2006

Study of visualizations for n-body stellar models

Christian Gray

M.S. Computer Science Project

Rochester Institute of Technology

Rochester, New York

Study of Visualizations for N-Body Stellar Models

By

Christian Gray

Faculty Chair: Professor Hans-Peter Bischof

Signature: _____ Date: _____

Faculty Reader: Professor Joe Geigel

Signature: _____ Date: _____

Faculty Observer: Professor Sidney Marshall

Signature: _____ Date: _____

Abstract
The visualization of large datasets is invaluable to the advancement of scientific knowledge. N-Body problems produced large dataset that present several challenges to visualization. These challenges include over-plotting and self-obscuring data. Using the general visualization framework, Spiegel, five different visualization techniques are explored to solve these problems: slicing, projection, 3D projection, 3D density shells, and direct volume rendering. Each visualization looks to solve the problem in a slightly different manner while all have kernel density estimation as their starting point. While none of the visualization is ground breaking or perfectly solves the problems they do provide new insight allowing for new knowledge of the data being studied.

Abstract

Table Of Contents

References

1 Introduction

1.1 Problem Overview

Stellar modeling on the galactic level involves calculation of the *n*-body problem. The *n*-body problem uses Newton's laws of motion and gravity to calculate the motions of all the objects in the system given their initial positions, masses, and velocities. The result of the millions of calculations is a massive amount of data. To make use of these datasets they must be effectively visualized so that sense can be made from the raw numbers. Visualization also allows the observer to see information that is not normally visible. For stellar models this could be information such as gravity. Some challenges exist with the visualization of a truly three-dimensional dataset of this size.

Traditional methods of displaying datasets of this size suffer from over-plotting. Over-plotting occurs when the number of data points contained in one area exceeds our ability to discern or the medium's ability to display individual points. Graphs such as a scatter-plot become single blobs, losing all detail and structure of the data.

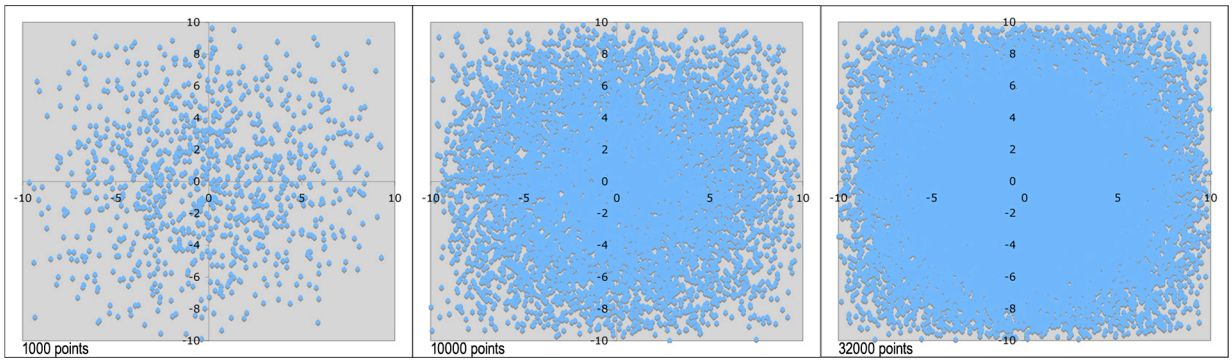


Figure 1: Affects of over-plotting as dataset size increases

Another problem stems from the three-dimensional nature of the data. Since we can only see in 3D, data points can potentially block other data points.

This paper attempts to take some known approaches to visualize large datasets and apply them to the *n*-body problem. The approaches chosen were slicing, projection, projection contours, iso-surface extraction and volumetric rendering.

The slicing and projection methods were chosen because they have been successful in the visualization of medical data. Iso-surface extraction was chosen for its ability to extract 3D "slices" based on a values other then position. The final method, volumetric rendering, was chosen because it had the potential to provide the most complete picture since it reduces the amount of data filtered out.

1.2 Objectives

The primary objectives of this project are as follows:

- Determine problems involved with visualizing large datasets
- Experiment with ways to visualize these datasets in a meaningful way
- Try to overcome the problems identified earlier in the visualizations
- Compare results to determine how successful the visualization was

Problems identified:

- Over-plotting
- Self-obscuring (data hides other data due to 3D nature)

2 Architecture

2.1 Visualization Frameworks

Visualization frameworks attempt to provide a structured environment for transforming data into visual images that are comprehensible to a human observer. These systems are usually built for scientific analysis of large datasets were conventional analysis may not be able to give an insight into the subtle interactions of the data. Visualization of non-observable data is also possible such as density, temperature, and pressure.

2.2 Spiegel

Spiegel is a general-purpose visualization framework. Spiegel's main advantages are its extensibility and pipeline architecture modeled after the Unix shell.

Spiegel uses a plug-in architecture. This architecture allows plug-ins to be written to expand the system to meet new needs. This allows the system to accommodate a wide range of systems.

Spiegel's pipeline system allows for each plug-in to be specialized and reusable, much like a Unix command. Each plug-in in the system allows for typed input and output. Inputs and outputs of the same type can be piped together to form a chain. Along with simple chains Spiegel allows for branching a single output to multiple inputs and in some cases the merging of outputs to a single input.

The Spiegel system uses a GUI to allow users to select plug-ins from different menus, link them together, and manipulate the various parameters for each plug-in. This allows someone with little programming skill to use the system. The plug-ins are broken into different classes that have a general concept and are grouped under those classes in the menus.

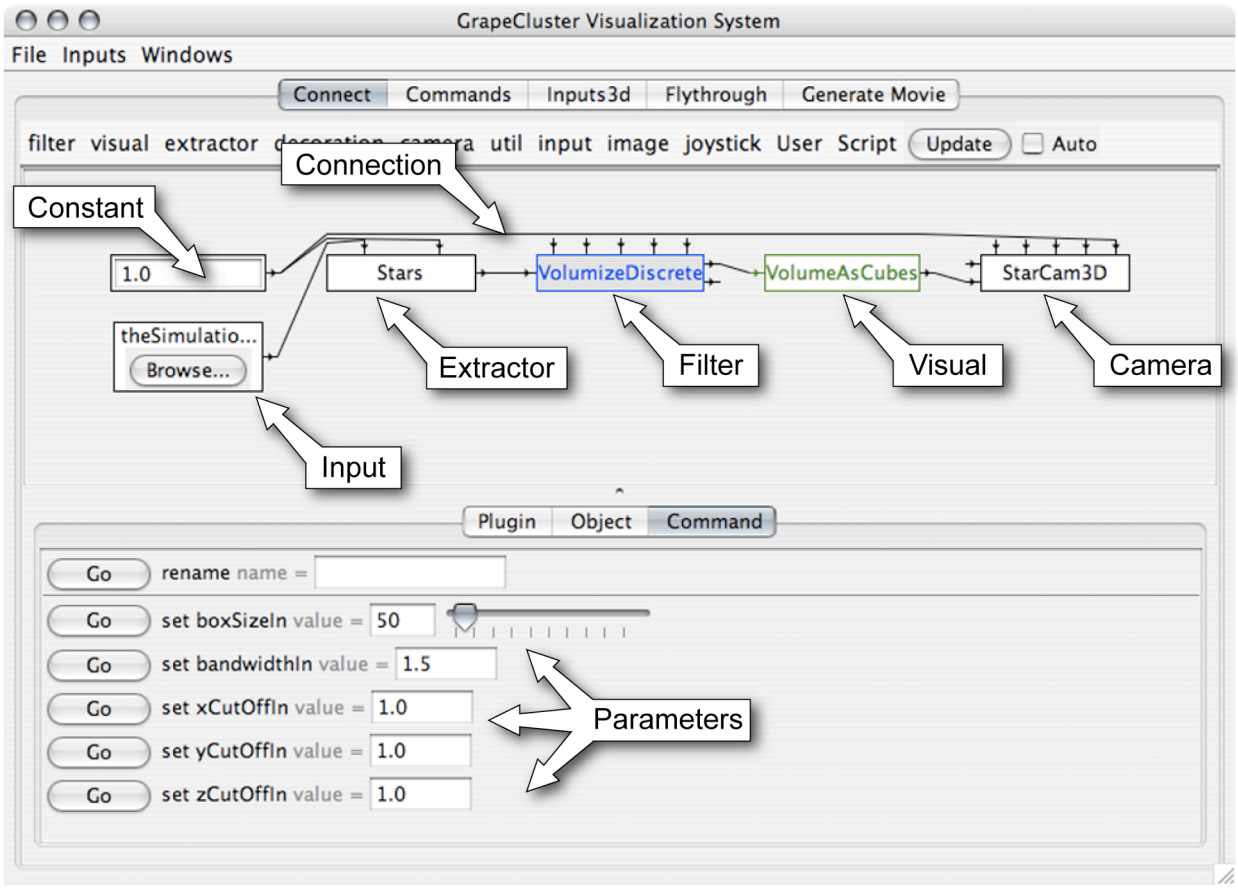


Figure 2: Spiegel GUI showing various plug-ins and their connections forming a visualization.

3 Visualizations

All my visualizations have the same basic steps involved. First, the input space is partitioned into a three dimensional grid. Then kernel density estimation is used to calculate the density at the grid points. The density is then mapped to visual elements in the visualization.

3.1 Density Estimation

3.1.1 VolumizeDiscrete Spiegel Plug-in

The job of the VolumizeDiscrete plug-in is to partition the input space and produce the volume density map. The VolumizeDiscrete plug-in has five parameters: boxSizeIn, bandwidthIn, xCutOffIn, yCutOffIn, zCutOffIn. xCutOffIn, yCutOffIn, and zCutOffIn set the boundaries of the visualization space that will be partitioned. BoxSizeIn and bandwidthIn have the most affect on the quality of our final image. VolumizeDiscrete expects stars as input and outputs a volume.

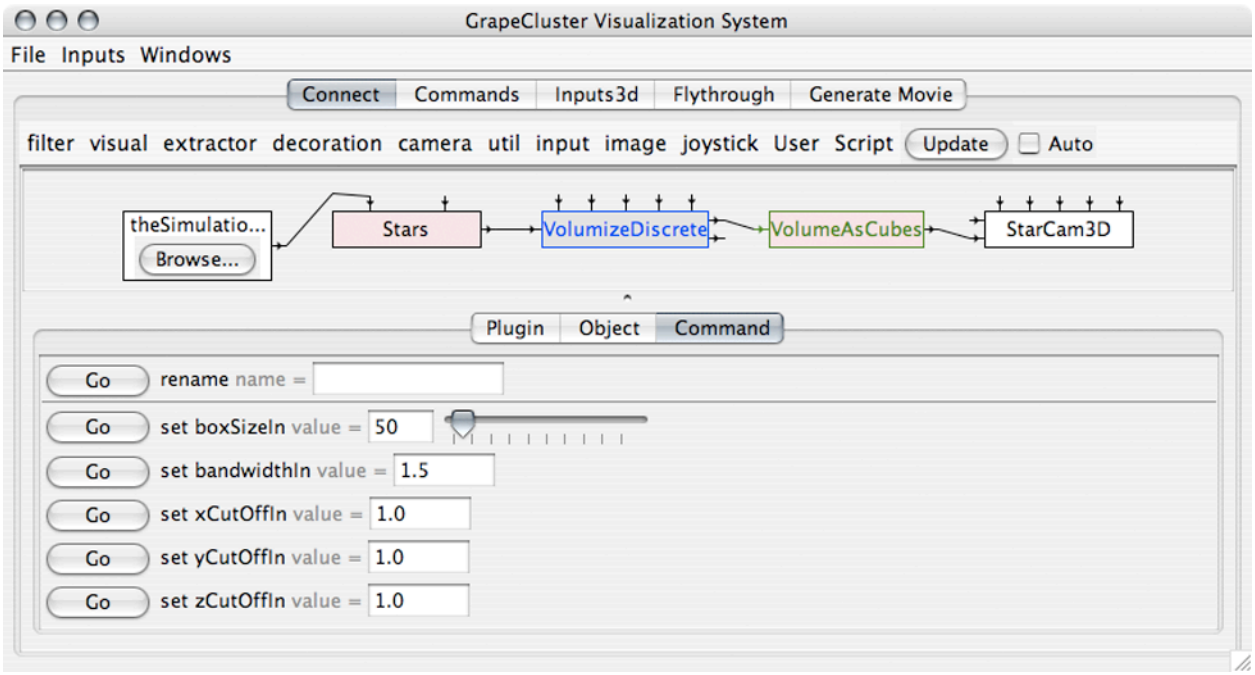


Figure 3: Visualization utilizing VolumizeDiscrete plug-in

Box size (boxSizeIn) refers to the size of the cubes that the space will be divided into. Input is 1 – 1000, representing visualization space units of 0.001 – 1. The smaller the cube the more detail in the resulting volume density map.

The bandwidth to be used in our kernel density estimator is a combination of boxSizeIn and bandwidthIn. The actual bandwidth is the multiplication of boxSizeIn and bandwidthIn (.05 * 1.5 = 0.075). Since the kernel is used to smooth out the visualization bandwithIn should in most case be greater then 1, the default is 1.5.

The following figure shows the affects of boxSizeIn and bandwidthIn:

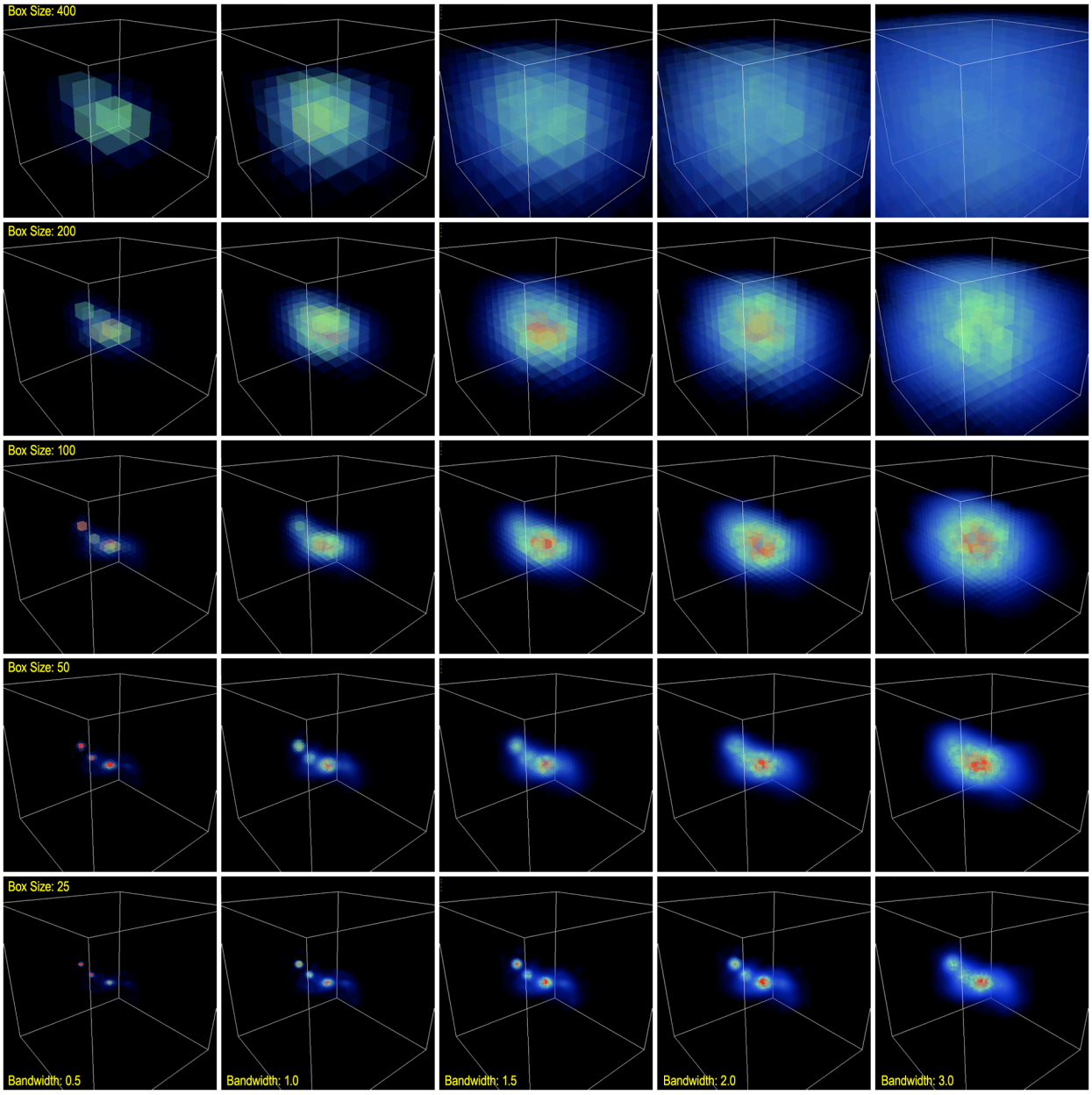


Figure 4: Affects of bandwidth and box size selection

3.1.2 Continuous Estimation

To overcome the effects of over-plotting and to transform the dataset into a volume dataset I will use a density map. The density map will be created using a kernel density estimator. Density maps allow the underlying structure of the data to be maintained though individual points are still lost. A kernel density estimator performs a local averaging on a dataset to smooth out and fill in missing data. The formula for a kernel density estimator is given as follows [9]:

$$\hat{f}(x;h) = (nh)^{-1} \sum_{i=1}^{n} K\left\{ \frac{x - X_i}{h} \right\}.$$

Equation 1: kernel density estimator

K is the kernel, it is a unimodal probability density function, satisfying $\int K(x)\, dx = 1$. Bandwidth is represented by h. Bandwidth determines the amount of smoothing that takes place.
A small bandwidth produces a density estimation with a large amount of variance. A large bandwidth can over smooth the data causing a loss in structure. A good bandwidth maintains the structure of the data while smoothing out the local variance in the data.

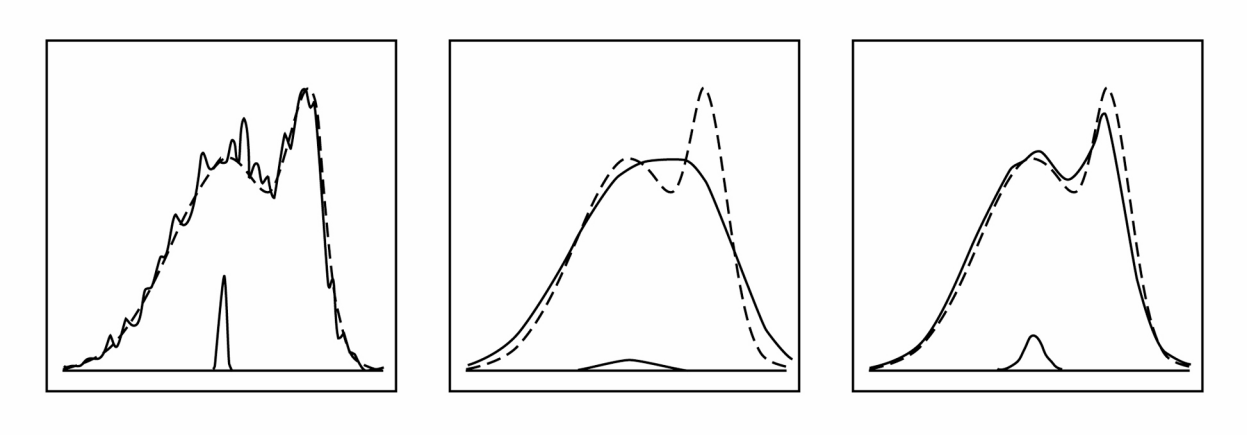


Figure 5: Affects of bandwidth selection

The choice of bandwidth is perhaps the single most important factor in the kernel density estimation, in my case the choice will be largely subjective and adjusted primarily on the quality of the image produced rather then how well it reproduces some underlying function.

The choice of the function for K is secondary though related to the bandwidth selection. Any function that satisfies the specifications above can be used; for this project I used the normal density function [9]:

$$f(x) = (2\pi)^{-1/2} e^{-x^2/2}$$

Equation 2: normal density function

This function gives the standard bell shaped curve centered at 0. Substituting equation 2 into equation 1 for K gives:

$$\hat{f}(x;h) = (nh)^{-1} \sum_{i=1}^{n} (2\pi)^{-1/2} e^{-\left(\frac{x-X_i}{h}\right)^2/2}$$

Equation 3

Since we are interested in the density of the number of stars in the neighborhood of some point P_a (x_a, y_a, z_a) we can substitute the distance between the point of interest and the positions of the stars for $x - X_i$,

$$\hat{f}(x;h) = (nh)^{-1} \sum_{i=1}^{n} (2\pi)^{-1/2} e^{-\left(\frac{\sqrt{(x_a-x_i)^2+(y_a-y_i)^2+(z_a-z_i)^2}}{h}\right)^2/2}$$

Equation 4

The bandwidth determines how far away we want to look for determining the density at P_a (x_a, y_a, z_a). The result from the kernel density should be a value between 0 and 1.

3.1.3 Discrete Estimation

The continuous estimation described above while accurate is very CPU intensive. An effort was made to reduce the number of individual calculations. Since the density map is being calculated upon a three dimensional grid I reduce the number of calculations by treating all stars in a cube of the grid as being located at the center point of the cube. Equation 4 can then be replaced with the following,

$$\hat{f}(x;h) = (nh)^{-1} \sum_{i=1}^{c} \left((2\pi)^{-1/2} e^{-\left(\frac{\sqrt{(x_a-x_c)^2+(y_a-y_c)^2+(z_a-z_c)^2}}{h}\right)^2/2} \right) \times c_{stars}$$

Equation 5

where c is the number of cubes, P_c (x_c, y_c, z_c) is the center point of the cube and c_{stars} is the number of stars in the cube.

This improvement allowed me to reduce the lower limit of cube size on my PowerBook G4 from 50 to 25, and renders at a cube size of 50 were approximately 2x as fast. Because the number of stars is in most cases many times larger then the number of cubes the space is divided in, the number of calculations involved is reduced. The change allows for further optimizations by checking for the number of stars in a cube before calculating the summation, which would be 0 if

c_{stars} were 0. We also improve memory requirements by only storing data about cubes that have stars and the locations of its surrounding neighbors.

3.2 Projection and Slicing

Medical imaging inspired the slicing method of visualizing the stellar image data. The medical field uses CAT scans and MRIs to view the human body. In these procedures a volume data set is produced by the machinery and images are produced of slices through that data set the represents the area scanned. I thought that I could apply this technique to the volume density dataset of the stellar image to try and extract detail images of the internal workings of the data.

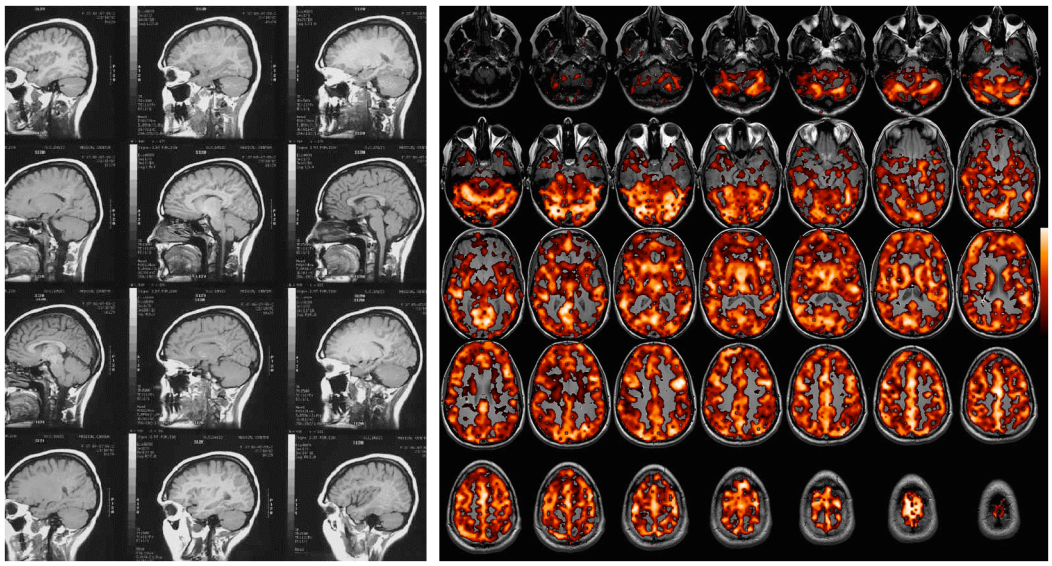


Figure 6: CT Scan left, MRI Right

3.2.1 VolumePlane Spiegel Plug-in

The VolumePlane plug-in produces all images described in this section. It job is to calculate where the display plane will be located, calculate the cubes that will be used at each patch of the plane and map that data to be displayed. The plug-in expects a volume as input, outputs a Java3D BranchGroup, and has eight parameters: distanceIn, planeIn, deltaIn, projectionIn, contourProjectionIn, meshIn, cullIn, maxContourHeightIn.

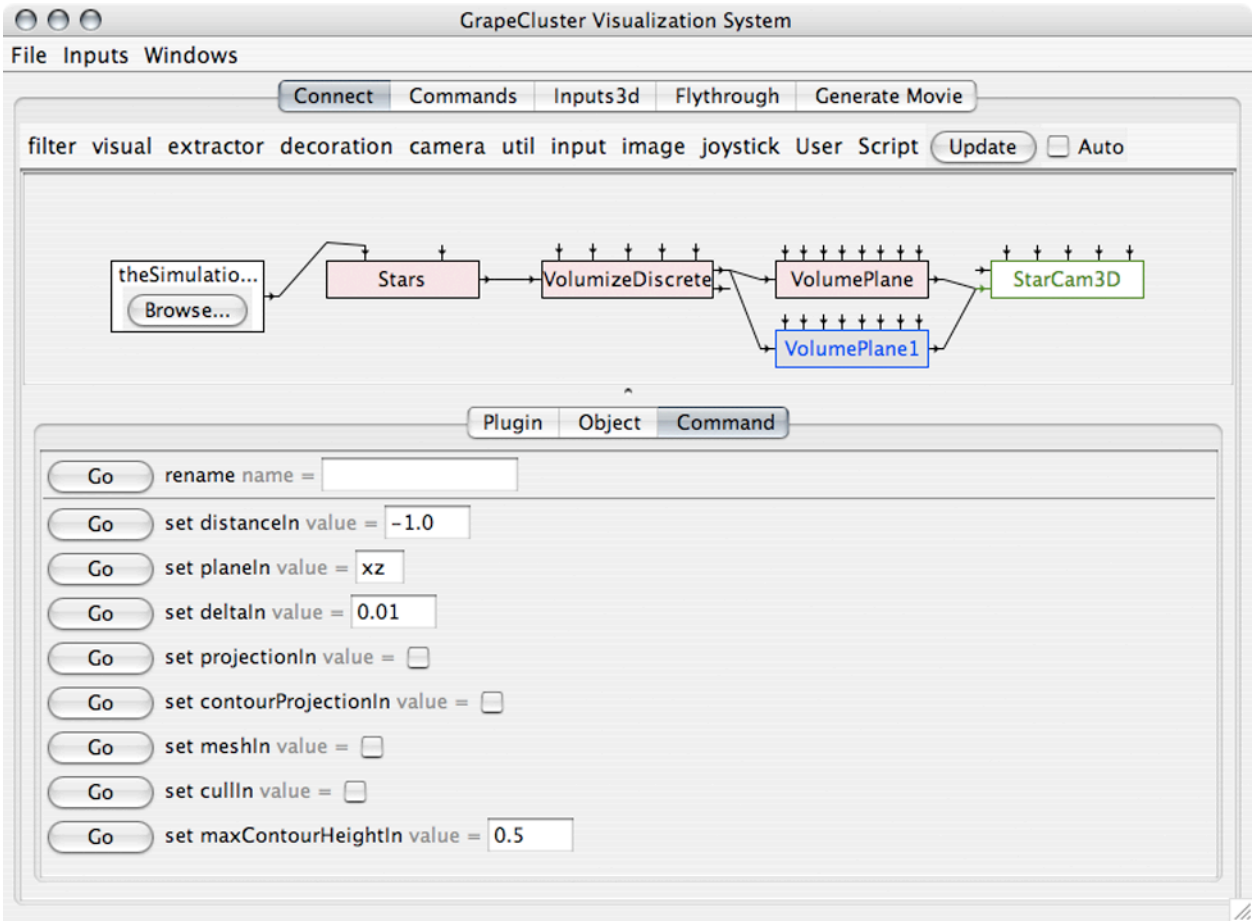


Figure 7: VolumePlane plug-in in a Spiegel visualization

The distanceIn textbox set the distance the given plane will be place from the center on the axis perpendicular from the plane.

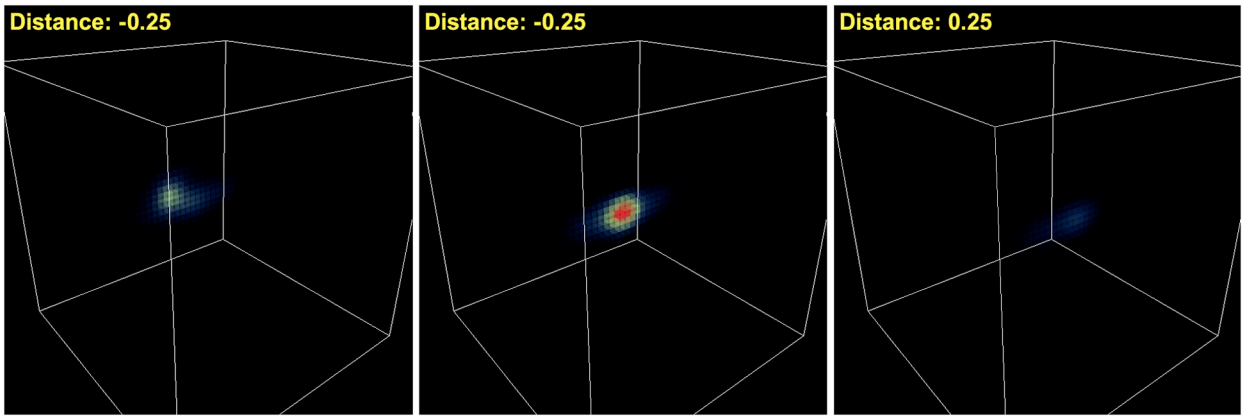


Figure 8: Affects of distanceIn textbox

The planeIn textbox currently takes the values xy, yz, and xz, which denotes the plane to use for rendering.

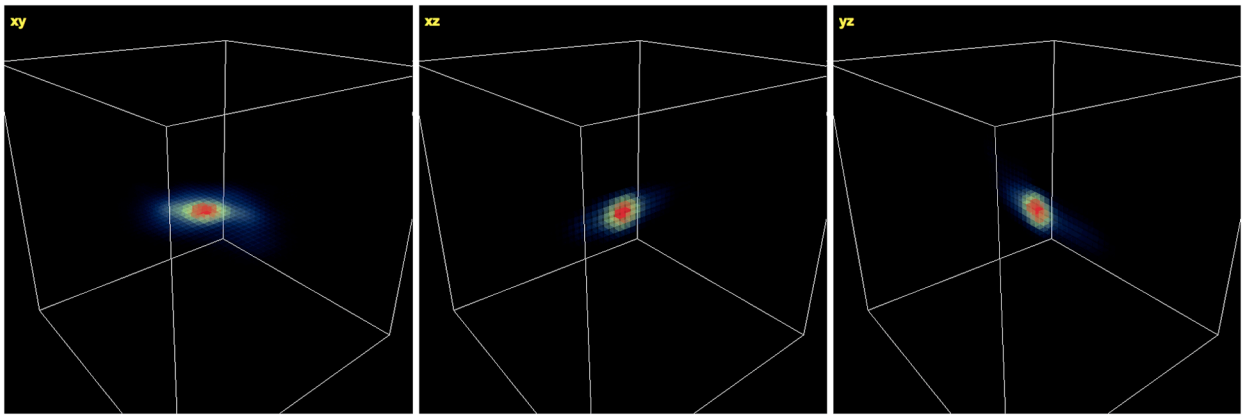


Figure 9: Changing the planeIn textbox (xy, xz, yz)

The textbox deltaIn determines the thickness of the slice to be looked at.

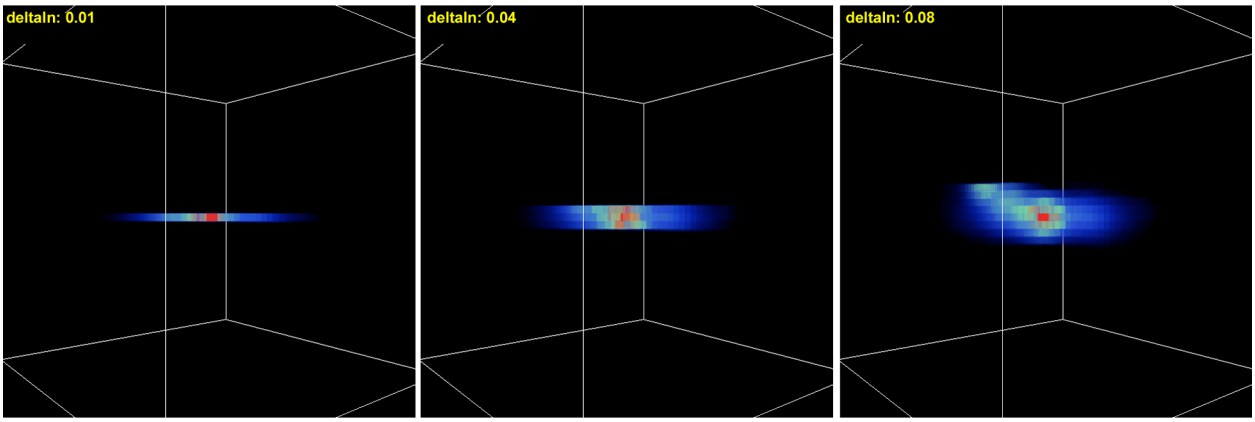


Figure 10: Changing deltaIn

If the projectionIn checkbox is set a projection of the volume is rendered on the plane. The contourProjectionIn checkbox renders a projection as a 3D contour with the plane being density of 0 and each patch having a height proportional to its density. The meshIn check box is combined with the contourProjectionIn checkbox and achieves the same result except that the patches are now rendered in wire frame.

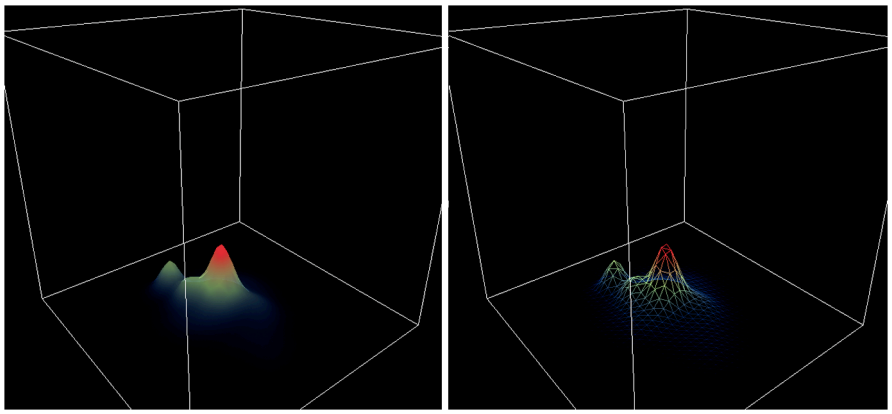


Figure 11: contourProjectionIn selected and with meshIn selected.

If cullIn is checked the back faces of the contour are excluded; this is useful in wire frame mode were the back faces can make it clutter up the image.

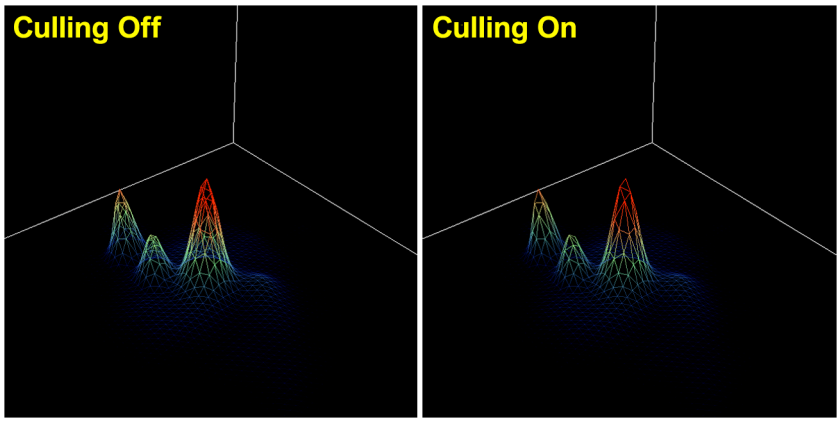


Figure 12: cullIn on and off

The final parameter maxContourHeightIn controls the height of the 3D contours with a density of 0 being the plane position and a density of 1.0 being this distance from the plane.

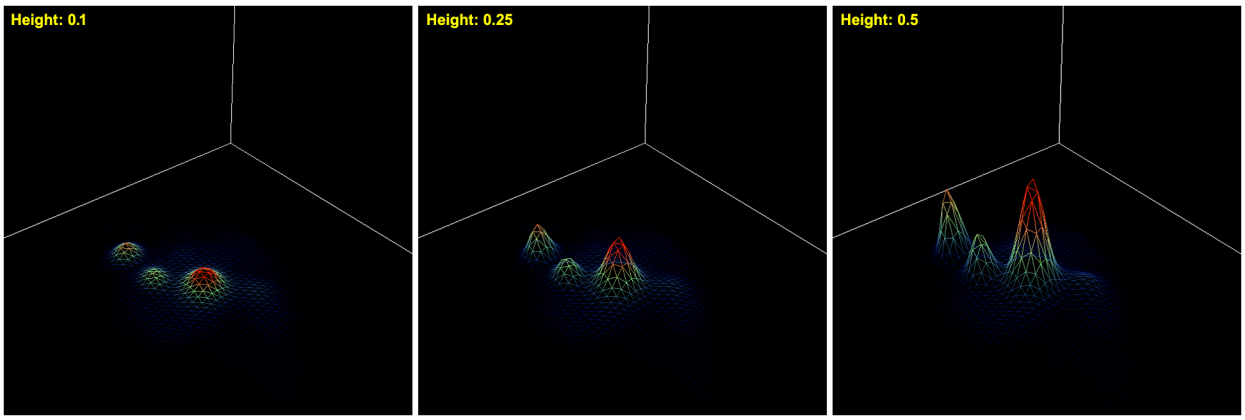


Figure 13: Affects of height on projection contours

3.2.2 Slicing

The first method for display of the stellar data is the use of slices. These slices are planer cross sections of the volume data.

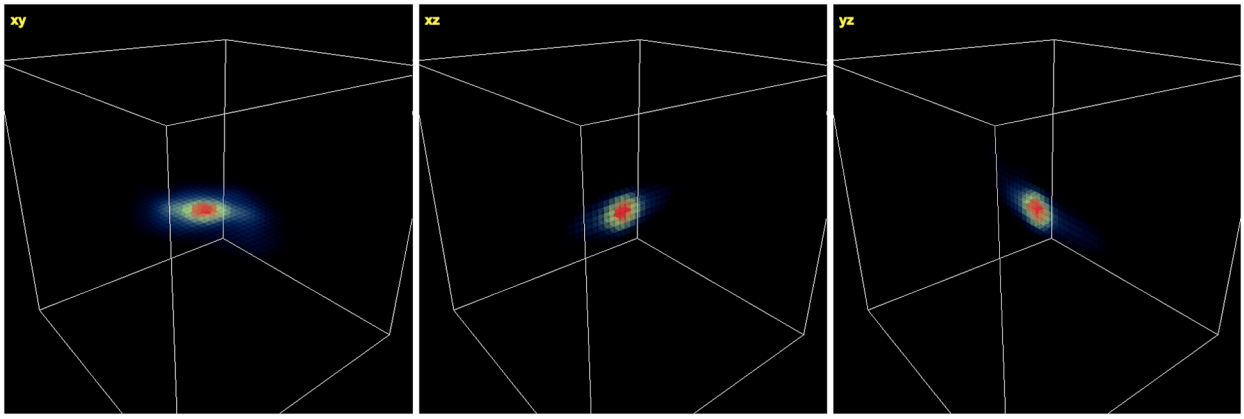


Figure 14: Spiegel VolumePlane plug-in slice renders

3.2.2.1 Slicing Algorithm

3.2.2.1.1 Define Plane

The first step in the algorithm is to define the plane we are interested in. A plane can be defined by a point $P_a(x_a, y_a, z_a)$ and a normal vector N(A, B, C) that is perpendicular to the plane.

3.2.2.1.2 Determine Data on the Plane

All data points from the kernel density estimation some threshold *t* from the plane will be used for rendering. Given a data point $P_b(x_b, y_b, z_b)$ and the plane given above we calculate the minimum distance D from the plane and a point as follows:

$$D = \frac{A(x_b - x_a) + B(y_b - y_a) + C(z_b - z_a)}{\sqrt{A^2 + B^2 + C^2}}$$

Equation 6: distance from plane to a point

3.2.2.1.3 Map Values

Data from the kernel density estimation will contain values ranging from 0-1. In this step those values are mapped to a color and alpha value for display.

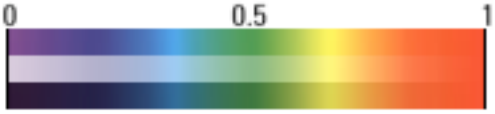


Figure 15: density to color map

3.2.2.1.4 Render Plane

The final step is to render the plane using the value at each intersecting point and looking up the color to be rendered in the color map. Multiple planer slices can be rendered to produce a more three-dimensional feel for the data.

3.2.3 Projection

Projection was added as an extension to the slicing method because I thought it could be useful in combination with other display methods. Projection works by first sub-dividing a plane into patches. A line is then drawn from each patch perpendicular to the plane. The densest cube is then selected from the cubes in the volume density map that intersect the line to represent the patch. The values are then mapped and rendered as they were in the slicing algorithm above

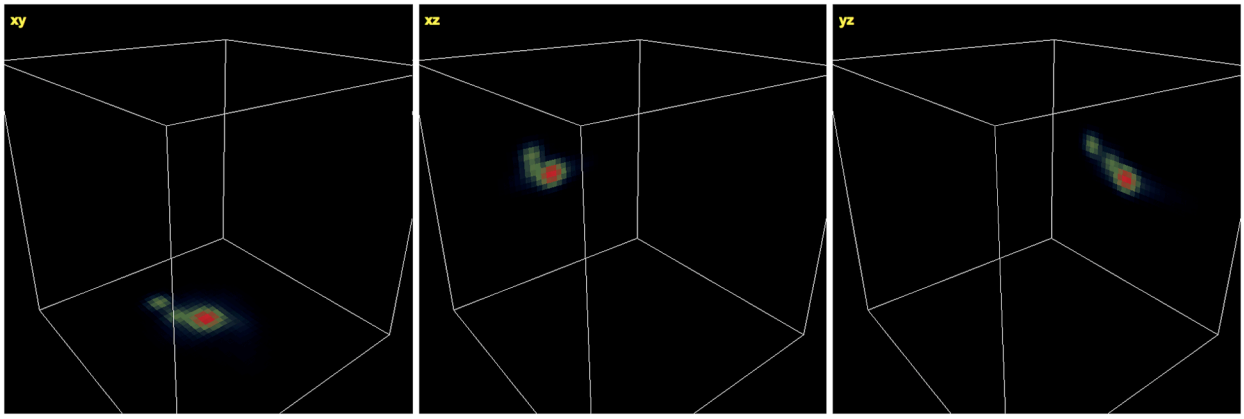


Figure 16: Spiegel VolumePlane plug-in projection renders

3.2.4 Projection Contour/Mesh

So far the visualizations techniques have taken a volume and reduced it to a two-dimensional plane. In the projection mesh visualization I take the projection of the volume and add a third dimension. Along with the color representing the density I add a height value for the density as well.

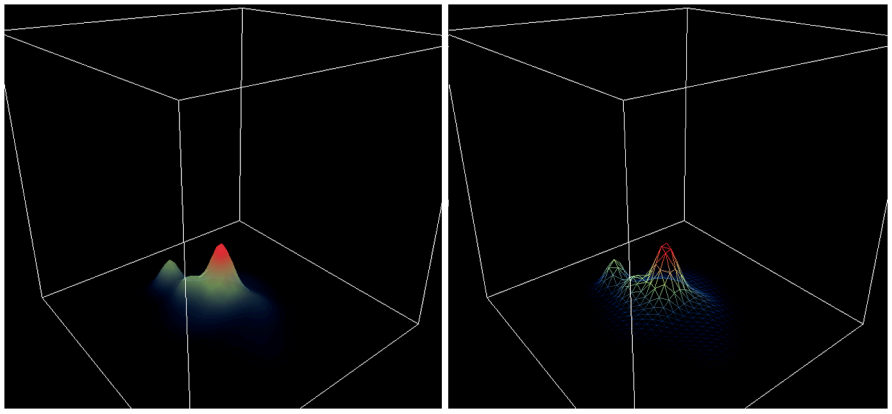


Figure 17: Spiegel VolumePlane plug-in projection contour and mesh renders

3.3 3D Shells

In this visualization I was trying to bring a traditional 2D contour map such as a topographic map into 3D space, so instead of concentric outlines I have concentric shells. This visualization extracts polygon surfaces representing a given density. Everything on one side of the surface would be of higher density and on the other would be of lower density then the selected density. In most cases this would give nested shells if more then one density were represented at a time. To extract the polygon surface I use an algorithm called Marching Cubes.

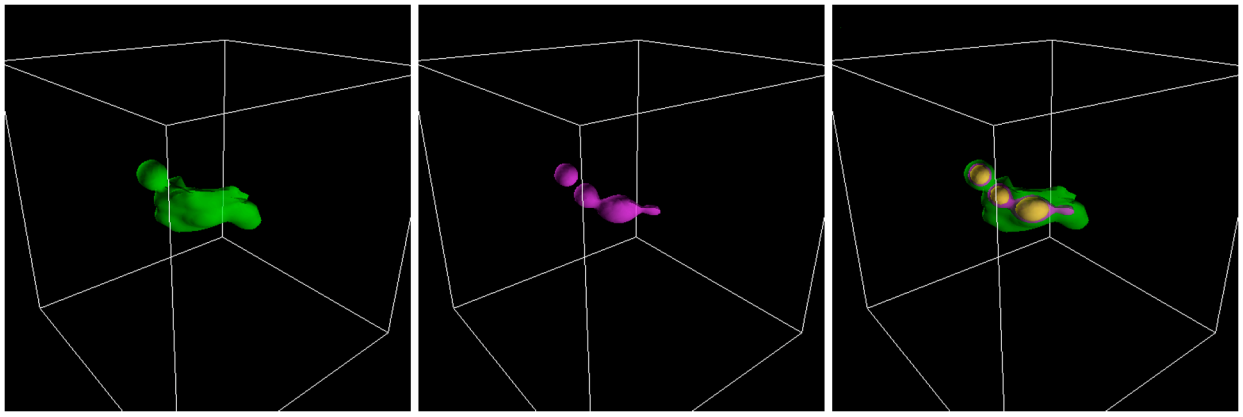


Figure 18: 3D Contour Shells

3.3.1 VolumeContour Spiegel Plug-in

The VolumeContour plug-in implements the Marching Cubes Algorithm. The VolumeContour plug-in expects a volume as input, outputs a Java3D BranchGroup, and has three parameters: densityIn, colorIn, and alphaIn.

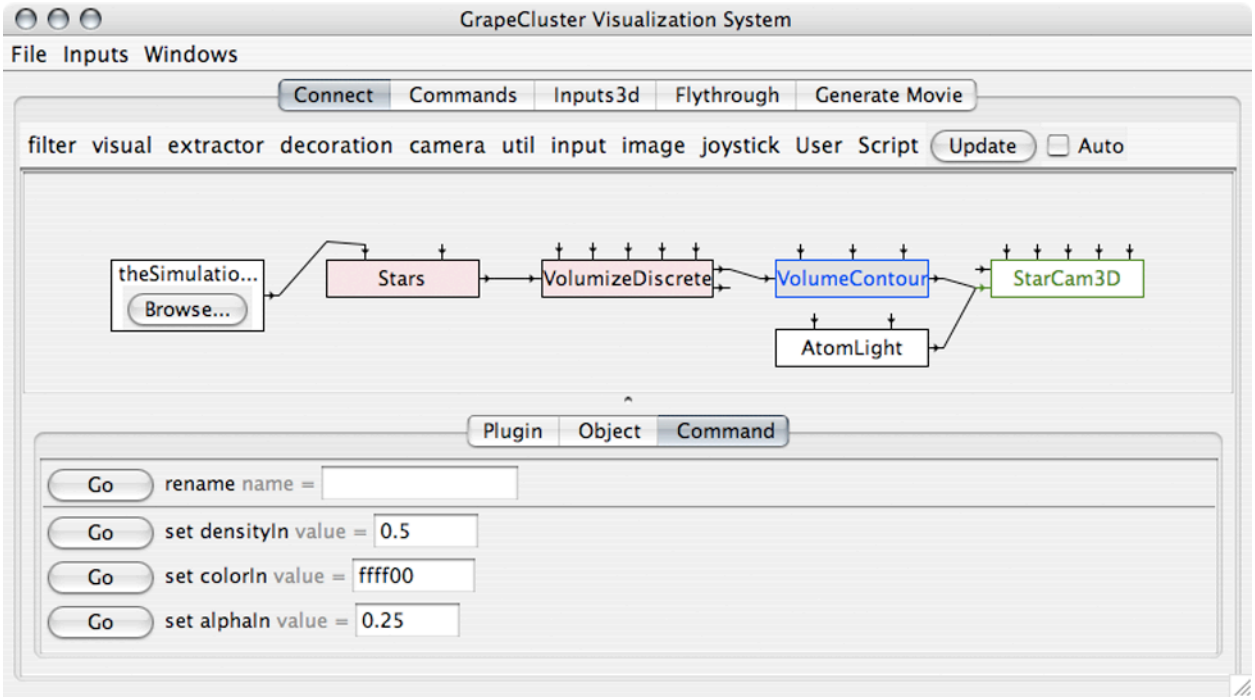


Figure 19: VolumeContour plug-in being used in a visualization

Setting densityIn determines the density contour that is rendered (all densities are between [0-1]).

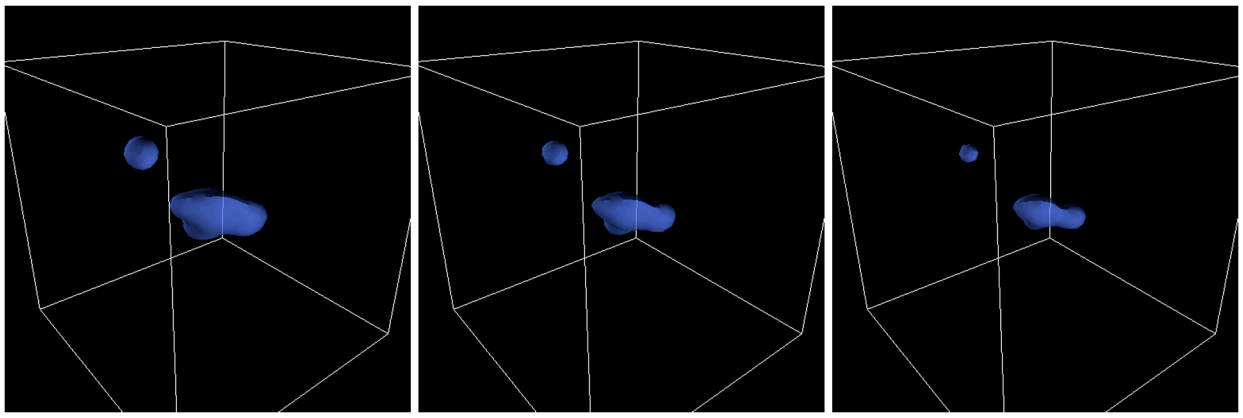


Figure 20: Changes in densityIn

ColorIn and alphaIn affect the rendered appearance changing the color and transparency of the contour respectively.

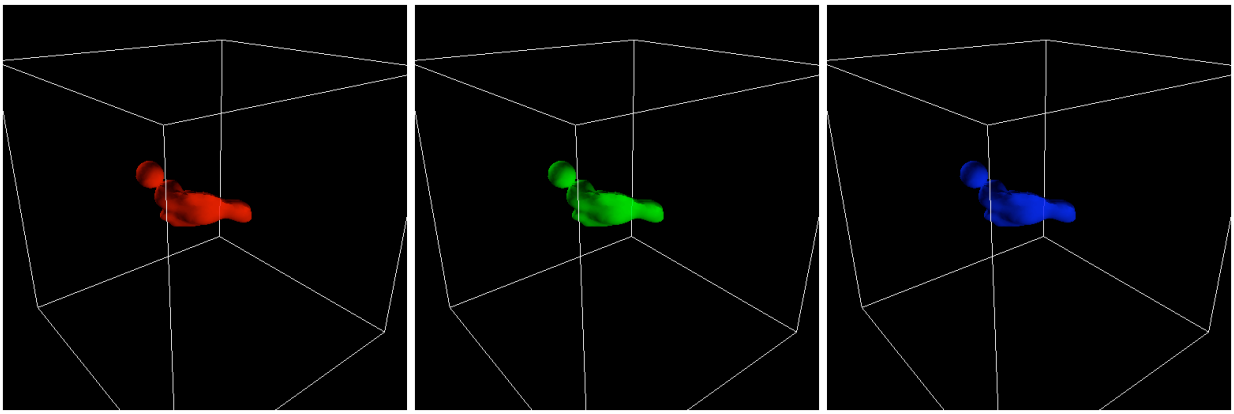


Figure 21: Affect of colorIn

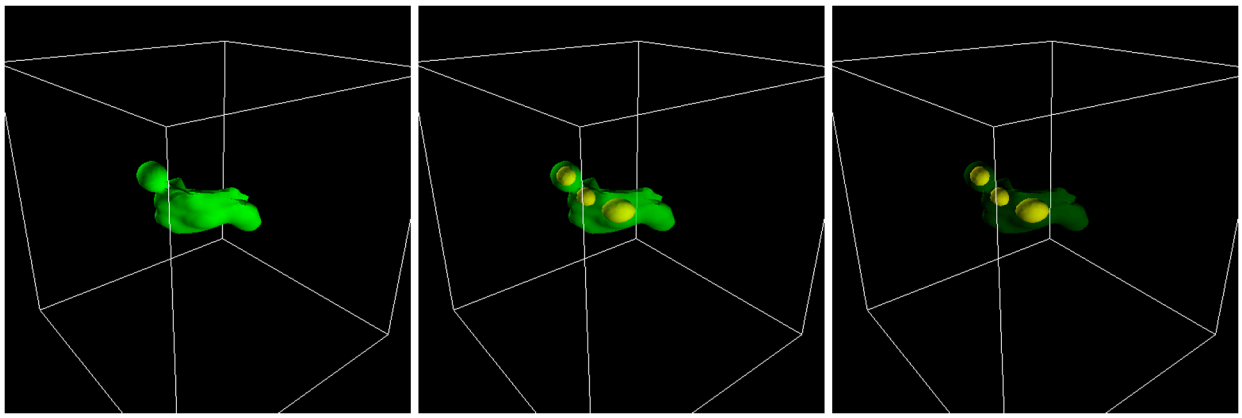


Figure 22: Affect of alphaIn

3.3.2 Marching Cubes Algorithm

The Marching Cubes algorithm extracts an isosurface from a 3D data set. The algorithm works by dividing space up along a regular 3D grid forming a series of cubes throughout the space. Each of these cubes is then looked at and each corner is compared to an input value. This corner is then marked as less then or greater then the input value. If all the corners are less then or greater then the input value then I know the surface does not pass through this cube. If there is at least one corner that differs from the others we know the surface passes through, and by looking at the relationships between the less then and greater then corners we can approximate the part of the surface that pass through this cube.

3.3.2.1 2D Marching Squares

To understand the algorithm it is useful to start in 2D. In the diagram below space has been divided up by a grid into a series of squares. At each intersection is a density value. From this grid I want to extract an approximate density boundary.

The first step is to determine which corners are less then (blue dots) the input density and which are greater (purple dots). I can then approximate the boundary by placing a vertex (red dots) on the grid between the corners that are less then and those that are greater then the input value. The exact location of the vertex on the grid line is calculated through linear interpretation of the corners. Finally the calculated vertices are connected to form the boundary.

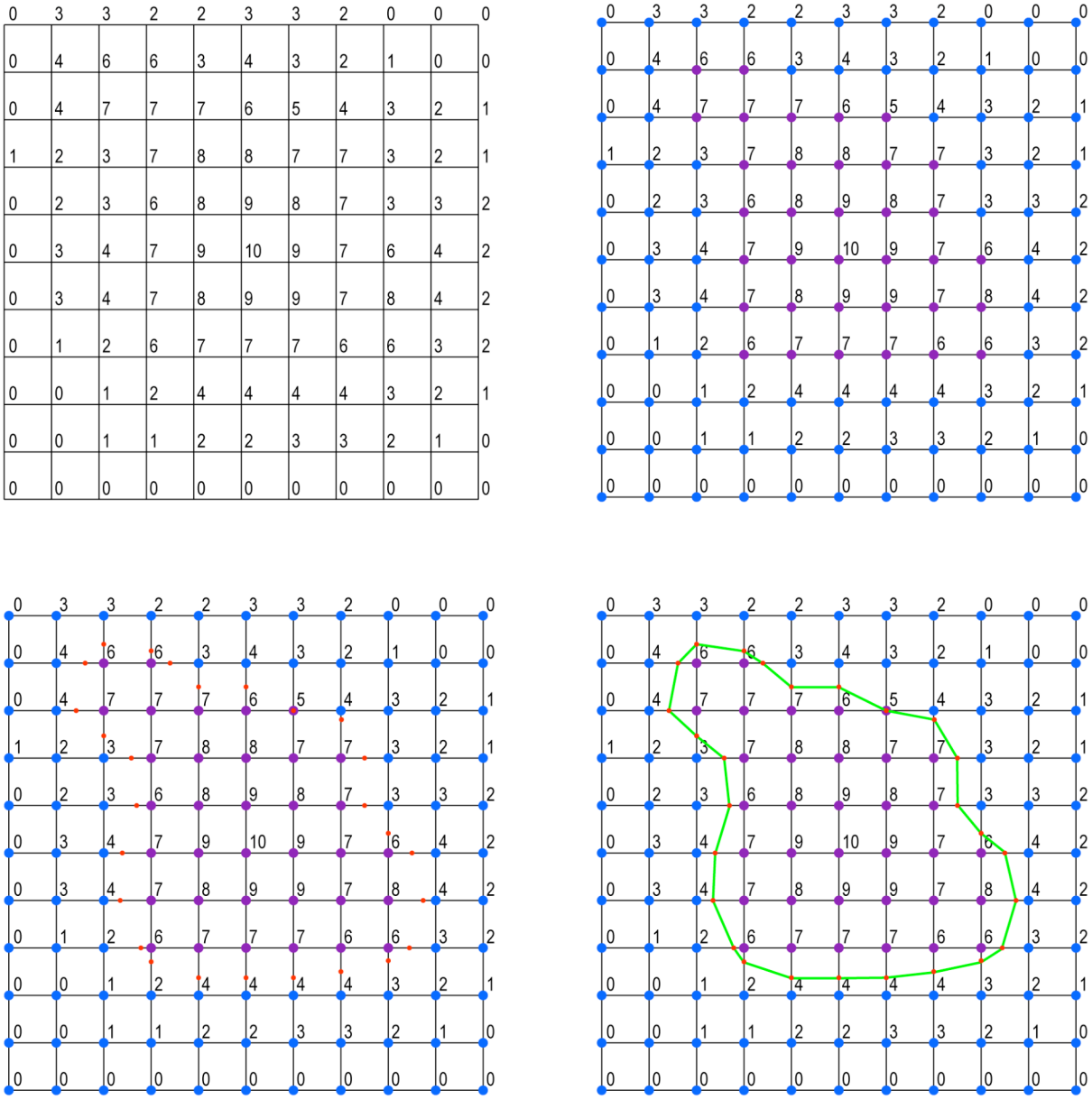


Figure 23: Marching Squares example

3.3.2.2 3D Marching Cubes

The same principles apply in 3D as in the 2D case except now there are eight corners and a possibility of 256 corner combinations. The 256 cases can be reduced to the following 15 cases through the use of rotation, mirroring, and by swapping the color of the corner in each case.

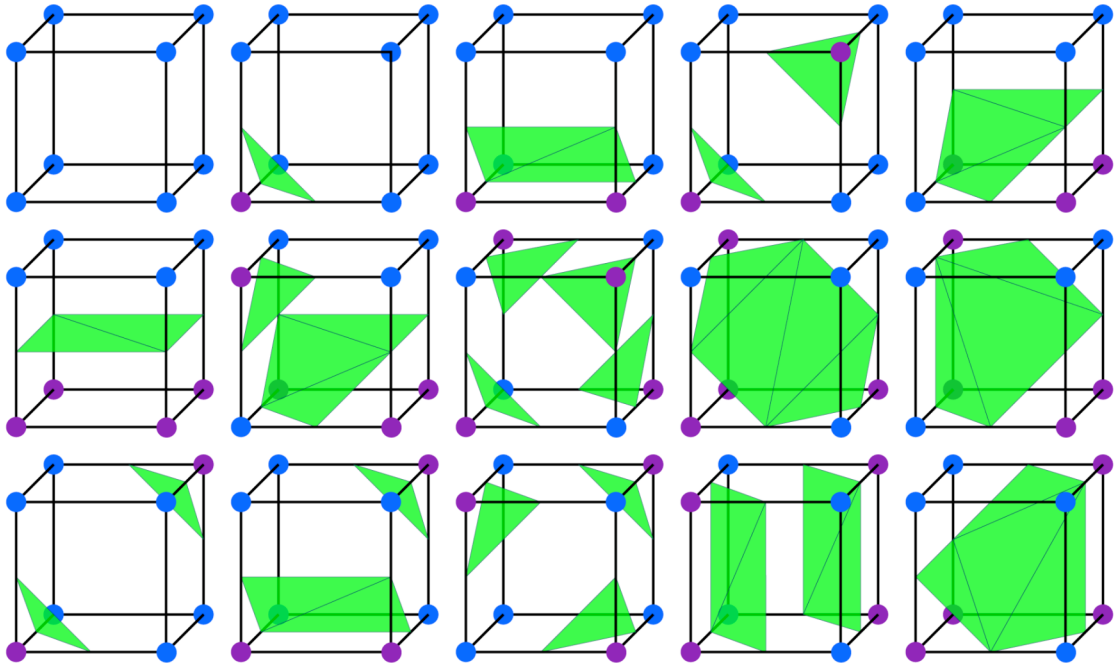


Figure 24: 15 Marching Cubes cases [4][7]

Once we know the 256 possible cases we can "march" through all the cubes in the volume and use a lookup tables to determine the triangles to be rendered in that cube. When all cubes have been marched through the surface construction is complete and the triangles can be rendered.

3.4 Direct Volume Rendering

The final method I use for visualizing the data is direct volume rendering. In this method I take the volume and map the density values to a color and transparency. I then render a cube that represents the portion of the volume and render it with the mapped color and transparency. This renders a cloud like shape that has additive properties of the layers that a being looked through.

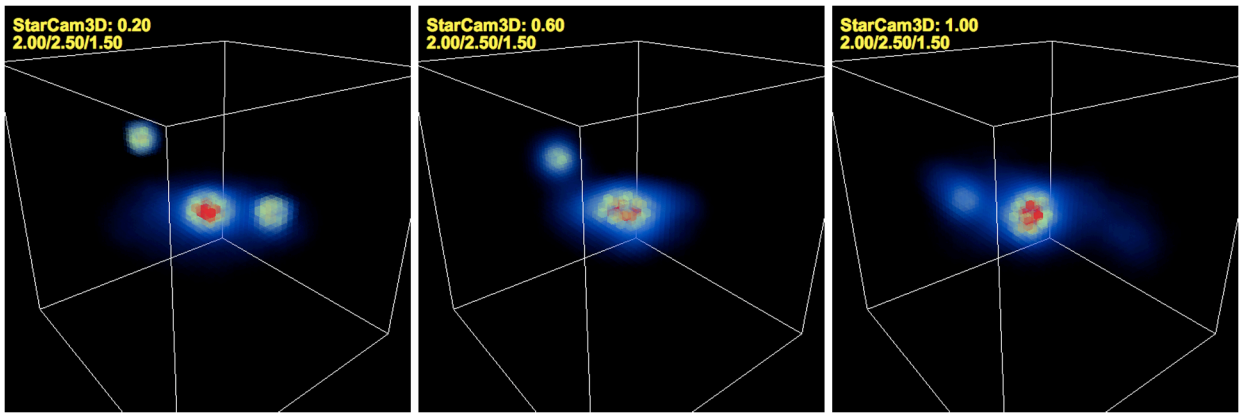


Figure 25: Volume density cubes rendered directly

3.4.1 VolumeAsCubes Spiegel Plug-in

This plug-in has no parameters, takes a volume as input and outputs a Java3D BranchGroup.

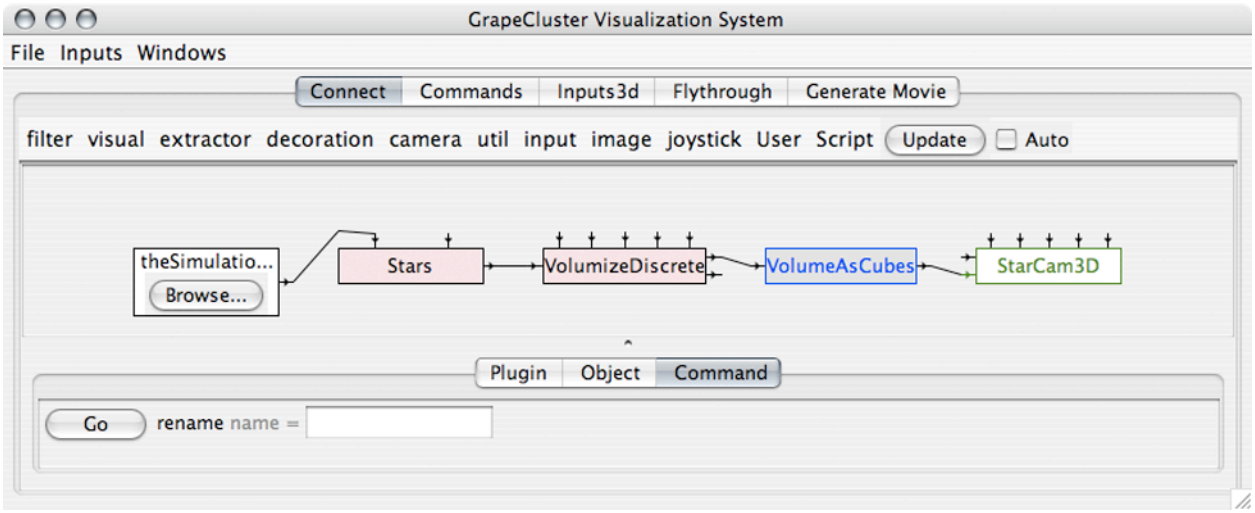


Figure 26: VolumeAsCubes plug-in in an Spiegel visualization

4 Results

Each of the visualizations explored in this project have try to solve the problems and achieve the objectives set out in the beginning of this paper. This section looks at how well the final visualization met those goals.

4.1 Kernel Density Estimation

All the visualizations in this paper attempt to over come over-plotting through the kernel density estimation. The process was described in section 3.1.

I believe this process resolves the issue of over-plotting fairly well. It allows me to take data that when plotted at reasonable sizes would result in a solid with fuzzy edges and turn that data into density which gives some sense of how may stars are in a given area relative to other areas as shown below:

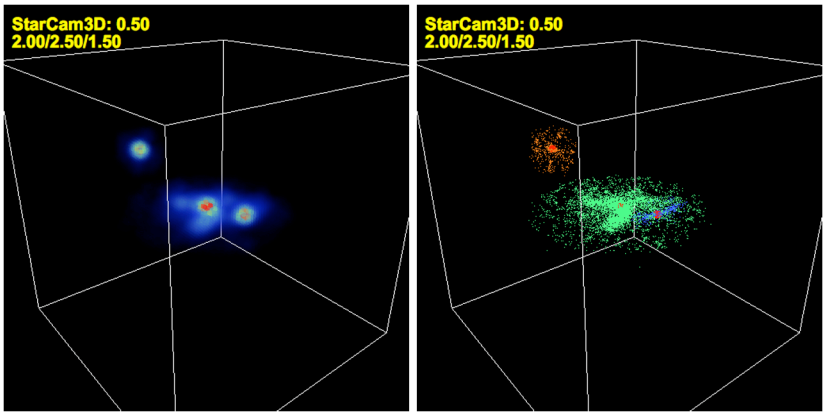


Figure 27: Density estimation vs. point plotting

The method is very accurate, which is important in visualizing scientific data, but is also very expensive to calculate. For this plug-in to be truly usable it would need to be run on very fast machines or a new method be found to calculate the volume density.

4.2 Slicing

The slicing visualization allows for a detailed look at the interior of a segment of data. This visualization solves data obscuring by eliminating it. This allows the observer the ability to cut away all outer layers to see what is occurring in the given layer of data. In the image below it is possible that density you can see in the slice may not have been apparent when looking at the image as a whole or through some other method.

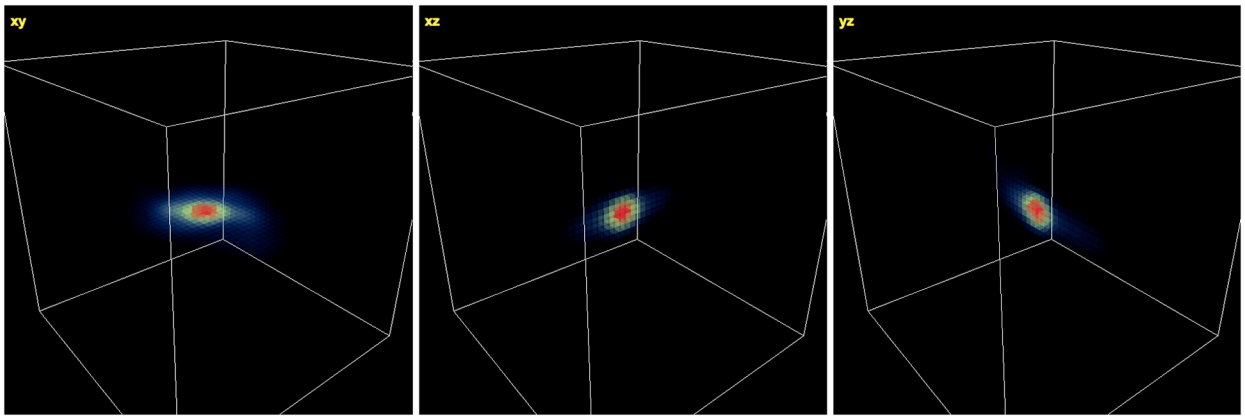


Figure 28: Slices of a data set

While this method is of great use in the medical field where the datasets are well defined and the users knows where he wants to look, it would have less use in stellar modeling were the points of interest may not be known ahead of time. This means that this form of visualization would need to be used in conjunction with other forms that allow the users to get a better overall sense of the model first.

4.3 Projection

The projection visualization is a collapsing of the data on to a 2D plane. This gives an overall sense of where the stars are in the system in a single axis. When all three axes are projected an observer can very quickly tell where most of the stars are concentrated. This method can also be combined with other methods to provide a better overall picture. The picture below shows how the projection gives quick visual reference to the density of the stars being plotted.

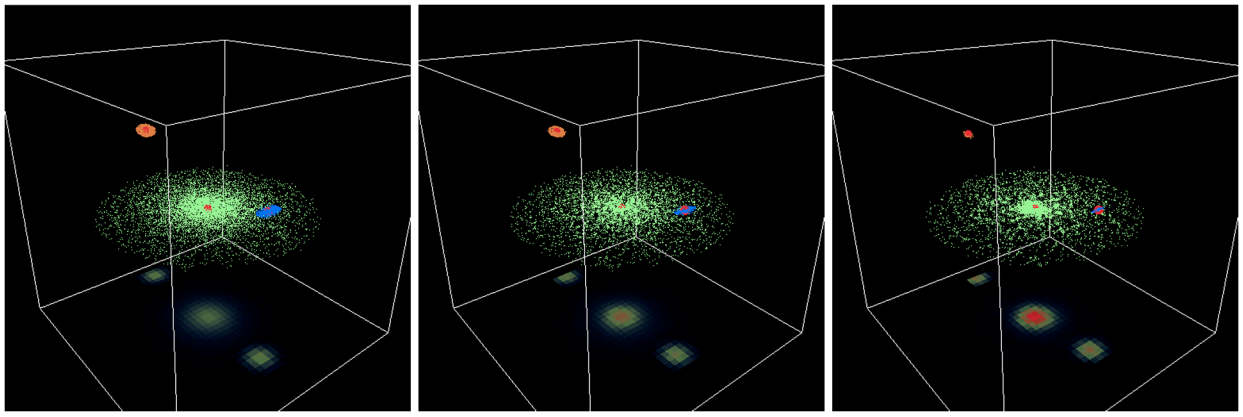


Figure 29: Project with stars as points

The projection visualization by being an inherently 2D method loses much of the nuances of the 3D information. While it is possible to gain some of this back through the use of multiple projections it can still be difficult to tell exactly how the stars may be spread through the 3D space by looking at the projections alone.

4.4 Projection Contour

The projection contour visualization is really an extension of the projection visualization above and suffers from the same problems. The advantage of this method is that it offers an additional dimension that clues the observer more quickly into the data being display. It is also more effective when being displayed over time. The image below is the same data and times as the previous image. One can see how the same data being projected into 3D gives a better sense of the difference in magnitude of the densities that is difficult to discern in the flat projection based sole on color.

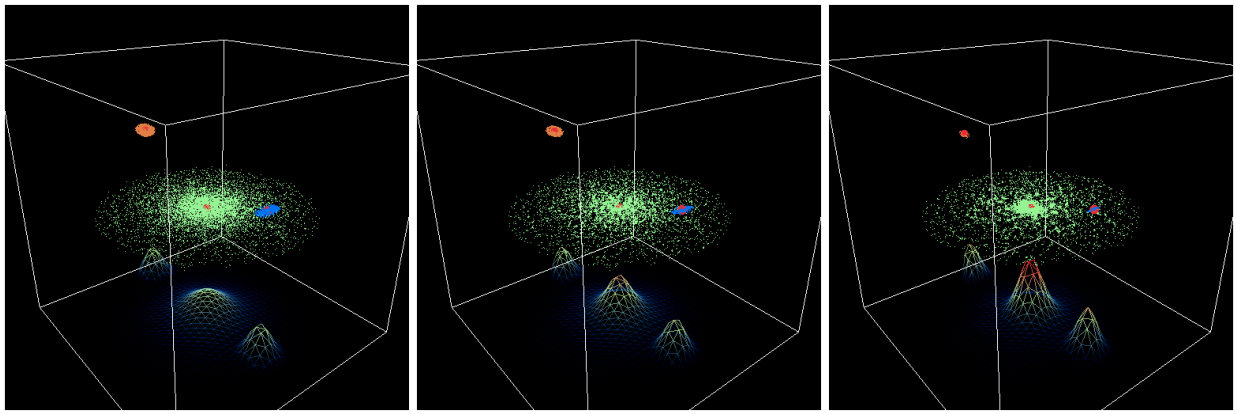


Figure 30: Projection contour with stars as points

4.5 3D Contour

The 3D Contour visualization offers a detailed view of segments of the volume. Much like the slice method this method give the control to the user and allows them to selectively view portions of the volume that they are interested in. Unlike the slice method this visualization can still give an overall sense of the data in 3D and becomes even more effective when used to multiple times to create nesting contours as seen in the right most image below.

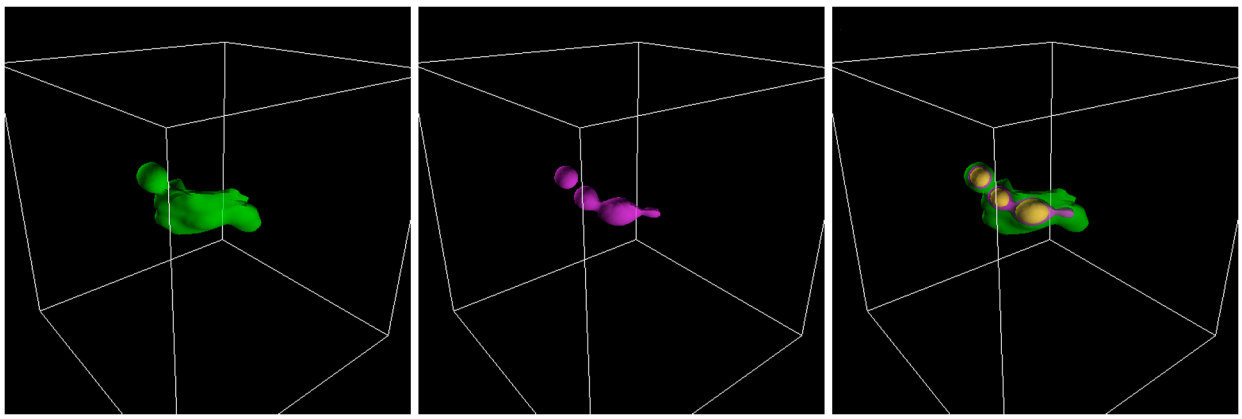


Figure 31: 3D Contours

Choosing the density values that will reveal interesting results is a downfall of this method. Through the use of several contours representing a range of densities interesting areas can be narrowed in on. An issue remains during the rendering of these shells stemming from my inability to control the order the shells are rendered. This can cause inconsistent results when rendering multiple shells. The problem makes the inner shells appear to flash in and out of existence.

4.6 Direct Volume Rendering

The direct volume rendering may offer the best overall picture of the data. It shows the full range of 3D data available giving the user the most information of any of the other rendering methods.

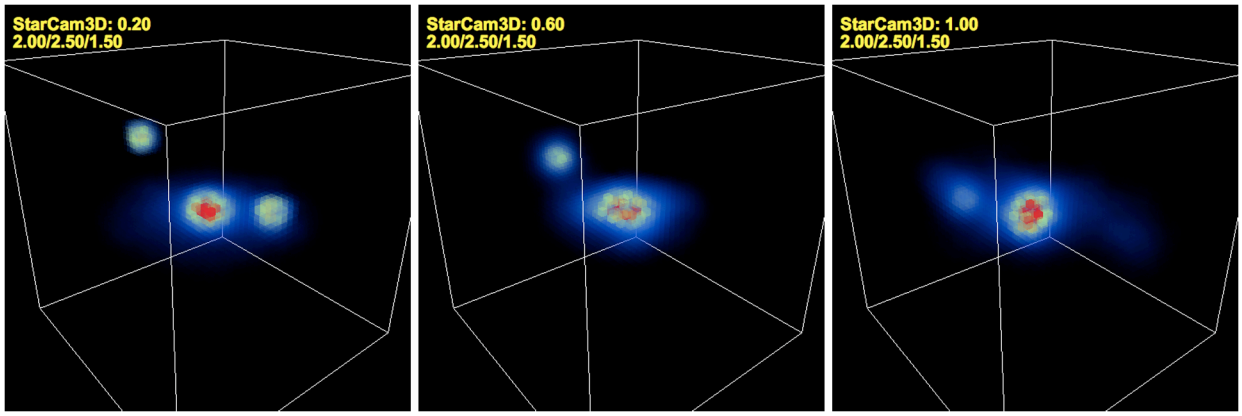


Figure 32: Direct volume rendering of volume density

The weakness in this method is that it make still sufferer from data obscuring when there are large areas of high density relative to the rest of the data set. Another weakness is that it has the slowest performance of any of the rendering methods.

5 Conclusions

The primary objectives of this project were as follows:

- Determine problems involved with visualizing large datasets
- Experiment with ways to visualize these datasets in a meaningful way
- Try to overcome the problems identified earlier in the visualizations
- Compare results to determine how successful the visualization was

Two main problems were identified, over-plotting and the self-obscuring nature of 3D datasets. In the course of this project I experimented with a variety of methods for displaying the datasets, each having its own strengths and weaknesses. I found out that the display of information is more art form then science.

6 Future Work

There are several improvements that could be made to increase performance. Along with these improvements there a few outstanding issues that remain unsolved.

The first thing that could be improved is the speed of the kernel density method. A simpler kernel function could be chosen then the normal density function to reduce the mathematical complexity of the calculation.

The VolumeAsCubes could be rewritten to use a technique called billboarding. Billboarding would replace the cube rendered at each square with a single polygon (square, circle, etc) that is rendered facing the camera. This would reduce the overall number of polygons being rendered, reducing the time and memory space required by the visualization.

Other improvements include the ability to define slices that are not perpendicular to an axis, define your own color maps, and a way to normalize the densities across different datasets so that two visualizations can be compared.

Appendix

1 User Guide

1.1 System Flow Chart

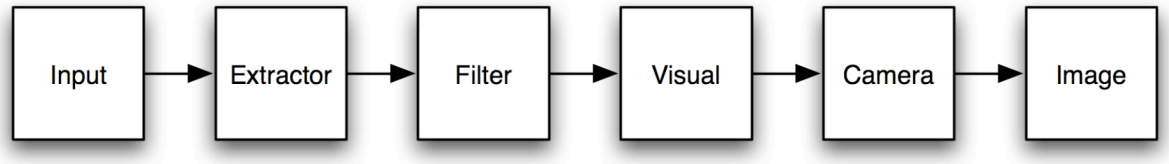
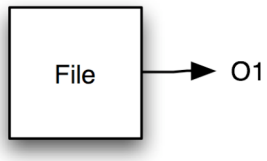


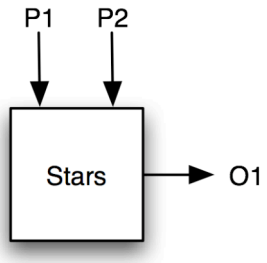
Figure 33: Typical Visualization Workflow



1.1.1 Input

Allows for file browsing

Outputs:
O1: file: String

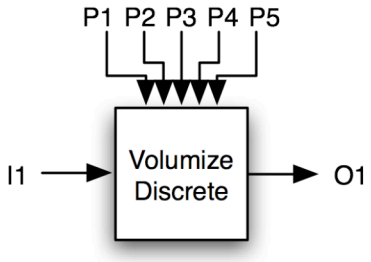


1.1.2 Extractor

Extracts star information from a directory

Parameters:
P1: file: String
P2: time: Double

Outputs:
O1: stars: StarIDMap



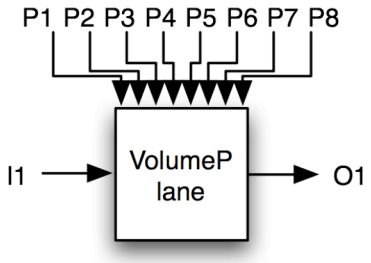
1.1.3 Filter

Creates a volume density map from star data

Input:
I1: stars: StarIDMap

Paramters:
P1: boxSizeIn: Double
P2: bandWidthIn: Double
P3: xCutOffIn: Double
P4: yCutOffIn: Double
P5: zCutOffIn: Double

Outputs:
O1: volume: Volume



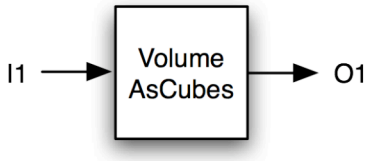
1.1.4 Visual

Used for all plane and projection based visualizations.

Input:
I1: volumeIn: Volume

Parameters:
P1: distanceIn: Double
P2: deltaIn: Double
P3: planeIn: String
P4: projectionIn: Boolean
P5: contourProjectionIn: Boolean
P6: maxContourHeightIn: Double
P7: meshIn: Boolean
P8: cullIn: Boolean

Outputs:
O1: object: BranchGroup

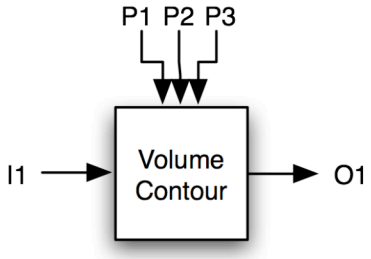


Used for isosurface contour visualiztions.

Inputs:
I1: volumeIn: Volume

Parameters:
P1: densityIn: Double
P2: colorIn: String
P3: alphaIn: Double

Outputs:
O1: object: BranchGroup

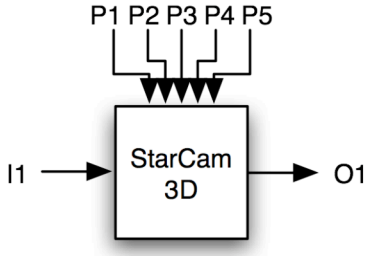


Used for volume rendering of volume density as cubes.

Inputs:
I1: volumeIn: Volume

Outputs:
O1: object: BranchGroup

1.1.5 Camera/Image



Combines camera and image functions into one plug-in.

Inputs:
I1: objects: BranchGroup

Parameters:
P1: time: Double
P2: location: Point3d
P3: lookat: Point3d
P4: up: Point3d
P5: size: Dimension

Outputs:
O1: image: BufferedImage

1.2 Spiegel Plug-in Update Method

```
294   protected void update () {
295
296         allStars = starsIn.get();
297         boxSizeV = boxSizeIn.get() / 1000.0;
298         bw = bandwidthIn.get();
299
300         xCutOff = xCutOffIn.get();
301         yCutOff = yCutOffIn.get();
302         zCutOff = zCutOffIn.get();
303
308               // find max and min star positions in each direction
309               findSpaceSize();
310
311               // divide up the space
312               createBins();
313
314               // count stars in each bin
315               placeStarsInBins();
316
317               // create volume density
318               calculateDensity();
319
322               // output results
323               volumeOut.set( volume );
326
331   }
```

VolumizeDiscrete does most of the heavy lifting in all my visualizations. The previous code block of code is the update statement of the plug-in, each plug-in has this method. In lines 296-302 all inputs are read in. Method call findSpaceSize on line 309 loops through the input stars and finds the min and max star positions in each direction that are within the input cut offs. The area contained in this cube is the divide by the box size in the createBins method. Then the stars are placed into the bins in placeStarsInBins. Finally the density is calculated and the volume density map created.

1.3 Volume Data Type

The volume object represents the volume density map. The volume consists of VolumeDensity objects, which store the density, position, index within the volume and cube size of each region of space contained in the volume. The volume maintains seven references to the individual VolumeDensity cubes. These bins are linked hash sets, which allow for fast and iterative access to groups of cubes. Passing an index component accesses the xBin, yBin, and zBin and every cube with that same index component is returned. The xyBin, xzBin and yzBin are accessed by passing two out of the three indexes and pass the column of cubes with the matching indexes in the indicated components

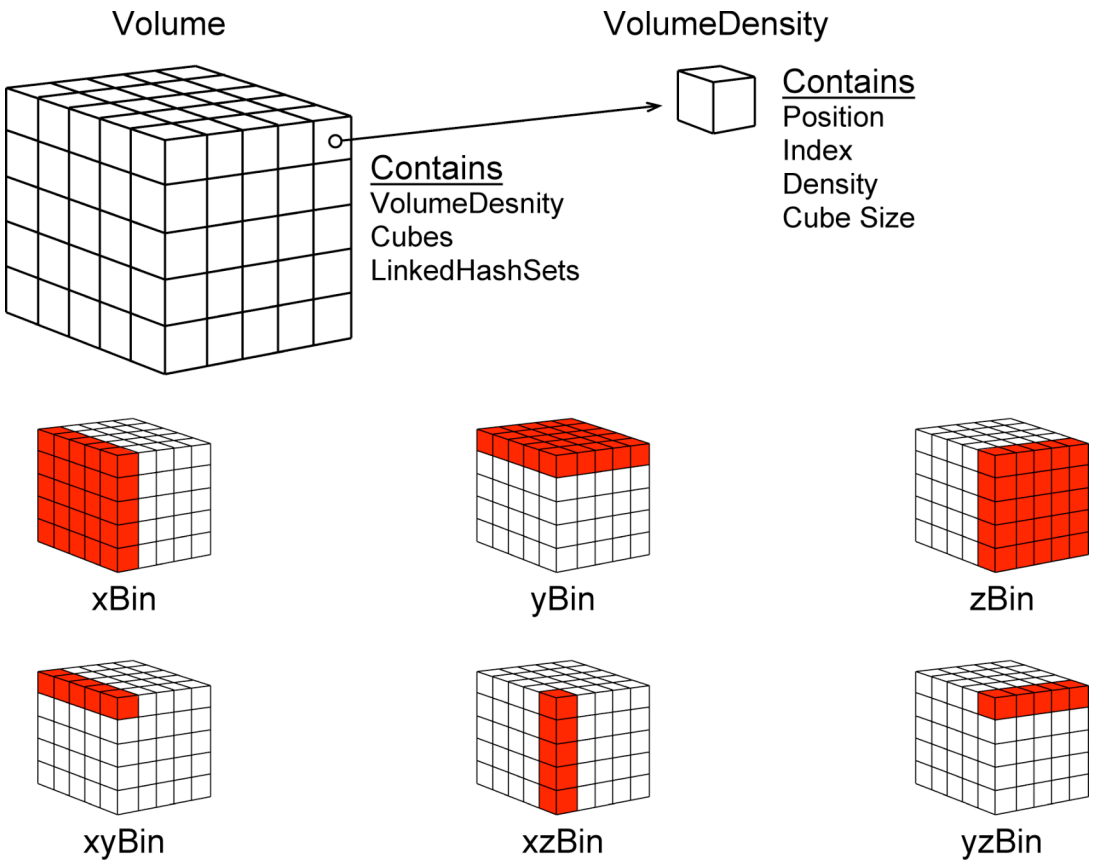


Figure 34: Anatomy of Volume

1.4 Issues

The main unresolved issue seems to be a memory leak within the system that effect the VolumeAsCubes visualization the most since it renders large numbers of polygons (6 for each cube division in the space that has a density greater then 0). I was unable to track down the cause of the leak and believe it may be a Java3D issue.

References

[1] Bischof, Hans-Peter, Edward Dale, and Tim Peterson. "Spiegel - A Visualization Framework for Large and Small Scale Systems." Proceedings of the 2006 International Conf. of Modeling Simulation and Visualization Methods. MSV'06: June 26-29, 2006, Las Vegas. 2006. 199 - 205. The Anhinga Project. Rochester Institute of Technology. 15 Aug. 2006 <http://www.cs.rit.edu/~hpb/Publications/msv_06.pdf>.

[2] Bloomenthal, Jules. "An Implicit Surface Polygonizer." Graphics Gems IV. Ed. Paul S. Heckbert. Graphics Gems 5. San Diego: Morgan Kaufmann, 1994. 324-349.

[3] Bourke, Paul. Polygonising a Scalar Field. May 1994. 15 Aug. 2006 <http://local.wasp.uwa.edu.au/~pbourke/modelling/polygonise/>.

[4] Lingrand, Diane, et al. The Marching Cubes. 31 May 2002. 15 Aug. 2006 <http://www.essi.fr/~lingrand/MarchingCubes/accueil.html>.

[5] Lorensen, William E., and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." Proceedings of the 14th Annual Conf. on Computer Graphics and Interactive Techniques. International Conf. on Computer Graphics and Interactive Techniques. New York, NY: ACM, 1987. 163169. ACM Portal. Wallace Lib., Rochester Inst. of Technology, Rochester, NY. 12 July 2005 <http://doi.acm.org/10.1145/37401.37422>.

[6] Scott, David W. Multivariate Density Estimation: Theory, Practice, and Visualization. New York: John Wiley & Sons, Inc., 1992.

[7] Sharman, James. "The Marching Cubes Algorithm." Exaflop. 1999. 15 Aug. 2006 <http://www.exaflop.org/docs/marchcubes/>.

[8] Shreiner, Dave, et al. OpenGL Programming Guide. Ed. Tyrrell Albaugh and John Fuller. 5th ed. Upper Saddle River: Addison-Wesley, 2006.

[9] Wand, M. P., and M. C. Jones. Kernel Smoothing. Boca Raton: Chapman & Hall/CRC, 1995.

[10] Wood, Zoe J., et al. "Semi Regular Mesh Extraction from Volumes." VISUALIZATION. In Proceedings of the 11th IEEE Visualization 2000 Conf. Washington, DC: IEEE Computer Society, 2000. ACM Portal. Wallace Lib., Rochester Inst. of Technology, Rochester, NY. 12 July 2005 <http://portal.acm.org>.

[11] Film 10 Sagittal MR Brain Scan. Image. Human Gross Anatomy. UC Davis Health System. 27 Aug. 2006 <http://medocs.ucdavis.edu/cha/400/rdifolder/headfilm/headfilm.htm>.

[12] MRI scan from Stanford's 3T MRI system. Image. Gary Glover and Lara Foland. 2004. BIRN: Biomedical Informatics Research Network. 27 Aug. 2006 <http://www.nbirn.net/Publications/Articles/press-temp.htm>.