2006

Visualizing the inner structure of N-body data

Edward Dale

Visualizing the Inner Structure of N-Body Data using Splatting and Skeletonization

by

Edward Robert Dale

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Supervised by

Hans-Peter Bischof
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
June 2006

Approved By:

Hans-Peter Bischof
Graduate Coordinator, Department of Computer Science
Primary Adviser

Stefan Harfst
Astrophysics Group, Department of Physics

Dan Batcheldor
Astrophysics Group, Department of Physics

Joe Geigel
Assistant Professor, Department of Computer Science

Thesis Release Permission Form

Rochester Institute of Technology

Golisano College of Computing and Information Sciences

Title: Visualizing the Inner Structure of N-Body Data using Splatting and Skeletonization

I, Edward Robert Dale, hereby grant permission to the Wallace Memorial Library reproduce my thesis in whole or part.

Edward Robert Dale

Date

Acknowledgments

I would first like to thank my thesis advisory committee, Dr. Hans-Peter Bischof, Dr. Stefan Harfst, Dr. Dan Batcheldor, and Dr. Joe Geigel, for their guidance. I would also like to thank Dr. William Dale for his support and the tales of his travels down the path I'm on now.

Abstract

N-body simulations solve the n-body problem numerically and determine the trajectories of the n point masses. The result of these calculations is a huge amount of data (up to tens of gigabytes) detailing the positions and other properties of each body such as mass and velocity. To effectively draw conclusions from this data, one must employ scientific visualization to create images and movies that illustrate the structure of the data. This thesis seeks to apply the computer animation technique of skeletonization to the volume data produced by n-body simulations in order to extract the inner structure of the data. This novel application will be compared to traditional point and volume rendering methods in terms of their ability to visualize astrophysical phenomena hypothesized to be present in the data. All of the techniques will be implemented in Java for the Spiegel visualization framework.

Contents

Acknowledgments . . . . . **iv**

Abstract . . . . . **v**

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Outline 2

2 Background and Related Work **3**
2.1 N-Body Simulations 3
2.1.1 The N-Body Problem 3
2.1.2 Algorithms 3
2.2 Visualization 4
2.2.1 Introduction 4
2.2.2 Related Visualization Systems 5
2.2.3 Spiegel Visualization Framework 5
2.3 Rendering 6
2.3.1 Introduction 6
2.3.2 Point Rendering 6
2.3.3 Volume Rendering 7
2.4 Skeletonization 11
2.4.1 Introduction 11
2.4.2 Algorithm 11

3 Implementation **13**
3.1 Data Structures 13
3.1.1 Quadtrees 13
3.1.2 Octrees 18
3.1.3 Octree Implementation 20

3.1.4 Input Data . . . 20
3.2 Program . . . 20
3.2.1 Extraction . . . 23
3.2.2 Filtering . . . 24
3.2.3 Classification . . . 24
3.2.4 Rendering . . . 28
3.2.5 Skeletonization . . . 29
3.2.6 Sample Program . . . 31

4 Results . . . 32
4.1 Dark Matter Random Reduction . . . 32
4.2 Rendering Method Comparison . . . 32
4.3 Skeleton Parameters . . . 35
4.4 Skeleton Tree Artifacts . . . 35
4.5 Skeleton Tree α Parameter . . . 35
4.6 Age-Related Visualizations . . . 38
4.7 Velocity . . . 43

5 Conclusions . . . 44

6 Future Work . . . 45

Bibliography . . . 47

A CD Contents . . . 50

List of Figures

2.1 $2 \times 2 \times 2$ volume with 1 white voxel and 7 grey voxels 7
2.2 Gaussian kernels with varying σ. 10
2.3 The results of the skeletonization algorithm 11

3.1 BSP Tree decomposition of a square 14
3.2 One level quadtree decomposition of a square 14
3.3 Sequence of inserting 7 points into a PR quadtree 15
3.4 Final space decomposition and associated PR quadtree 16
3.5 Sequence of inserting 7 points into a MX quadtree of height 3 16
3.6 Final space decomposition and associated MX quadtree of height 3 17
3.7 Final space decomposition and associated MX quadtree of height 2 17
3.8 Error introduced from voxelization 18
3.9 Location code calculation 19
3.10 Possible neighbor configurations 19
3.11 Octree class hierarchy 21
3.12 Data line format 22
3.13 Program architecture 23
3.14 Sample `SumOctree` 26
3.15 Effect of different density exponents. 27
3.16 Vertices per particle for different billboarding techniques. 29
3.17 A Spiegel program and the images it produces. 31

4.1 Random reduction of dark matter particles. 33
4.2 Different rendering techniques applied to gas particles 34
4.3 Comparison of different connectedness parameter settings for skeletal voxels and skeleton trees. Thinness: 2.0 36
4.4 Comparison of different thinness parameter settings for skeletal voxels and skeleton trees. Connectedness: 10 37
4.5 Sometimes the skeletonization parameters can hide complex features. . . . 38
4.6 Comparison of pairwise consecutive skeleton trees 39

4.7 Effects of α parameter on generated skeleton tree. 40
4.8 Star density for old and new stars. 41
4.9 Time of birth colored stars. 42
4.10 Velocity of star particles over time. 43

Chapter 1

Introduction

1.1 Motivation

The increasing speed of computers means that scientists are able to calculate and simulate situations that were previously impossible. The potential discovery of new truths resulting from these simulations is amazing. However, simulations don't produce truth as their end product. They produce data and likely lots of it. How to distill this large amount of data down to a form that can be digested is potentially a problem just as difficult as the initial simulation. The two issues involved are how to deal with the immensity of the data and how to map a physical property (density, velocity, etc.) to an image that can be understood to represent that property.

This thesis deals with interpreting the results of n-body simulations. The size of the data being visualized is on the order of 40 gigabytes. Examining this data without a visual representation is very difficult. To extract some kind of meaning, a concept called skeletonization will be borrowed from the field of computer animation and applied to scientific visualization. Skeletonization is an technique that, when applied to a dataset, yields a "thinner remnant that largely preserves the extent and connectivity of the original region" [9]. It has the immediate benefit of reducing the amount of data being processed. It will be shown that it can also be used in the mapping of physical properties to images. This technique will be compared to more traditional point and volume based methods.

1.2 Thesis Outline

In Chapter 2, a background summary of the concepts and subject matter being used will be presented. A large part of this work is the unification of common techniques from the fields of astrophysics, visualization, computer graphics, and computer animation. As such, the essence of each should be understood before any attempt to join them is made. The astrophysics topics being treated here will be confined to the n-body problem and it's simulation. The background of the visualization topics will be limited to a review of existing visualization systems and algorithms. The computer graphics background will be the largest as it will provide background for the two rendering techniques being compared. Computer animation is the source of skeletonization, the final topic to be discussed.

Chapter 3 will discuss the implementation of the entire system in terms of data structures and algorithms. When working with the amount of data provided by the n-body simulations, the data structure is of critical importance, therefore it will be highlighted in this chapter.

Following the implementation will be a discussion of the results in Chapter 4, replete with images exposing the inner structure of n-body data. An evaluation and comparison will be made of the ability of the three techniques to accurately extract structure from the data. Additionally, the algorithms will be compared in terms of their performance and execution time. Finally, some conclusions will be made about the viability of skeletonization as a visualization technique and what future work is possible.

Chapter 2

Background and Related Work

2.1 N-Body Simulations

2.1.1 The N-Body Problem

The n-body problem is the problem of finding the "trajectories in time of *N* point masses whose only interaction is Newtonian gravitation" [16]. This is a very computationally expensive problem ($\mathcal{O}(n^2)$) as each body exerts a force on every other body according to Newton's law of gravity. A closed solution only exists for the $n = 2$ case. For all $n > 2$, numerical approximations must be used. Typically, an n-body simulation is used to model another process. In this thesis, the simulations being visualized are models of the evolution of galaxies consisting of stars, molecular clouds, gas, and dark matter.

2.1.2 Algorithms

Because of the immense computational complexity of the n-body problem, a number of approximations have been created that make the problem more reasonable. The most successful thusfar is the TREE algorithm [1], which uses the same data structure implemented in this thesis, an octree (see Section 3.1).

The TREE algorithm is based on the assumption that from a far enough distance, a group of particles can be treated as a single particle. At short distances, smaller groups of particles are used in the computation. This has reduced the $\mathcal{O}(n^2)$ complexity of the n-body

problem to $\mathcal{O}(n \log n)$. Further improvements have been made in [5] reducing complexity to $\mathcal{O}(n)$ in some cases.

The method above was used in tandem with smoothed particle hydrodynamics (SPH) and chemo-dynamical (CD) models to model galaxies in [11]. It is data from simulations done using this technique that will be visualized in this thesis.

Just as these simulations take a lot of time to execute, they also produce a lot of output. For every particle in the system, it's position, velocity, mass and any other physical properties simulated. Section 3.1.4 discusses the data format in detail.

2.2 Visualization

2.2.1 Introduction

"Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations." [2] The amount of data produced by the n-body simulations is so great as to necessitate a tool that can extract some meaning. Visualization is the tool that can provide that insight by creating images of the data. However, images are only one piece of the puzzle. To truly be able to understand the dataset, the tool should allow one to navigate through time and space creating movies that can be analyzed.

Visualization can be categorized into online and offline. Online visualization implies some type of interaction between the visualization tool and the system producing the data (typically, a simulation). Taking this to an extreme, the simulation system and the visualization tool can be completely integrated. A better approach might be to have the simulation provides hints in the form of control data to the visualization system to tell it what might be interesting. Likewise, the visualization tool could tell the simulation what events have been discovered to instruct it to focus on those. Offline visualization can be thought of as a batch processing job to visualize a static set of data. Except where noted, the tools below do not enforce either paradigm.

2.2.2 Related Visualization Systems

The Visualization Toolkit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization. VTK aims to provide high-level abstractions for common visualization data structures and algorithms. The benefit of this is clear in light of the time spent in the implementation of common data structures and algorithms for this project. Solving a visualization problem is a matter of writing code that interfaces with the VTK libraries [15].

VisIt is a free interactive parallel visualization and graphical analysis tool for viewing scientific data. Similar to VTK, it has primitives for many common visualization requirements. The basic paradigm for solving a visualization problem is to apply one or more operators data and then plot it using one of the tools provided (mesh, pseudocolor, etc.) [18].

Xnbody is a an online visualization tool designed for the direct n-body code. The n-body simulation is instrumented with code to send current data to xnbody to be visualized using VTK. In this way, xnbody is an adapter to allow online visualization of a simulation using VTK and itself does little in the way of simulation or visualization [13].

2.2.3 Spiegel Visualization Framework

Spiegel is a visualization framework for large- and small-scale systems that uses a pipeline approach similar to that of Unix to create a visualization. Functional blocks are assembled in the pipeline using a graphical programming environment. A visualization problem is solved by implementing new and reusing existing blocks to solve individual tasks [21]. This thesis will be implemented in Spiegel using the constructs it provides and creating many more to implement the algorithms described below.

2.3 Rendering

2.3.1 Introduction

Rendering is the process of generating an image from a model. The image is a two-dimensional array of RGB (red, green, blue) pixels. The model, at the lowest level, is the data being generated by the n-body simulation. To facilitate rendering, a more complex model will be overlaid on the n-body data in Section 2.3.3.

A number of rendering methods will be discussed below, but they all share the need to transform the model to an RGB value. This step is called classification and the mapping is done using a transfer function. Because of the unapparent nature of this mapping, this is the most complex task in extracting the inner structure of data. For example, if one is rendering velocity, what color/opacity best signifies "fast"? Another challenge of classification is to not have the opacity become completely saturated before the data is completely traversed.

Creating the transfer function is typically done using trial-and-error [7] and this is the approach taken below. There has been research done to attempt to create transfer functions more automatically. In [7], the transfer function is defined as a sequence of three-dimensional image processing operations with tunable parameters. In [14], it is posited that a volume should suggest an appropriate transfer function. The authors go on to describe a *histogram volume* that is used to create a transfer function.

2.3.2 Point Rendering

The first technique being compared is to render the data as discrete points. Looking at the n-body data, creating a graphical point for each data point would be a natural technique. However, artifacts begin to manifest themselves where data is particularly sparse. In this case, the individual points become distinguishable when what is desired is a more diffuse, cloudy appearance.

2.3.3 Volume Rendering

Introduction

If the space to be visualized is treated like a volume, a number of advanced algorithms become available. A volume is a dataset defined on a three-dimensional lattice with one or more scalar or vector values at each gridpoint on the lattice [6]. In Figure 2.1, a $2 \times 2 \times 2$ volume is shown. Each white circle is a gridpoint and the space containing each gridpoint is called a voxel. Per convention and to simplify calculations, the gridpoint is located in the minimum position of the voxel. The position of the white voxel in Figure 2.1 is $(0.5, 0.5, 0)$.

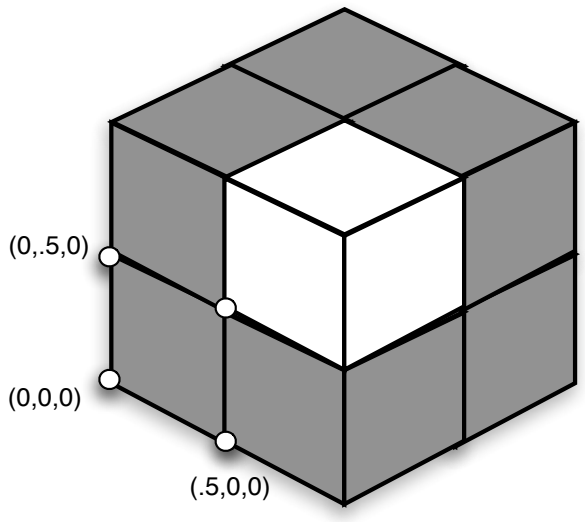


Figure 2.1: $2 \times 2 \times 2$ volume with 1 white voxel and 7 grey voxels

Except for the lattice requirement, n-body data fits this definition. The process of voxelization, the first step of the implementation below, resamples the input data onto a regular grid to prepare it for volume rendering.

Two major types and three primary volume rendering algorithms will be discussed below. Indirect volume rendering algorithms render a model that is based on the volume data whereas direct volume rendering attempt to map voxels with no intermediate model.

Indirect Volume Rendering

Indirect volume rendering (IVR) methods are so called because they convert the volumetric data into a polygonal representation which is then rendered. This technique is sometimes referred to as surface fitting, stemming from the fact that it is used to render isosurfaces in volume datasets. Isosurfaces are determined by finding all voxels that cross a particular threshold value and creating polygons between them. IVR typically only has to traverse the data once in order to extract and polygonize surfaces which can then be rendered and manipulated using standard methods [6]. Isosurfaces have the disadvantage of being incapable of showing the continuity of a dataset; a given datum is either inside the isosurface or not. Considering that a major advantage of volume rendering is the ability to render "structures of varying depth that attenuate traversing light" [6], isosurfaces are not an optimal solution. They are, however, useful for isolating larger surfaces without fine features in a dataset. In this case, other methods (e.g. direct volume rendering) would be used to render the macrostructure of the data with indirect volume rendering being used to draw attention to particular areas of the volume. Small features are not as easily exposed because of the lack of resolution provided by isosurfaces. To accurately model a complex isosurface would require a large number of polygons, reducing the advantage of using IVR.

Marching cubes [19] is the original algorithm used to determine isosurfaces. In marching cubes, the volume data is traversed in scan-line order, the surface found, and triangle vertices calculated using linear interpolation. Normals and shading are then calculated using gradients from the original data.

The fact that IVR still uses polygon rendering in the end limits its usefulness in galactic visualization. IVR cannot create images that show the cloudy nature of n-body data, which is desirable. It could possibly be used to draw attention to the arms of a spiral galaxy or the location of a black hole, but the structure as a whole needs a more advanced technique. For this reason, indirect volume rendering was not implemented.

Direct Volume Rendering

Direct volume rendering methods forego the middle step of polygonization and map the voxels directly to pixels. This mapping can occur a number of ways, however, the two classic ways are ray casting and splatting.

Volume Ray Casting

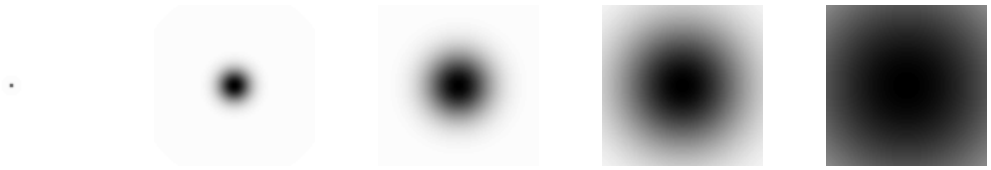
Ray casting can be thought of as the volume equivalent of ray tracing. A ray is followed through the volume and voxels or interpolated voxels are sampled at regular intervals. It is an image order algorithm because the image is built pixel-by-pixel as rays are fired into the scene. One implementation of volume ray casting uses a three-dimensional version of Bresenham's line drawing algorithm to create the ray path [12]. As with ray tracing, this is a very elegant, but slow algorithm.

There are numerous acceleration techniques that apply to ray casting and fall into two categories: "do less", "do it faster". The main method to reduce the amount of work to do is early ray termination. Once a pixel has reached maximum opacity, examining further voxels along the ray won't yield any additional information, so that ray can be terminated [17].

The next way to process less voxels is to skip empty space. This necessitates an additional data structure to keep track of stretches of empty space. This structure is built using a coarse traversal of the volume recording empty spaces and their extent [17].

Doing things faster usually means taking more advantage of hardware. There's been a lot of progress made in recent years in crafting algorithms that can run efficiently on the GPUs[1] present in the typical desktop computer. In addition, there are some very expensive dedicated hardware cards that process volume data using ray casting algorithms [22].

[1] Graphics Processing Unit - The CPU of a graphics card.



(a) $\sigma = 0.01$ (b) $\sigma = 0.10$ (c) $\sigma = 0.20$ (d) $\sigma = 0.40$ (e) $\sigma = 0.80$

Figure 2.2: Gaussian kernels with varying σ.

Volume Splatting

Splatting can be thought of as taking each voxel and throwing it, as one would with a snowball, at the image plane. Each voxel is projected onto the image plane using a gaussian kernel function scaled by the distance of the voxel from the image. The volume is traversed from back to front yielding a rough approximation that is progressively refined as more slices are processed [20]. This is an object order approach, which is also sometimes called forward mapping.

An important relationship to notice is that point rendering can be implemented using volume splatting with a gaussian kernel with a very low σ, approximately a discrete point. This reduction shows that the power of volume splatting comes from blurriness that a medium-σ gaussian kernel provides. A number of different gaussian kernels are shown in Figure 2.2.

Similar to early termination in ray casting, splatting has early splat elimination. An additional data structure storing the occluded parts of the image plane is necessary. This saves the work of rasterizing the gaussian kernel on the image plane, but the transformation to determine the position of the voxel on the image plane is still necessary.

2.4 Skeletonization

2.4.1 Introduction

Skeleton extraction and manipulation is a very intuitive technique for volume animation. The main benefit being the large body of algorithms and techniques for dealing with articulated skeletons stemming from the research done regarding character animation. Skeletons also capture the essential topology of an object. Figure 2.3 shows a rectangle and it's skeleton. Notice that the main components of it's structure are preserved: the four corners and lengthened body. For the purposes of this thesis, a skeleton consists of the set of voxels that form the medial surface (centered with relation to the boundaries) of the volume. A centerline is curve-like representation of the medial surface. The skeleton and centerline need not be completely connected, however, a later step in the algorithm discussed below ensures this. Both of these representations have a single parameter that determines how thin the resulting object should be [10].

Figure 2.3: The results of the skeletonization algorithm

2.4.2 Algorithm

Extracting a skeleton from a volume is done by thinning that volume until a desired thinness is reached. This is the thinning parameter that was mentioned earlier. The method described in [10] uses a weighted distance transform that eliminates the need for costly

floating point operations. The algorithm proceeds by propagating boundary distances inward until there are no new voxels. For each voxel, the average of all its neighbors is calculated and compared with the thinning parameter to determine if it is a skeletal voxel.

It's possible to stop the algorithm at this point and use just the unconnected skeletal voxels. To create an actual hierarchical skeleton, a graph theory approach is taken. A completely connected graph is created with the skeletal voxels as the vertices. The edge weights are a linear combination of spatial distance between two voxels and the difference in their distance transform values. The minimum spanning tree of this graph is the skeleton tree of the dataset.

Chapter 3

Implementation

3.1 Data Structures

When dealing with particle numbers in the hundreds of thousands, the data structure used needs to be sufficiently advanced to mask the inherent complexity. The ideal data structure would divide the input space into manageable subspaces. There are a number of common spatial data structures including Binary Space Partitioning (BSP) trees, k-d trees, quadtrees, and octrees. These structures represent a space as a tree with different divisions and number of child nodes at each level.

The BSP tree divides the space into two arbitrary size parts at each subdivision stage using a hyperplane. Divisions are not necessarily orthogonal nor parallel. For a two-dimensional space, this yields a decomposition of arbitrarily shaped polygons (Figure 3.1). In three dimensions, the result is a set of polyhedra. The BSP tree resulting from the decomposition of the n-body data would be too irregular to be of use, so it was not implemented. A regular subdivision of a space can be achieved with a region octree and it's two-dimensional analogue, the region quadtree, simplifying many of the calculations.

3.1.1 Quadtrees

This section will be written in terms of quadtrees because they are easier to describe and illustrate in two-dimensional space than octrees. The theory of the two structures will be

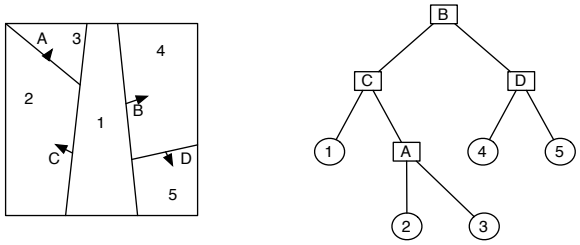


Figure 3.1: BSP Tree decomposition of a square

discussed in this section followed by implementation details in Section 3.1.2. The implementation of these data structures was heavily inspired by [23, 24], where the reader is directed for a complete treatment of spatial data structures.

Quadtrees are similar to BSP trees in that they divide a space at each level of the tree. The difference is that that quadtrees divide the space in both dimensions in half (Figure 3.2). From the figure, one can see that a quadtree divides the space into $2^2 = 4$ subspaces at each level. This can be generalized to d dimensions by observing that the quadtree will divide each dimension in half yielding 2^d subspaces at each tree level. This shows that quadtrees and octrees are just two- and three-dimensional specializations of such a decomposition.

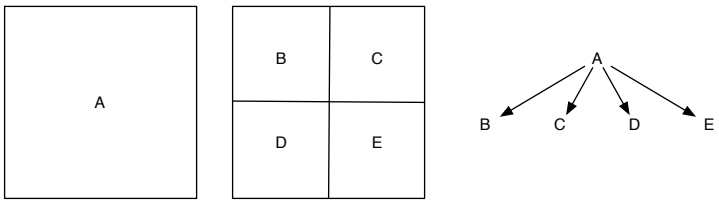


Figure 3.2: One level quadtree decomposition of a square

Obviously, a space could be decomposed into infinitely many boxes, so a criteria for decomposition must be established. Different criteria yield quadtrees with different behaviors, which may be better suited a particular use. Two such criteria were implemented to fill two different needs: the PR quadtree and the MX quadtree.

The PR (Point Region) quadtree recursively divides a space down to regions in which at most one point exists. Subdivision stops when each point is in a region by itself or a maximum tree height is reached. Reaching the maximum tree height is an issue common to both quadtree types, so it is discussed below.

Figure 3.3 shows the decomposition sequence when inserting 7 points into a space. Notice how the initial state of the space is empty. This is a valid quadtree and is indeed the simplest possible decomposition of a space. There is no decomposition of the space when the first point is added because there is still ≤ 1 point in every box. At steps III, IV, and VII, the space is decomposed and another 4 children are added to one of the tree nodes. Figure 3.4 shows the final decomposition and the associated PR quadtree.

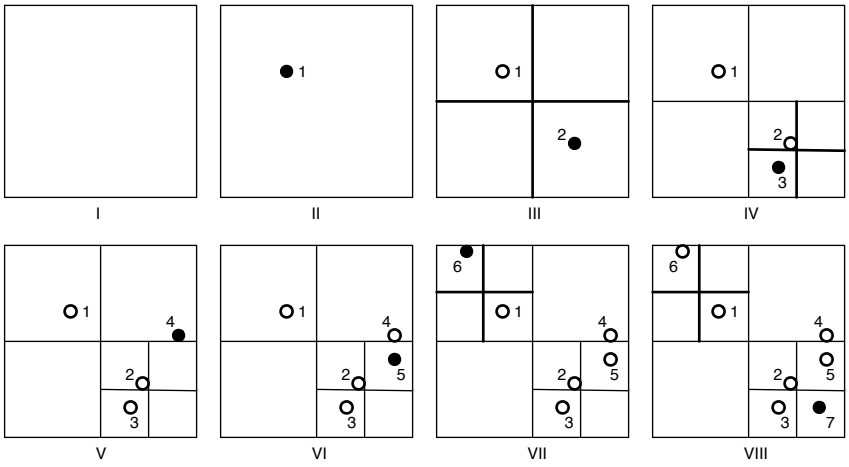


Figure 3.3: Sequence of inserting 7 points into a PR quadtree

The MX (MatriX) quadtree similarly divides a space into regions, except the sizes of the regions are constant. When inserting a point into an MX quadtree, the space will be recursively decomposed until the desired size is met. This is best expressed by saying that the tree representation of an MX quadtree will have a set height and every point will exist in the lowest level.

Figure 3.5 shows the same 7 points being inserted into an MX quadtree of height 3. Notice that inserting the first point does 2 decompositions as opposed to the 1 that was

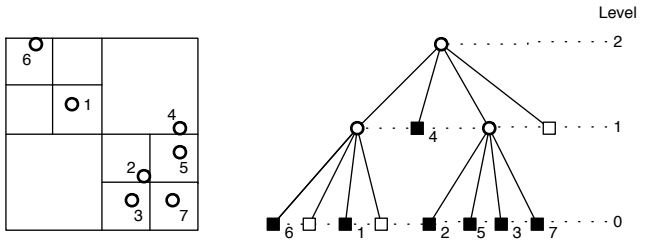


Figure 3.4: Final space decomposition and associated PR quadtree

done for the PR quadtree. Figure 3.6 shows the final space decomposition and associated MX quadtree. The fourth point is located at the level 0 of the tree now instead of level 1, causing an extra internal (round) node to be created. This illustrates the fact that an MX quadtree will likely use more space than a similar PR quadtree.

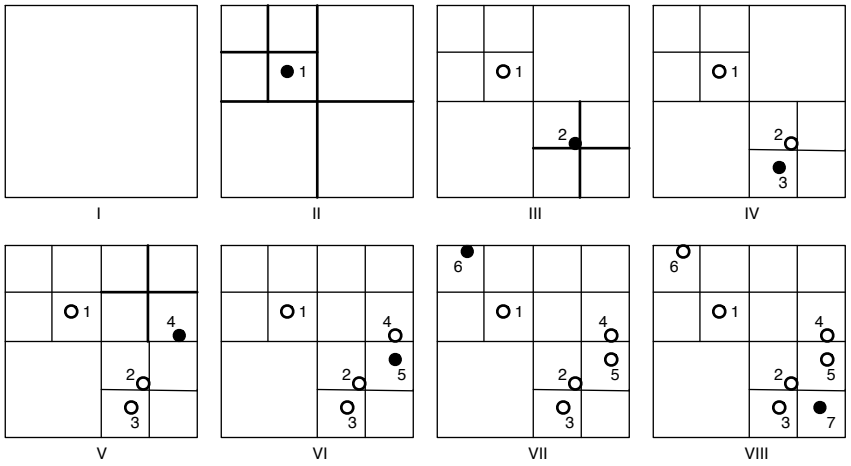


Figure 3.5: Sequence of inserting 7 points into a MX quadtree of height 3

Although an MX quadtree will take up more space than a similar PR quadtree, it has the particularly good property that every point is located in an equally-sized region. This is very useful in the density calculations described in Section 3.2.3. For situations where this equal sizing isn't needed, a PR quadtree works equally well as an MX quadtree.

Both types of quadtrees have the same issue of what to do when it's not possible to

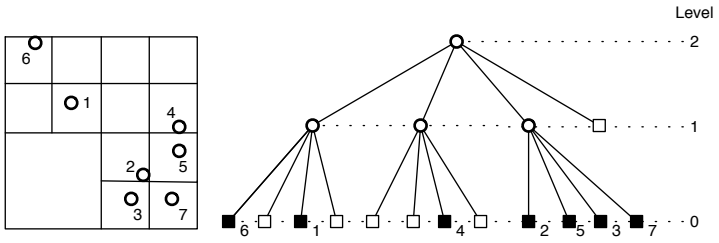


Figure 3.6: Final space decomposition and associated MX quadtree of height 3

divide a space any further to insert a point. This will happen anytime a leaf node with a point is encountered at level 0. There are a couple strategies one could take. The height of the quadtree could be increased, but this has the very undesirable side effect that all of the nodes in an MX quadtree would have to be shifted down a level. Alternatively, a linked list could be created at level 0 and points prepended to it when division isn't possible. This is called chaining and is similar to chaining from hashing. If the MX quadtree from Figure 3.6 was instead created with a height of 2, the MX quadtree in 3.7 would result.

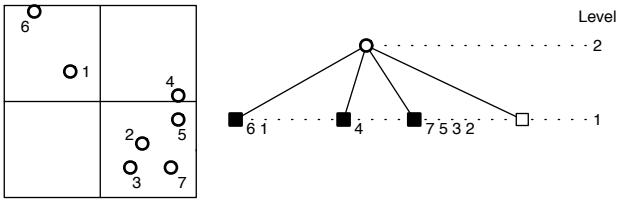


Figure 3.7: Final space decomposition and associated MX quadtree of height 2

As mentioned in Section 2.3.3, the first step in rendering a volume is voxelization, converting it to a regularly, discretely sampled dataset. Fortunately, this happens implicitly when the points are inserted into the quadtree. Once inserted, the location of a point can be treated as the location of the region that it occupies. The regions of a quadtree are all squares, so the dataset becomes regular and discrete.

3.1.2 Octrees

Some positional error may be introduced from voxelization, but if the tree is deep enough, this error becomes negligible. The octrees were created were mostly of height 16. This decomposes the space into $2^{16} \times 2^{16} \times 2^{16}$ cubes. In the worst case, the actual location of a point will be on the exact opposite side of the region it is placed in. The dimension of a single region is $\frac{1}{2^{16}}$, so the possible error is $\frac{\sqrt{3}}{2^{16}}$. This is illustrated in Figure 3.8.

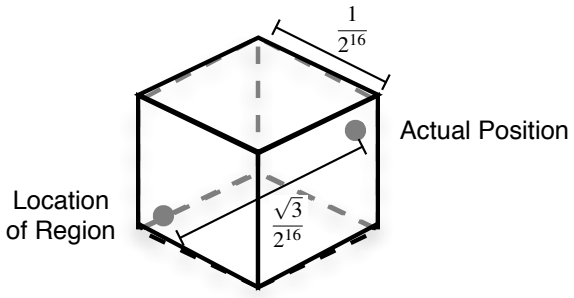


Figure 3.8: Error introduced from voxelization

Insertion and region location is accomplished using location codes as described in [8]. A location code is calculated for each dimension by multiplying the coordinate value by 2^{ROOT_LEVEL}, truncating it, and representing it as a binary value. At each level k of the octree, the $(k-1)st$ digit of the location code is used to determine which branch to take. The location code is processed until the desired region is found or until the insertion criteria described above is met. A one-dimensional version of this calculation is shown in Figure 3.9.

Insertion and region location operations on both types of quadtrees have a time complexity $\mathcal{O}(\log(height))$. Moreover, the time complexity when inserting into an MX quadtree is $\Theta(\log(height))$ because the full height of the tree will always be traversed. The PR quadtree will tend towards this as data density increases. For extremely dense datasets,

more instances of chaining will occur, but these will not affect insertion complexity because chained nodes are inserted at the head of the linked list.

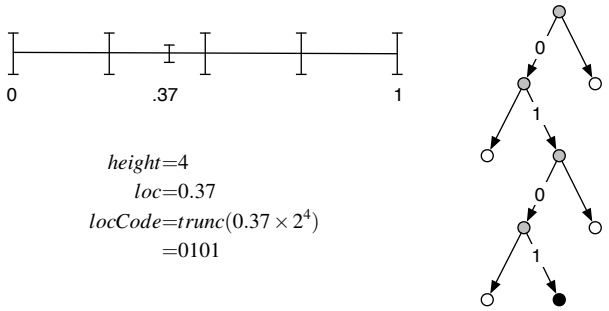


Figure 3.9: Location code calculation

The only other operation needed is one to find all the neighbors of a leaf node, a requirement of the skeletonization algorithm discussed in Section 2.4.2. Note that a neighboring node can be either a vertex, edge, or face neighbor depending on whether the two nodes share 0, 1, or 2 location codes, respectively (Figure 3.10).

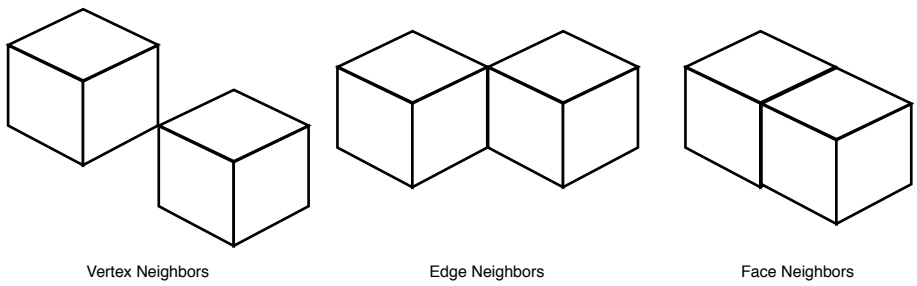


Figure 3.10: Possible neighbor configurations

The process of finding the neighboring nodes of a node A again uses the nodes' location codes. First, the location code of a possible neighbor N is found by adding (or subtracting for a neighbor on the opposite side) 1 to A's location code. A *difference code* is then calculated by XORing the two location codes. The difference code is used to find a common ancestor between the two nodes. From the common ancestor, N's location code is used

to traverse to *N*, if it exists. According to [23], these operations are all bounded by by constants less than 5. It follows that any traversal from one node to another can be done in linear time.

3.1.3 Octree Implementation

The `Octree` class hierarchy shown in Figure 3.11 is designed to allow maximum flexibility. If a behavior needs to be added to an `Octree`, it is extended and one or both of the `create*Node` methods are overridden. After classification, an `Octree` implementing `ColorableOctree` is sent to the splatting code. A `ColorableOctree` knows how to convert the values stored in the leaf nodes into an RGBA value that can be rendered.

3.1.4 Input Data

The n-body data will be the result of calculations from [11] and will come in the form of a number of text files, optionally GZipped, each representing a snapshot of the system at a certain time. Each file has a three line header containing the snapshot number (starting at zero), the number of total particles as well as the number of each type of particle (star, cloud, gas, dark), and the time of snapshot. Following the header is N lines in the format defined in Figure 3.12 and Table 3.1.

3.2 Program

As mentioned in Chapter 1, all implementation was done in the Spiegel visualization framework. Spiegel offers a choice of Java3d or JOGL (Java OpenGL bindings). Due to the wealth of OpenGL resources available online and the performance increase studied in [4], JOGL was chosen. Spiegel enforces a separation of concerns, so all implementation details will be discussed in terms of functions and their inputs, outputs, and parameters.

The general architecture (Figure 3.13) is: extraction, filtering, classification, rendering. Data extraction is handled by the *OctreeExtractor* and will be discussed in Section 3.2.1.

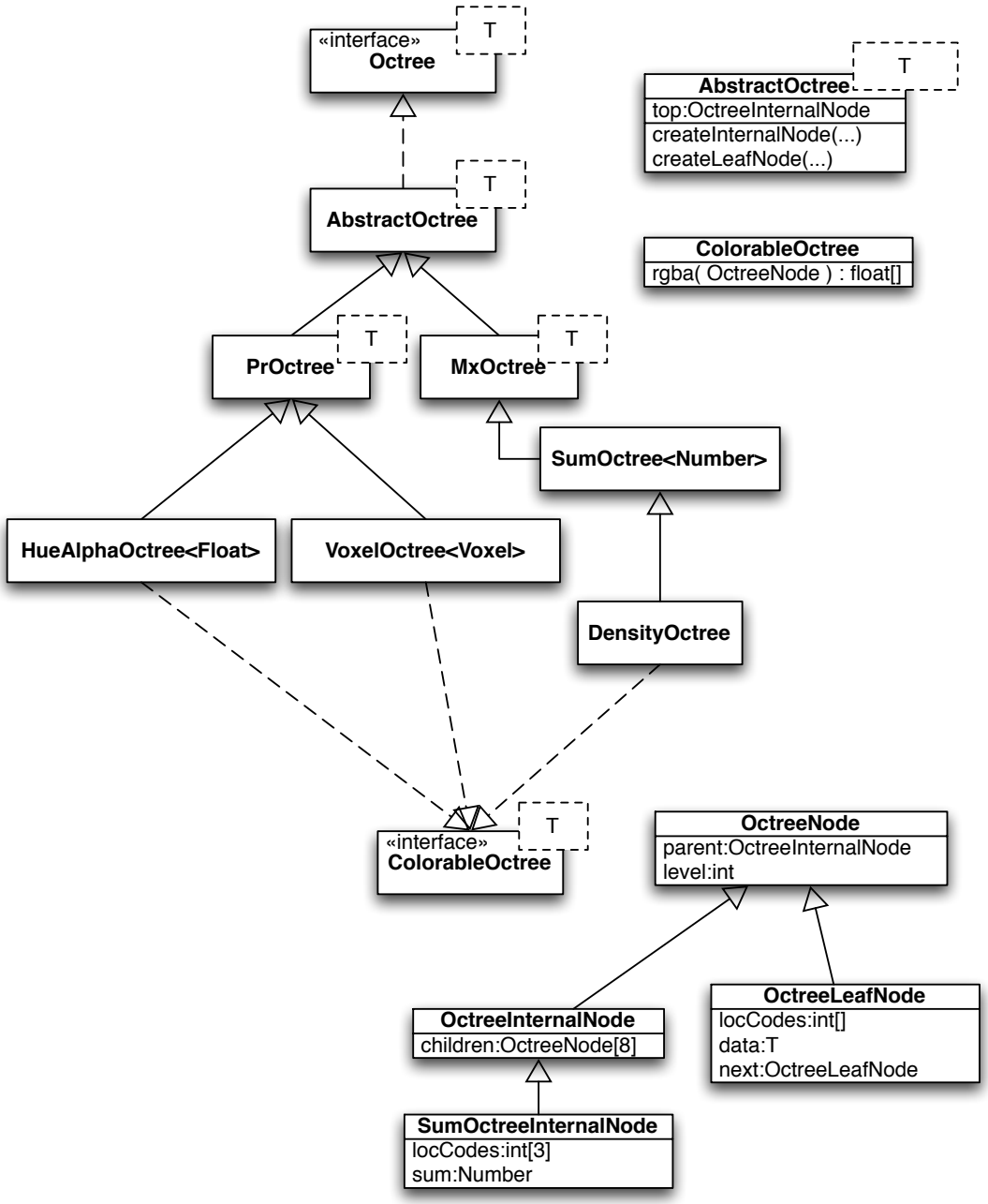


Figure 3.11: Octree class hierarchy

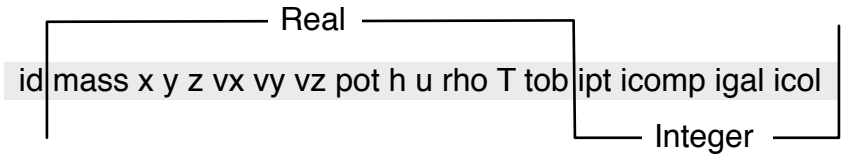


Figure 3.12: Data line format

Table 3.1: Data line format

	Description
id	Particle index starting from 1 (Not unique)
mass	Particle mass (Not constant)
x y z	Particle position
vx vy vz	Particle velocity
pot	Potential
h	Particle radius (Undefined for stars and dark matter but gives a true radius for clouds and the smoothing length (kernel size) for gas (SPH) particles)
u	Internal specific energy (Only definded for gas particles (energy per unit mass))
rho	Mass density of the gas at the position of particle (Only defined for gas and cloud particles and possibly stars)
T	Temperature, otherwise same as rho
tob	Time of birth (When a particle was introduced into the simulation, only meaningful for stars and clouds. Needs to be divided by 0.6711 to be in correct myr units)
ipt	Particle type: 1=stars, 2=clouds, 3=gas(SPH), 4=dark matter
icomp	Galaxy component: 1=disc, 2=bulge, 3=halo
igal	Galaxy identifier if two or more galaxies
icol	The same as ipt

Filtering, getting rid of particles that aren't of interest, will be discussed in Section 3.2.2. Classification and rendering are computer graphics heavy topics and will be covered in Sections 3.2.3 and 3.2.4. Weaving through this pipeline is skeletonization, which will act on the unclassified data and provide input to the classification and rendering stages. This will be discussed in Section 3.2.5.

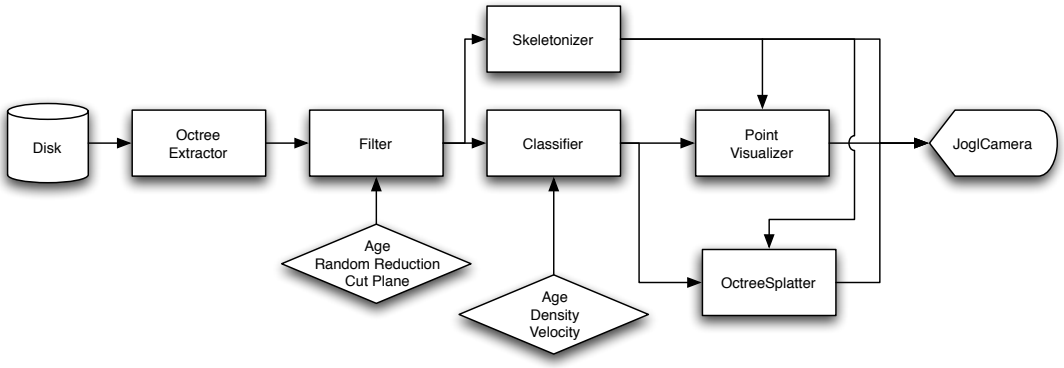


Figure 3.13: Program architecture

3.2.1 Extraction

The *OctreeExtractor* accepts a filename which is expected to be in the format described in Section 3.1.4. If this is the first time a file has been read by this extractor, it makes one pass through the file to determine the extent of the data. An optional scale value can increase the size of the space manually if needed in case particles leave the space defined by the first timestep during the simulation. If the extent has already been determined previously, it is used.

With the extent of the data calculated and the octree height given as a parameter, a `PrOctree` is created for each particle type (star, cloud, gas, dark matter). Each line of the file is then inserted into the corresponding octree. These four octrees are the outputs for the *OctreeExtractor*, along with the timestep that was extracted.

3.2.2 Filtering

Filtering applies a predicate to all the voxels in an octree in order to remove some subset of them and reduce the amount of data being classified and displayed. It's important to realize that filtering is just a special case of classification. A filtered voxel has the same visual appearance as one that has been assigned an opacity of zero. Instead of removing the filtered voxels from the octree, a new octree is created at each filter to which the non-filtered voxels are added.

Three filters were created to help extract structure from the data. The most useful was the `AgeFilter`, which removes particles older than a certain age. This filter looked at the *tob* (time of birth) property of a particle and compared it with the current time. It also had a parameter to invert the comparison to find particles younger than a certain age. This filter, when applied to stars and combined with the standard density classifier is what exposed the tidal dwarf galaxies discussed in the next chapter.

The `CutPlane` was the second filter that was created. It cuts a dimension in half and accepts particles that exist in the desired side. This was useful to look inside a splatted volume. Ideally, though, a cut plane should be unnecessary with a transfer function that manages opacity well.

The final filter that was implemented was the `RandomReduction` filter. It uses a uniform probability distribution to determine whether a particle should pass the filter. This was only used with dark matter in an attempt to find some structure by thinning it.

3.2.3 Classification

The initial work into classification was done by means of a generic `Classifier` function. Java code could be entered into a parameter of the function and compiled on the fly and applied to the octree. This amounted to a classification prototyping system. Most of the other classifiers that were built were initially created in this manner. The classifiers that will be discussed are the `VelocityClassifier`, `AgeClassifier`, and

`DensityClassifier`, in order of importance and utility.

The output of all of the classifiers is in terms of hue, part of the hue, saturation, brightness (HSB) color representation. This was chosen because hue is a real number on the range [0..1], much like the scaled values to be rendered. Saturation and brightness were chosen in order to have a very bright, pure color that did not vary within an image. Any variation in brightness could possibly be misconstrued as a variation in opacity, which is usually mapped to another quantity.

The `VelocityClassifer` was the first and most trivial classifier created. It does exactly what one would expect and maps the velocity to an RGB value. The velocity is scaled down to [0..1] and interpreted as the hue of a HSB tuple. The `AgeClassifier` scales particle age to [0..1] and then renders this as a color similar to the `VelocityClassifier`. In both cases, the alpha value of the voxel can be a function of the scaled input or a constant. The `DensityClassifier` differs from the other classifiers in that the hue it renders is constant whereas the alpha value is dependent upon the value in the octree. A full discussion of the formulation of this alpha value can be found in the next section.

Density Calculation

Deciding which volume to measure the density of is a decision that needs to be made. The method implemented below is to measure the density of the space defined by a node at a certain level of an octree. In astrophysics, because the space has not necessarily been voxelized and stored in an octree, a density estimator is used. The order of the density estimator determines the volume that the density is calculated in. A density estimator of order j centered on a particle a will calculate the density of a sphere centered at a and extending to contain the closet j stars to a. A more complete discussion of density estimators as applied to astrophysics can be found in [3].

One of the most direct ways to discern the inner structure of the data is by splatting the density of the volume at every voxel. A `MxOctree` subclass was created called a `SumOctree` which accumulates the sum of the value of the leaf nodes in all of the internal

nodes between each leaf and the root of the tree. A sample quadtree implementing this behavior is shown in Figure 3.14. Notice how every internal (round) node contains the sum of all of its children. Density calculation is rather straightforward using a `SumOctree` if mass is the value that the `SumOctree` accumulates, because every node in an `MxOctree` has the same volume that can be dropped from the equation. The density for each node then becomes the sum of its children.

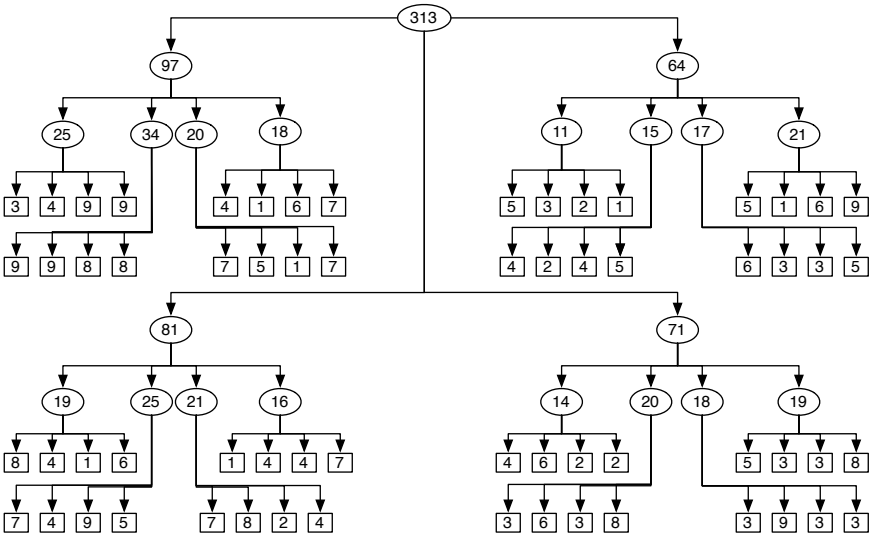
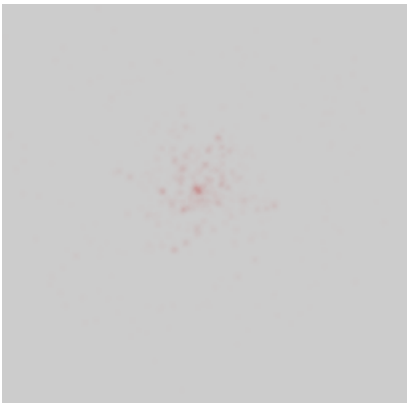
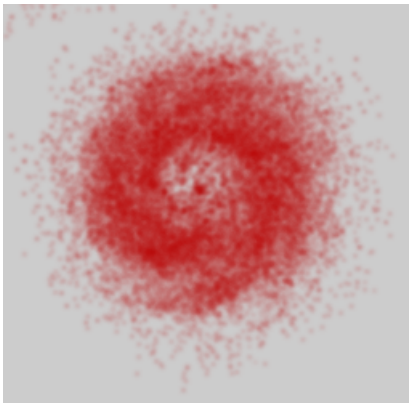


Figure 3.14: Sample `SumOctree`

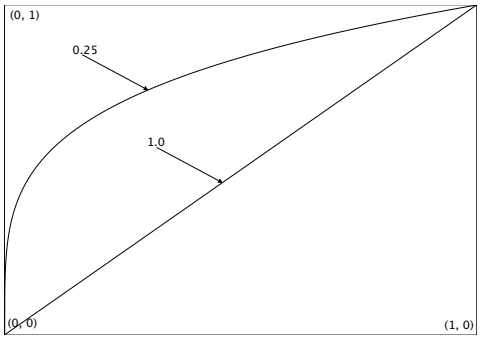
Opacity was used to visualize density under the premise that a dense material absorbs more light, as fog would. To this end, the density value from the `SumOctree` was scaled to $[0..1]$ and used directly as the alpha value of the splat. This yielded a very sparse image as seen in Figure 3.15a. In order to have the lower density areas contribute more to the image, the density was raised to the power of 0.25 (Figure 3.15b), bringing out much more of the structure of the particles. Figure 3.15c shows the weighting that is applied to the density for both functions.



(a) Exponent=1.0



(b) Exponent=0.25



(c) Density Curve

Figure 3.15: Effect of different density exponents.

3.2.4 Rendering

Point Rendering

As discussed in Section 2.3.2, point rendering in the simplest of the studied rendering methods. As such, the implementation is the most straight forward. The *PointVisualizer* is a `Drawable` function that accepts an `Octree` as input and simply renders a point at every occupied position in the tree. If the input is a `ColorableOctree`, then the point output will be colored.

Volume Rendering

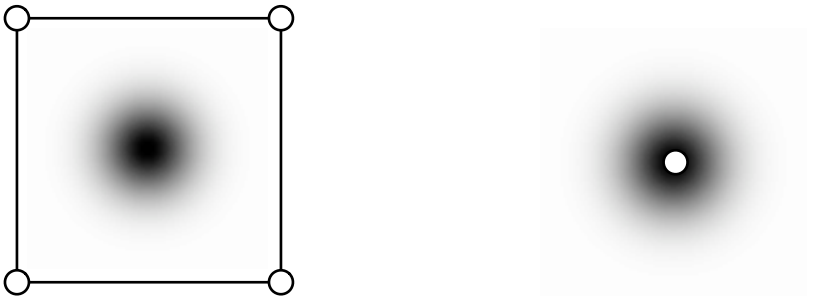
Volume splatting is the volume rendering algorithm that was chosen for this portion of the thesis. It requires that a gaussian kernel be projected onto the image plane for each point being rendered. To this end, a `GaussTextureSplat` class was created to contain this behavior. It takes advantage of OpenGL textures to create the gaussian kernel once and stores it on the graphics card. Two methods of rendering the `GaussTextureSplat` were created, with different graphics cards requirements.

The first GTS implementation requires no special OpenGL extensions. For every point to be rendered, a quad polygon is created that is textured with the gaussian kernel. This polygon is billboarded [1] and sent to the graphics card. The billboarding process is the most computationally expensive step with this method as 14 floating point vector arithmetic operations have to be performed. The billboarding algorithm used is from [25].

For systems with a modern graphics card, a lot of this work can be pushed off to OpenGL and the GPU. If a system supports the OpenGL point sprite extension, then point sprites are used to render the gaussian kernel. Point sprites are points that can be textured. In this implementation of the `GaussTextureSplat`, texturing is enabled and then each point is sent straight to the video card.

Point sprites have a distinct advantage over manual billboarding in that the CPU needs

[1] rotated to face the camera



(a) Polygon Billboard (4 vertices)

(b) Point Sprite (1 vertex)

Figure 3.16: Vertices per particle for different billboarding techniques.

to do less because there are no manual vector operations, and the GPU needs to do less because a quarter of the number of points are being sent to the video card. This is illustrated in Figure 3.16; the white dots are vertices that need to be sent to the video card to be rendered.

OpenGL has the ability to query what extensions are available. This is used to determine the best rendering primitive to use. The other requirement of the gaussian kernel is that it is scaled to be smaller when it is farther away from the image plane. This is implicit in the perspective projection that the OpenGL uses.

3.2.5 Skeletonization

The skeletonization algorithm described in Section 2.4 had to be modified slightly because of the characteristics of the volume data being processed. The volume data that the algorithm was designed for is typically a solid block of voxels, some of which are object voxels and the rest of which are background voxels (*i.e.*Figure 2.3). However, because of the scatteredness of n-body data and the initial voxelization step, the data will be regular, but there won't necessarily be a strict border between object and background voxels.

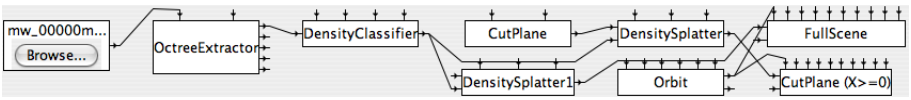
As implemented here, if a voxel is present, it is an object voxel. There are two ways to

tune this. The first is to modify the height of the octree input to the skeletonization algorithm. This is a critical parameter as too many or too few initial voxels will make accurate skeleton extraction impossible. To reduce the number of object voxels, a heuristic approach was taken to determine the type of a voxel. A parameter determines the connectedness that a voxel must have to be an object voxel. In practice, a connectedness value of 10 yielded an accurate skeleton.

When determining the skeletal voxels, the only other parameter is the thinness parameter *tp*. The distance transform of a voxel *q* needs to be within *tp* of the mean of all 26 neighbors of *q* in order to be a skeletal voxel. For low values of *tp*, the distance transform of a voxel need be only a little higher than it's neighbor to be a skeletal voxel, yielding a thick skeleton. A higher value will create a much more disconnected skeleton [9].

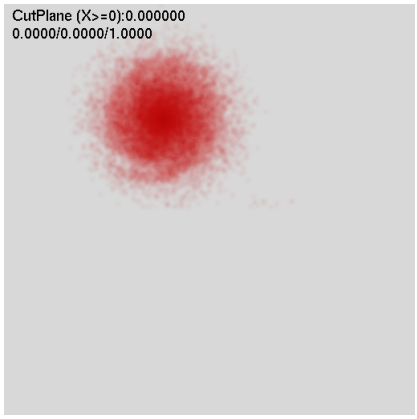
The preceding steps extract the skeletal voxels of the n-body data. A number of techniques were implemented to visualize the skeleton. The first method was to simply render the skeletal voxels as points using the `SkeletonVisualizer`. An alternative to this was to render all voxels with distance transforms above a certain threshold. In general, this was less expressive than rendering the skeletal voxels. The final method used a `SkeletonTree` function that continued the algorithm described in [9]. All of the skeletal voxels were entered into a fully connected graph with edge weights from voxel *V1* to voxel *V2* calculated according to Equation 3.1. The α adds yet another parameter to the process and it's effects are shown in Section 4.5. A minimum spanning tree of this tree was created and rendered using line segments, yielding a very pared down version of the original data. The option also exists to create a skeleton tree with voxels with distance transform values above a certain threshold.

$$EW_{V1 \leftrightarrow V2} = \alpha \times DIST(V1, V2) + (1 - \alpha) \times |DT_{V1} - DT_{V2}| \tag{3.1}$$



(a) Sample Spiegel Program

(b) Full Image



(c) CutPlane Image

Figure 3.17: A Spiegel program and the images it produces.

3.2.6 Sample Program

A sample Spiegel visualization program exercising all steps of the pipeline is shown in Figure 3.17a. Particles are extracted from the data file using the `OctreeExtractor` and classified using the `DensityClassifier` as described in Sections 3.2.1 and 3.2.3. From here, the classified octree is sent to two different `DensitySplatter`s. One simply splats the octree to an image while the other employs a `CutPlane` to filter the octree and only splat particles that have an X-coordinate ≥ 0. Finally, there is an `Orbit` utility function which sets the position of the camera to be above the galaxies looking down.

Chapter 4

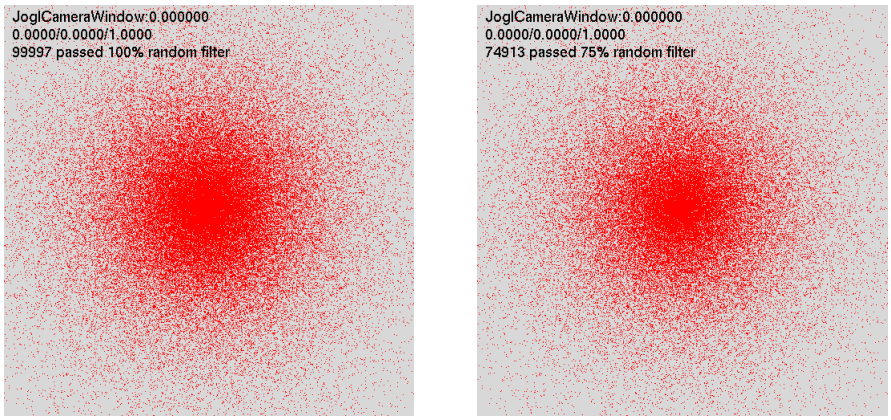
Results

4.1 Dark Matter Random Reduction

The `RandomReduction` filter was used along with the `PointVisualizer` to determine if there was an inner structure to the dark matter particles. The unreduced particles can be seen in Figure 4.1a). As more of the particles are randomly removed (Figures 4.1b, 4.1c, and 4.1d), notice that no structure emerges. This shows the fairly smooth distribution of the dark matter and indicates there is either no inner structure to the data or these visualization methods are unable to reveal it.

4.2 Rendering Method Comparison

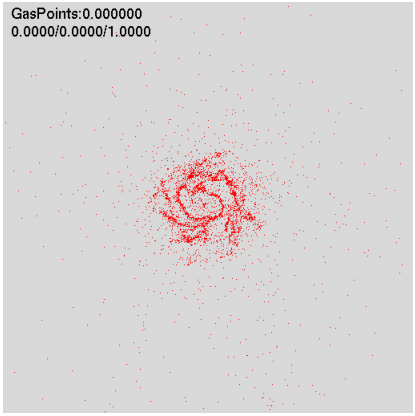
Figures 4.2a, 4.2b, 4.2c, and 4.2d show the different rendering methods applied to the gas particles of an isolated spiral galaxy. Notice how all four methods capture the inherent structure of the data. The two skeleton methods have the benefit of completely eliminating noise outside of the main body of the data. However, they both also lose some of the structure in the center of the data, in particular the absolute center, which has a increase in density that's missing in the skeleton representations.



(a) 0% reduction (b) 25% reduction

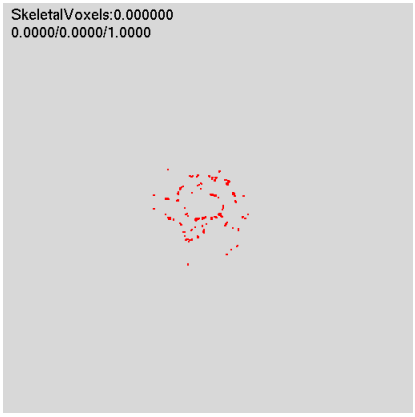
(c) 50% reduction (d) 75% reduction

Figure 4.1: Random reduction of dark matter particles.

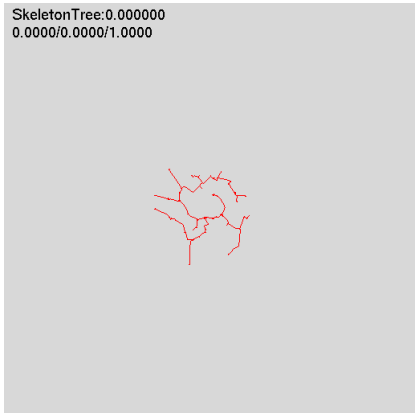


(a) Point rendering

(b) Volume density splatting



(c) Skeletal voxels



(d) Skeleton tree

Figure 4.2: Different rendering techniques applied to gas particles

4.3 Skeleton Parameters

Figures 4.3 and 4.4 illustrate the effect that the skeletonization parameters have on the final skeleton. Both sets of figures are skeletonizations of the same particle set from Figures 4.2a and 4.2b. Both connectedness and thinness parameters have a direct effect on the generated skeleton. The determination of these values is currently also a trial-and-error procedure.

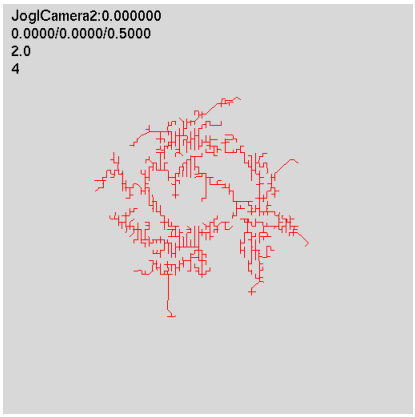
Sometimes the skeletonization parameters can hide complex features. In Figure 4.5, the resulting two arms of two colliding galaxies can be seen or hidden, depending on the thinness chosen. A similar thinness value that shows an accurate skeleton in Figure 4.4c completely hides the arms in Figure 4.5b. This highlights the highly data-dependent nature of the skeletonization algorithm.

4.4 Skeleton Tree Artifacts

The changing positions of the stars leads to very noticeable artifacts in the skeleton tree when examined over time. At any one timestep, the skeleton tree looks like a reasonable approximation of the structure of the galaxy. Over time, though, the skeleton's structure can change greatly enough that a "jumpiness" effect appears when viewed as an animation. Pairs of skeleton trees at consecutive timesteps can be seen in Figure 4.6. Note the changes to a few of a large line segments between timesteps.

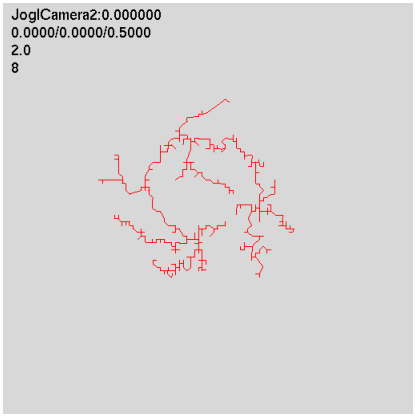
4.5 Skeleton Tree α Parameter

The α parameter determines the contribution of each of euclidean distance and distance transform to the graph from which the skeleton tree is generated. An α value close to 1 means the skeleton is generated based more upon the distance between two particles. This can be seen in Figure 4.7d. As α decreases, the skeleton becomes based more upon the difference in distance transform between two particles. This has the effect that particles from different sides of the dataset can become connected, a feature that can be seen in

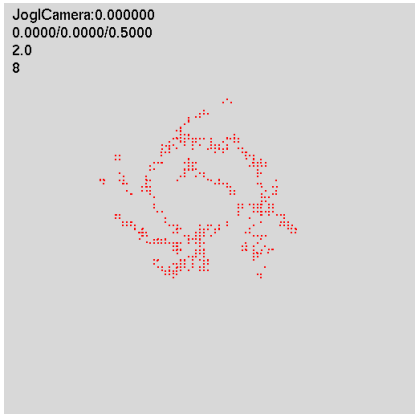


(a) Skeleton Tree, Connectedness: 4

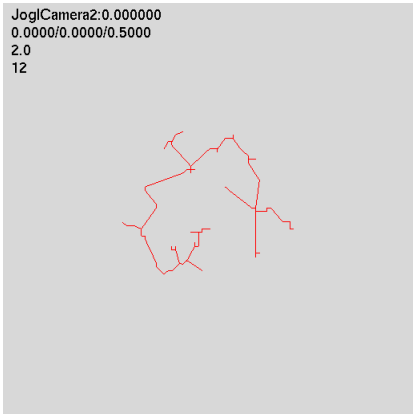
(b) Skeletal Voxels, Connectedness: 4



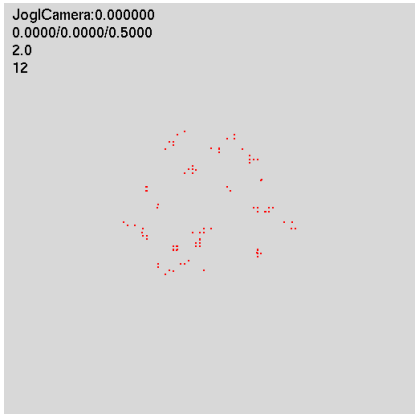
(c) Skeleton Tree, Connectedness: 8



(d) Skeletal Voxels, Connectedness: 8

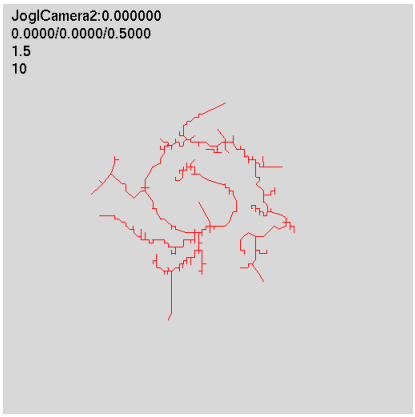


(e) Skeleton Tree, Connectedness: 12

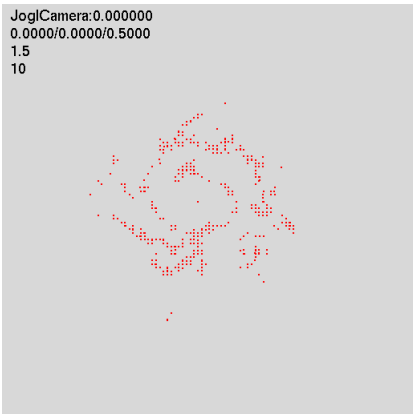


(f) Skeletal Voxels, Connectedness: 12

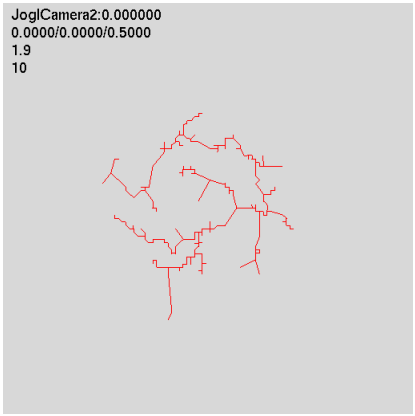
Figure 4.3: Comparison of different connectedness parameter settings for skeletal voxels and skeleton trees. Thinness: 2.0



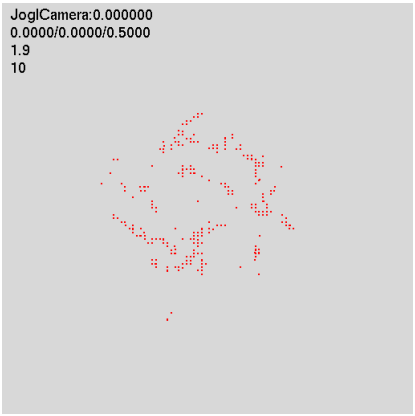
(a) Skeleton Tree, Thinness: 1.5



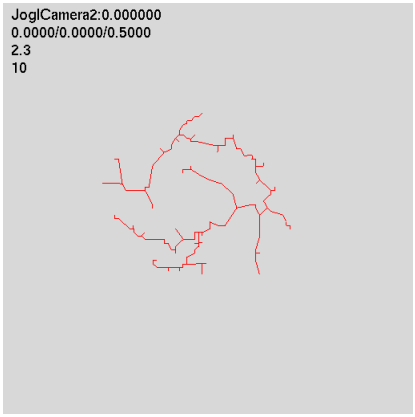
(b) Skeletal Voxels, Thinness: 1.5



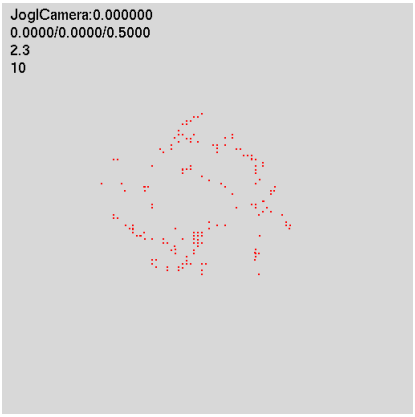
(c) Skeleton Tree, Thinness: 1.9



(d) Skeletal Voxels, Thinness: 1.9

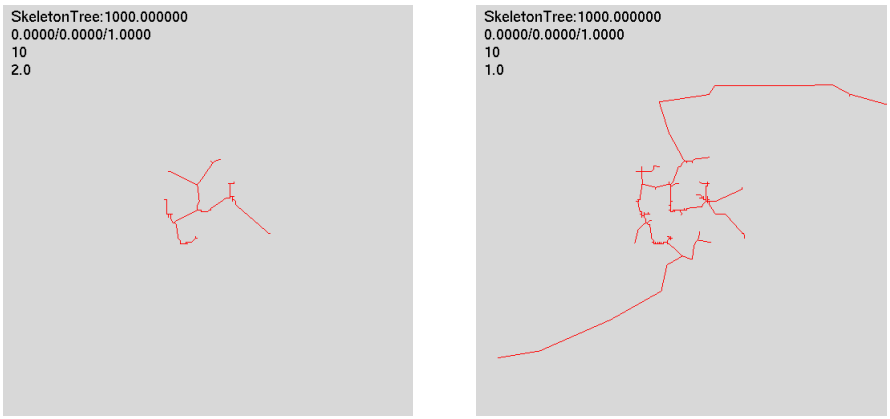


(e) Skeleton Tree, Thinness: 2.3



(f) Skeletal Voxels, Thinness: 2.3

Figure 4.4: Comparison of different thinness parameter settings for skeletal voxels and skeleton trees. Connectedness: 10



(a) Thinness: 2.0 (b) Thinness: 1.0

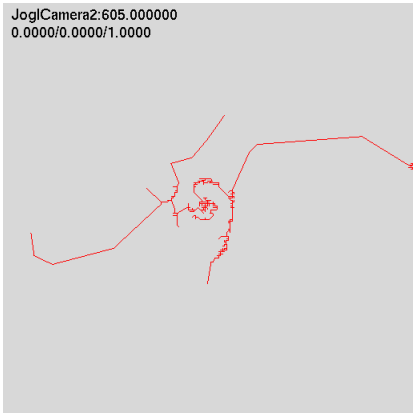
Figure 4.5: Sometimes the skeletonization parameters can hide complex features.

Figure 4.7a and which is generally undesirable because particles close to each other are more likely to be related.

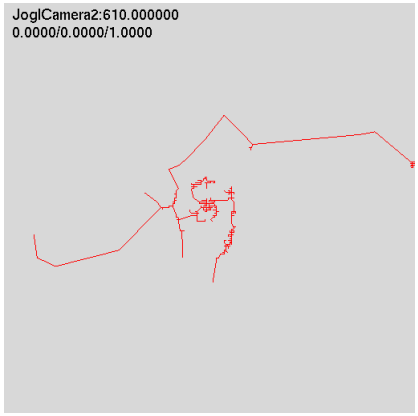
4.6 Age-Related Visualizations

The `AgeFilter` makes it possible to only visualize stars that were born at certain times of the simulation. The stars in Figures 4.8a and 4.8c are "old stars" meaning they were in existence at the beginning of the simulation. "New stars", seen in Figures 4.8b and 4.8d, were born sometime after $t = 0$. The most interesting things about the images in Figure 4.8 are the more concentrated groups of new stars, which can be interpreted to be tidal dwarf galaxies.

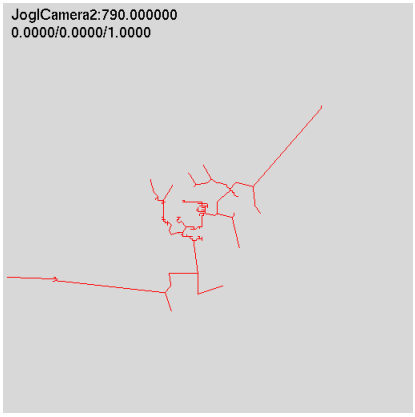
The `AgeClassifier` classifies stars according to their age or time of birth. The results of this can be seen in Figure 4.9. The scale on the side of the image shows the mapping from time of birth to color.



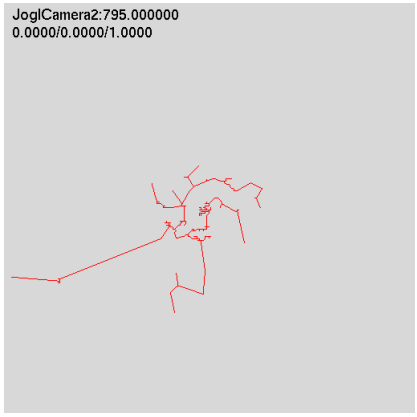
(a) Skeleton Tree, $t = 605$



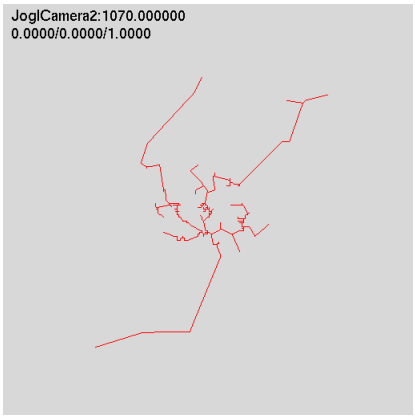
(b) Skeleton Tree, $t = 610$



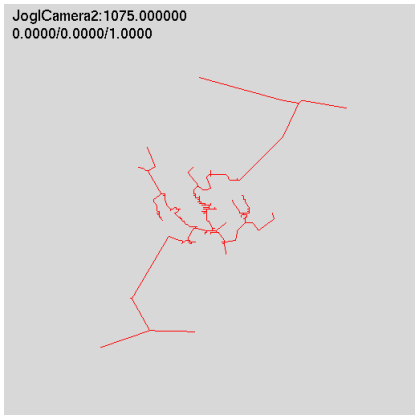
(c) Skeleton Tree, $t = 790$



(d) Skeleton Tree, $t = 795$

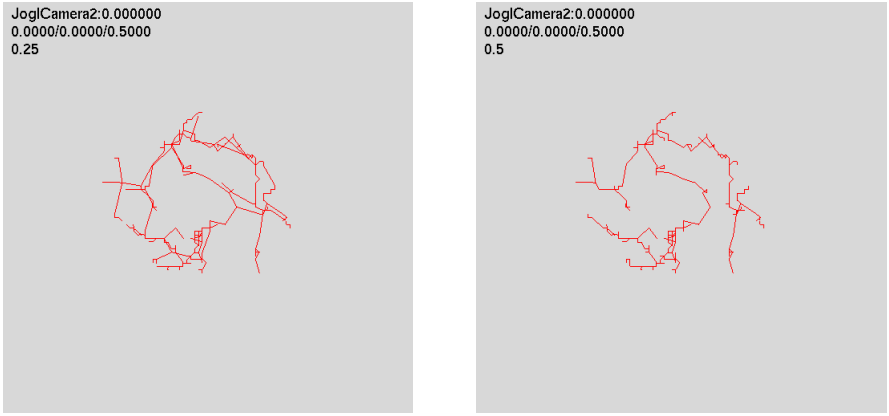


(e) Skeleton Tree, $t = 1070$



(f) Skeleton Tree, $t = 1075$

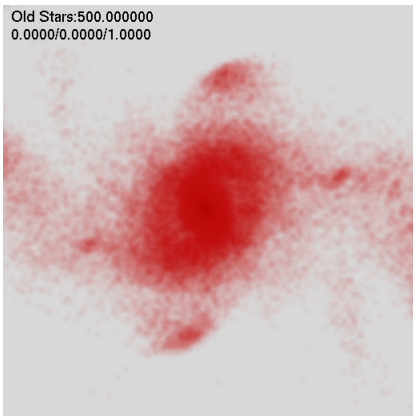
Figure 4.6: Comparison of pairwise consecutive skeleton trees



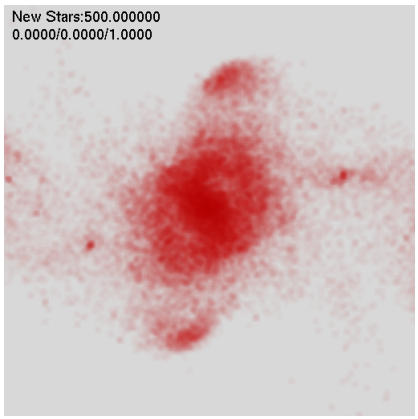
(a) $\alpha = 0.25$ (b) $\alpha = 0.50$

(c) $\alpha = 0.75$ (d) $\alpha = 1.00$

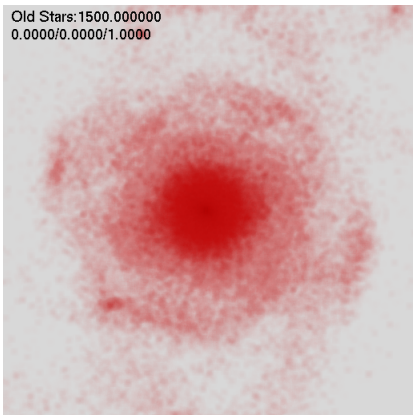
Figure 4.7: Effects of α parameter on generated skeleton tree.



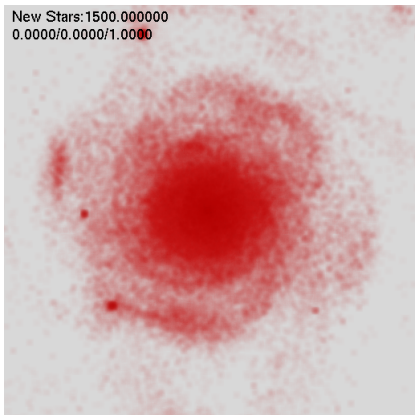
(a) Old stars at $t = 500$



(b) New stars at $t = 500$

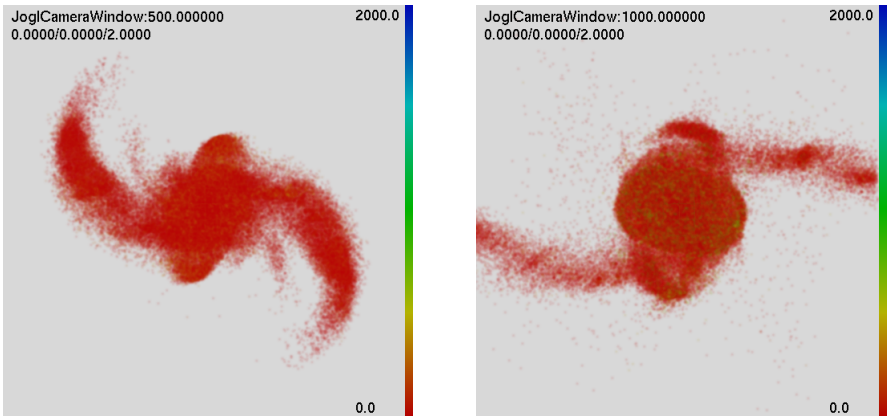


(c) Old stars at $t = 1500$



(d) New stars at $t = 1500$

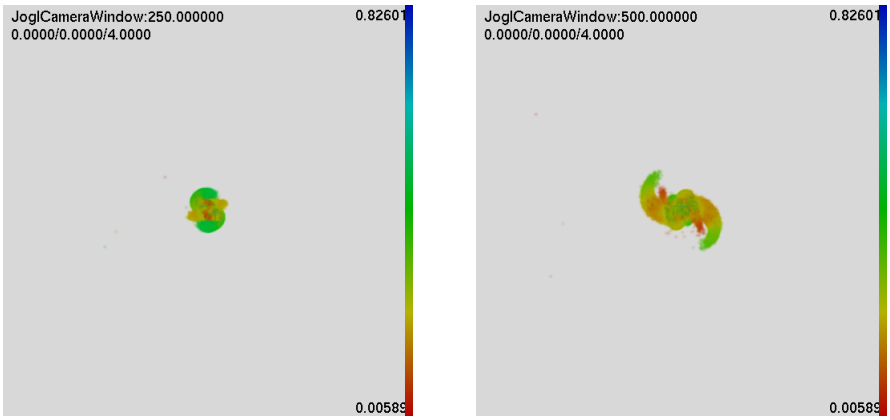
Figure 4.8: Star density for old and new stars.



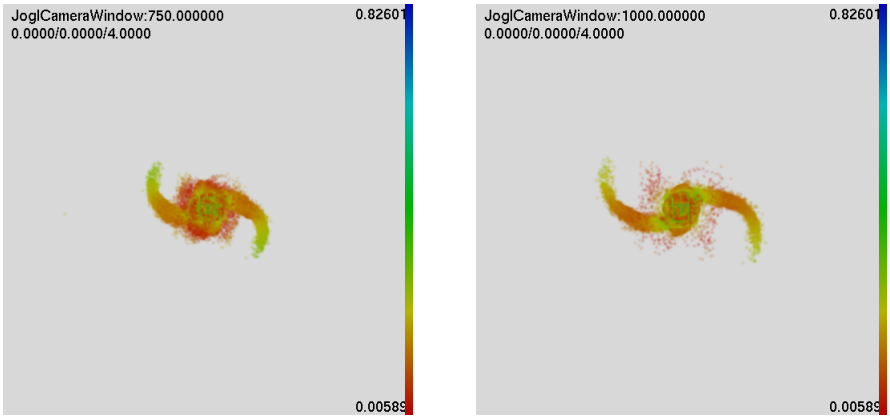
(a) $t = 500$ (b) $t = 1000$

(c) $t = 1500$ (d) $t = 2000$

Figure 4.9: Time of birth colored stars.



(a) $t = 250$ (b) $t = 500$



(c) $t = 750$ (d) $t = 1000$

Figure 4.10: Velocity of star particles over time.

4.7 Velocity

Figure 4.10 shows the velocity of star particles at various points in time. The visualized value is the magnitude of the velocity vector of the particle, normalized to [0..1].

Chapter 5

Conclusions

This work has successfully demonstrated that skeletonization is a valid technique for displaying the inner structure of n-body data. At discrete points in time, the skeleton tree is able to show a reasonable approximate of the structure of the data as verified by point and volume rendering. However, significant artifacts emerge when the skeleton tree is animated over time. Moreover, the skeleton tree has only been shown to be useful when looking at the position of particles. No attempt was made to skeletonize based on an attribute of the particles. Artifacts are also present when the particles are divided in space between multiple disconnected groups.

Skeletal voxels do not appear to be a useful visualization technique. Once the particles because disperse enough, the skeletal voxel visualization is too thin to derive any meaning. Some structure needs to be placed over these voxels, usually in the form of the skeleton tree.

Volume rendering has been shown to be best among these three techniques at showing the expected astrophysical phenomena. It's ability to capture the diffuse properties of gas and cloud particles is far superior to point rendering. It is also able to display the density of a dataset better than point rendering.

Point rendering remains a valid technique to render n-body datasets. However, its simplicity becomes apparent in comparison to more advanced techniques.

Chapter 6

Future Work

Density could be calculated using a density estimator to make it more in sync with the calculations done by astrophysicists during simulation. The density estimator would ideally work on the original particle locations instead of the voxelized ones in the octree. This would also potentially reduce any artifacts introduced by voxelization at the expense of a possibly higher time complexity.

The determination of the skeletonization parameters is currently a trial-and-error process. Automating this would take away some uncertainty. This could be accomplished by means of a machine learning algorithm with a fitness function that compares the generated skeleton to an alternate rendering, potentially the splatted density. This would be time consuming, but would only have to be performed once to get a good base set of parameters that could be tuned.

The skeletonization algorithm generates a potentially undesirable artifact when the data is divided into more than one physical group, such as the case with two galaxies before collision. The skeleton tree generated from this data contains a skeleton for each galaxy and a line connecting them. Ideally, this line could be removed algorithmically. This would also improve the skeleton generated from later time steps where galaxies have formed in the wake of the two colliding galaxies. Additionally, steps need to be made to reduce the artifacts that emerge when the skeleton tree is viewed over time. A possible solution would be to have the skeletonization algorithm take the skeleton tree of the previous timestep into account.

The generated skeleton tree could be used as input to other functions in the system. One such function would move the camera along the skeleton, effectively riding through the heart of the galaxy. Also, the skeleton could be used to accelerate the splatting process by only splatting voxels near the skeleton.

Classification is also a trial-and-error process. This could either be simplified or potentially automated. Simplification could be done by implementing a GUI widget to map input values to RGBA. In this way, a single classifier could be created that can create classification profiles for different dimensions of a dataset. This would be much more intuitive than the current system of hardcoding a function with a couple parameters. To automate the classification process would mean applying some of the automatic classification work discussed in Section 2.3 to n-body data.

Finally, velocity calculation, as discussed in Section 4.7, is done in a very simple way currently. A more useful astrophysical value would be the sum of mass-weighted velocities project along the line of sight.

Bibliography

[1] J. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature*, 324:446–449, December 1986.

[2] T.A.DeFanti B.H.McCormick and M.D.Brown. Visualization in scientific computing. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1987. ACM Press.

[3] S. Casertano and P. Hut. Core radius and density measurements in N-body experiments Connections with theoretical and observational definitions. *The Astrophysical Journal*, 298:80–94, November 1985.

[4] Edward Dale. Adding jogl to spiegel. Honors Capstone Project, May 2006.

[5] W. Dehnen. A Hierarchical $\mathcal{O}(N)$ Force Calculation Algorithm. *Journal of Computational Physics*, 179:27–42, June 2002.

[6] T. Todd Elvins. A survey of algorithms for volume visualization. *SIGGRAPH Comput. Graph.*, 26(3):194–201, 1992.

[7] Shiaofen Fang, Tom Biddlecome, and Mihran Tuceryan. Image-based transfer function design for data exploration in volume visualization. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 319–326, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[8] S. Frisken and R. Perry. Simple and efficient traversal methods for quadtrees and octrees, 2002.

[9] N. Gagvani and D. Silver. Parameter controlled skeletonization of three dimensional objects, 1997.

[10] Nikhil Gagvani, D. Kenchammana-Hosekote, and D. Silver. Volume animation using the skeleton tree. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 47–53, New York, NY, USA, 1998. ACM Press.

[11] Stefan Harfst, Christian Theis, and Gerhard Hensler. Modelling galaxies with a 3d multi-phase ism, 2005.

[12] William M. Hsu. Segmented ray casting for data parallel volume rendering. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, pages 7–14, New York, NY, USA, 1993. ACM Press.

[13] Forschungszentrum Jülich. Xnbody: an online visualization tool for nbody6++. `http://www.fz-juelich.de/zam/xnbody/`.

[14] Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 79–86, New York, NY, USA, 1998. ACM Press.

[15] Inc. Kitware. The visualization toolkit. `http://www.vtk.org/`.

[16] Jean Kovalevsky. *Introduction to celestial mechanics*, volume 7 of *Astrophysics and space science library*. Springer-Verlag, New York, 1967. Translated by Express Translation Service; 25 cm; Bibliography: p. [127]; Translation of Introduction la mecanique celeste.

[17] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Lawrence Livermore National Laboratory. VisIt. `http://www.llnl.gov/visit`.

[19] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[20] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, New York, NY, USA, 2000. ACM Press.

[21] Rochester Institute of Technology. The GRAPE cluster Project. `http://www.cs.rit.edu/~grapecluster/`.

[22] Hanspeter Pfister, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The volumepro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 251–260, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[23] Hanan Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

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

[25] Daqing Xue and Roger Crawfis. Efficient splatting using modern graphics hardware. *journal of graphics tools*, 8(3):1–21, 2003.

Appendix A

CD Contents

spiegel.jar This is the executable for the Spiegel visualization framework. It contains all of the functions described in the text above as well as the utility functions that Spiegel provides.

src The source tree used to create the executable above. To rebuild `spiegel.jar`, Apache Ant[1] is needed and must be run from the `src/grapecluster` directory. This will create `spiegel.jar` in the `src/grapecluster/dist` directory, which is the same as the file provided.

data A collection of data files used to create the images above. The `isolated` directory contains a single spiral galaxy developing over time. The `interact` directory contains two spiral galaxies that collide and interact over time.

images Full-resolution versions of the images used above.

movies A collection of movies demonstrating generated from the images from which the images above were taken.

thesis Documents relating to this thesis including the preproposal, proposal, thesis report, and defense presentation. `usersguide.pdf` is a guide to using the system and creating images similar to the ones found here.

[1] `http://ant.apache.org`