;
                pool[firstFree+3].cx=pool[node].cx+hs;
                pool[firstFree+3].cy=pool[node].cy+hs;
                pool[firstFree+3].size=s;
                firstFree+=4;
                for(int i=0; i<MAX_NUM; ++i) {
                        int pi=pool[node].particle[i];
                        double px=pop.getx(pi);
                        double py=pop.gety(pi);
                        addRecur(pop,pi,px,py,pool[node].child[child(px,py,pool[node].cx,pool[node].cy)]);
                }
                pool[node].numParts=0;
                addRecur(pop,p,x,y,pool[node].child[child(x,y,pool[node].cx,pool[node].cy)]);
        }
}

template<class Population,class CollisionManager>
void recurseForAll(Population &pop,CollisionManager &cm,int node1,int node2) {
        if(pool[node1].child[0]==-1 && pool[node2].child[0]==-1) {
                for(int i=0; i<pool[node1].numParts; ++i) {
                        int j;
                        if(node1==node2) j=i+1;
                                else j=0;
                        for(; j<pool[node2].numParts; ++j) {
                                double t=pop.collisionTime(pool[node1].particle[i],pool[node2].particle[j]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(pool[node1].particle[i],pool[node2].particle[j],t);
                                }
                        }
                }
        } else if(pool[node1].child[0]==-1) {
                recurseForAll(pop,cm,node2,node1);
        } else if(node1==node2) {
                // This is a special case that prevents me from
                // doing extra work. In this situation I know they
                // all overlap and I only need to recurse in 10 different
                // combination, not 16.
                for(int i=0; i<4; ++i) {
                        for(int j=i; j<4; ++j) {
                                recurseForAll(pop,cm,pool[node1].child[i],pool[node2].child[j]);
                        }
                }
        } else {
                double x1=pool[node1].cx;
                double y1=pool[node1].cy;
                double x2=pool[node2].cx;
                double y2=pool[node2].cy;
                double s2=pool[node2].size*0.5;
                if(x2-s2-searchRadius<x1) {
                        if(y2-s2-searchRadius<y1) {
                                recurseForAll(pop,cm,node2,pool[node1].child[0]);
                        }
                        if(y2+s2+searchRadius>y1) {
                                recurseForAll(pop,cm,node2,pool[node1].child[2]);
                        }
                }
                if(x2+s2+searchRadius>x1) {
                        if(y2-s2-searchRadius<y1) {
                                recurseForAll(pop,cm,node2,pool[node1].child[1]);
                        }
                        if(y2+s2+searchRadius>y1) {
```

```
                                recurseForAll(pop,cm,node2,pool[node1].child[3]);
                        }
                }
        }
}

template<class Population,class CollisionManager>
void recurseForOne(Population &pop,CollisionManager &cm,int p,int ignore,int node) {
        if(pool[node].child[0]==-1) {
                for(int i=0; i<pool[node].numParts; ++i) {
                        if(pool[node].particle[i]!=p && pool[node].particle[i]!=ignore) {
                                double t=pop.collisionTime(p,pool[node].particle[i]);
                                if(t>=0.0 && t<pop.getTimeStep()) {
                                        cm.addPotentialCollision(p,pool[node].particle[i],t);
                                }
                        }
                }
        } else {
                double x1=pool[node].cx;
                double y1=pool[node].cy;
                double px=pop.getx(p);
                double py=pop.gety(p);
                if(px-searchRadius<x1) {
                        if(py-searchRadius<y1) {
                                recurseForOne(pop,cm,p,ignore,pool[node].child[0]);
                        }
                        if(py+searchRadius>y1) {
                                recurseForOne(pop,cm,p,ignore,pool[node].child[2]);
                        }
                }
                if(px+searchRadius>x1){
                        if(py-searchRadius<y1) {
                                recurseForOne(pop,cm,p,ignore,pool[node].child[1]);
                        }
                        if(py+searchRadius>y1) {
                                recurseForOne(pop,cm,p,ignore,pool[node].child[3]);
                        }
                }
        }
}

template<class Population>
void setSearchRadius(Population &pop,int node) {
        if(pool[node].child[0]<0) {
                for(int i=0; i<pool[node].numParts; ++i) {
                        for(int j=i+1; j<pool[node].numParts; ++j) {
                                double dx=pop.getvx(i)-pop.getvx(j);
                                double dy=pop.getvy(i)-pop.getvy(j);
                                double dz=pop.getvz(i)-pop.getvz(j);
                                searchRadius+=dx*dx+dy*dy+dz*dz;
                                cnt+=1.0;
                        }
                }
        } else {
                setSearchRadius(pop,pool[node].child[0]);
                setSearchRadius(pop,pool[node].child[1]);
                setSearchRadius(pop,pool[node].child[2]);
                setSearchRadius(pop,pool[node].child[3]);
        }
}

std::vector<QuadTreeNode> pool;
int firstFree;
int root;
double searchRadius;
double cnt;
};
```