

**The Kooshball Algorithm: A Ray Tracing Region  
Growing Algorithm for Medical Data  
Segmentation**

by

Nicholas D. Levitt

Submitted to the Department of Nuclear Engineering  
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Nuclear Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author .....  
Department of Nuclear Engineering  
May 16, 1994

Certified by .....  
Dr. Andrew B. Dobrzeniecki  
Research Scientist  
Thesis Supervisor

Accepted by .....  
Prof. John E. Meyer  
Chairman, Departmental Committee on Undergraduate Students

ARCHIVES  
MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

JUN 30 1994

LIBRARIES

# **The Kooshball Algorithm: A Ray Tracing Region Growing Algorithm for Medical Data Segmentation**

by

Nicholas D. Levitt

Submitted to the Department of Nuclear Engineering  
on May 16, 1994, in partial fulfillment of the  
requirements for the degree of  
Bachelor of Science in Nuclear Engineering

## **Abstract**

A three-dimensional region-growing algorithm has been developed to segment medical image data. Instead of the traditional "marching cubes" approach, the algorithm grows out in one-dimensional rays. A convolution edge-detection is performed in one dimension along each ray to find the surface of the structure of interest. In order to navigate around convex curves, the algorithm chooses random points along the previous rays from which to begin again in a recursive manner. The resulting cloud of surface points is screened for points whose distance from the center of mass are close to that of their neighbours. After screening, the resulting reduced cloud of surface points is ready to be tessellated into a surface.

The algorithm does not screen points if the shape is overly convoluted; nor does it find any edge but the strongest along each ray. Its recursive structure, screening intensity, and number of surface points are variable. These parameters are tested on both geometric shapes and an MRI data set containing ventricles of the brain. The algorithm clearly finds the surface of the ventricle. Further improvements to be made in the algorithm are outlined.

Thesis Supervisor: Dr. Andrew B. Dobrzeniecki  
Title: Research Scientist

## **Acknowledgments**

There is one person who deserves all the acknowledgement in this section - and that is Andy Dobrzeniecki, my thesis advisor and mentor. He had the ideas which were implemented in this document, the time to explain, and the enthusiasm to make it always seem worthwhile. Without his guidance, none of this would have been possible. I have met few people in my life who are at the same time as knowledgeable and as approachable as Andy Dobrzeniecki, and I owe him a great debt for the time and effort he has contributed to make this project a success.

# Contents

<b>1</b>	<b>Background</b>	<b>9</b>
1.1	The Segmentation of 3-D Data Sets . . . . .	9
1.1.1	The Goal . . . . .	9
1.2	Significance to Medical Imaging . . . . .	10
1.3	Other Strategies Currently Being Used . . . . .	10
1.3.1	Brain Model Algorithms . . . . .	10
1.3.2	Surface Tracing Algorithms . . . . .	11
1.4	The Concept of Region Growing Algorithms . . . . .	11
1.4.1	General Concept . . . . .	11
1.5	Speed and Accuracy of Region Growing Algorithms . . . . .	11
1.5.1	Speed . . . . .	11
1.5.2	Traditional Problems with the Region Growing Approach . . .	11
1.6	Thesis Overview . . . . .	12
<b>2</b>	<b>Brief Description of the Kooshball Algorithm</b>	<b>13</b>
2.1	Algorithmic Strategy . . . . .	13
2.1.1	Finding a Group of Edge Points: The Simple Approach . . . .	13
2.1.2	Charting around Convex Curves: The Recursive Algorithm . .	14
2.1.3	Screening Surface Points . . . . .	15
2.2	Advantages over Previous Techniques . . . . .	15
2.2.1	Advantages of a Three-Dimensional Algorithm . . . . .	15
2.2.2	Further Applications for Edge Detection . . . . .	15
2.2.3	Advantages of the Ray Approach vs. Marching Cubes . . . . .	16

<b>3</b>	<b>The Algorithm - Creating a Single Koosh</b>	<b>17</b>
3.1	Determining a Uniform Angular Distribution of Rays . . . . .	17
3.1.1	Determining the Number of Latitudes vs. the Number of Rays	19
3.2	Finding the Intersection of the Rays and the 3-D Image Surface . .	21
3.3	Creating a Single Ray: The 3-D Bresenham Algorithm . . . . .	21
3.4	Surface Detection Along a Single Ray . . . . .	22
<b>4</b>	<b>Recursive Branching</b>	<b>23</b>
4.1	Convex Curves and Child Kooshes . . . . .	23
4.2	Set-up of the Recursive Tree . . . . .	23
4.3	Tree Level Calculation . . . . .	24
4.4	Recursive Structure of the Code . . . . .	25
<b>5</b>	<b>Screening the Cloud of Surface Points</b>	<b>27</b>
5.1	Choosing a Center Point . . . . .	28
5.2	Ordering the Surface Points in a One Dimensional Array . . . . .	28
5.3	Performing the Screening . . . . .	30
<b>6</b>	<b>User Parameters</b>	<b>31</b>
6.1	Surface Point Parameters . . . . .	31
6.2	Recursion Parameters . . . . .	32
6.3	Screening Parameters . . . . .	33
<b>7</b>	<b>Verification</b>	<b>34</b>
7.1	Test Data Sets . . . . .	34
7.1.1	The Three Sets of Test Data . . . . .	34
7.1.2	Accuracy . . . . .	35
7.1.3	Varying the Screening Stringency . . . . .	35
7.1.4	Tree Structure Tests on a Hollow Sphere . . . . .	35
7.1.5	Size of the Image . . . . .	38
7.1.6	Number of Surface Points . . . . .	38
7.2	Medical Data Sets . . . . .	40

7.2.1	The Test Image . . . . .	40
7.2.2	Potential Problems for the Kooshball Algorithm . . . . .	40
7.2.3	Accuracy . . . . .	42
7.2.4	Changing the Recursive Tree Structure for Better Results . . . . .	42
7.2.5	Testing the Screening Procedure On a Medical Data Set . . . . .	43
7.3	Summary of Results . . . . .	43
<b>8</b>	<b>Future Work</b>	<b>45</b>
8.1	Choosing Surface Points Closer to the Starting Point . . . . .	45
8.2	Creating and Processing Multiple Edge Points on a Single Ray . . . . .	46
8.3	Replacing the Insertion Sort . . . . .	46
8.4	Screening Surface Points on Convex Surfaces . . . . .	46
8.5	Time Optimization . . . . .	47
8.6	Incorporation into the WCBICL 3-D Viewing System . . . . .	47

# List of Figures

3-1	<i>Left: Phi and Theta Conventions Right: The Globe Model (Flat View)</i>	19
3-2	Conical Approximation of the Top Half of the Sphere. . . . .	20
5-1	Solid Angle Method for Comparing $r$ Values . . . . .	28
5-2	Charting a Spiral Path Down the Sphere . . . . .	29
7-1	Solid Sphere (10,000 Points) 10%, 20% ... 100% Points Passed through Screen . . . . .	36
7-2	Results on Hollow Sphere for Varying Tree Depths . . . . .	37
7-3	Graph of Results for Time vs. Number of Surface Points Trial . . . .	39
7-4	Slices of the Smoothed Ventricle Image: slices 13,16,23 and 29 . . . .	41
7-5	40,000 Point Trial of the kooshball algorithm on the medical test data. Here, slices 13,16,23 and 29 are shown. The starting point is on slice 17. Note that the algorithm does quite well, especially for those slices close to the starting slice. . . . .	41
7-6	Depth of Tree Trials: Slice 29 of a Brain Ventricle Data Set. The depth of the tree in the trials from left to right was 0,1,2,20. The trial with only one tree level yields the best results. . . . .	43

# List of Tables

7.1	Screening 10,000 Surface Points from the Solid Sphere . . . . .	37
7.2	Legend for the Tree Structures Tested . . . . .	38
7.3	Varying Data Volume Dimensions and CPU Run Times, 10,000 Point Trial on a Solid Sphere. The data shows a linear relation relation between the length of the cube and the CPU run time of the algorithm, suggesting that kocshball algorithm reacts very favorably to large data volumes. . . . .	38
7.4	Number of Surface Points vs. CPU Time for a 64x64x64 Data Volume Cube Test (Cube Length = 52) . . . . .	39
7.5	Information on Medical Data Trials . . . . .	41



# Chapter 1

## Background

### 1.1 The Segmentation of 3-D Data Sets

The purpose of the kooshball<sup>TM1</sup> algorithm is to provide a quick and easy means of segmenting a three-dimensional data set. Its use lies in the segmentation of medical image data. The segmentation of an MRI or CT medical image set is performed by taking advantage of the contrast in pixel intensity between different structures in the image. If an accurate outline of the structure can be provided quickly and semi-automatically, important volume data on the structure may be made easily available.

#### 1.1.1 The Goal

The goal of this project is to provide a method in which to segment an individual structure out of a three-dimensional data set. The system is to be semi-automated; therefore, the user will oversee the process and give some information concerning the general region in which the structure is located, as well as at least one point in the interior of the structure. Once finished, this algorithm will be incorporated into the medical imaging software system under development at the MIT Whitaker College Biomedical Imaging and Computation Laboratory.

---

<sup>1</sup>Named after the Kooshball<sup>TM</sup>, a common children's toy consisting of rubber strands branching out in all directions from a central mass. The algorithm spreads out from a point in a uniform distribution of rays resembling a Kooshball<sup>TM</sup>; hence the name of the algorithm.

## **1.2 Significance to Medical Imaging**

Volume determination can play a key role in the medical diagnoses made from patient image data. Some neurological disorders can lead to the atrophy of certain anatomical structures, thereby providing a means for identifying the disorder if an accurate volume determination of these structures can be provided [4]. A quick method of finding the volume of structures within a medical image set would save clinicians time, as well as providing consistent and accurate data for diagnosis.

## **1.3 Other Strategies Currently Being Used**

MRI segmentation is an extremely active area of research at this time, as can be seen from the 65 current papers which appear in the INSPEC library database on the subject. Currently, many systems used to segment MRI images are based on geometric models, threshold algorithms and edge detection techniques, but work is being done with neural networks, wavelet texture identification [1], and traditional region growing techniques combined with other methods to improve reliability [6, 8].

### **1.3.1 Brain Model Algorithms**

Several strategies are being investigated for automatic and semi-automatic segmentation of an MRI scan of the brain. Collins, Peters, Dai and Evans [2] have worked on a method using both raster and geometric data models to fit by deformation onto the MRI data. These techniques take advantage of a top-down approach, in that the algorithms have prior topological knowledge of the entire structure. However, these techniques are not easily adaptable to different types of data, and must be combined with either an interactive local deformation algorithm, or a low level approach to perform edge detection procedures.

### **1.3.2 Surface Tracing Algorithms**

Another method of segmenting an MRI data set is to trace the surface by starting at a point on the surface, and tracing a path along the surface with a set of voxels until the whole surface is connected. Several methods exist for performing this task, three of which are described in [7].

## **1.4 The Concept of Region Growing Algorithms**

### **1.4.1 General Concept**

A region growing algorithm is one which starts at a single point, and grows out from that point until it reaches what it perceives to be the edge of the structure volume. The algorithm “grows” by slowly including more and more voxels (or “volume cubes”) around the central point until it reaches an edge which surrounds it on all sides. In this way, a solid volume is created in the interior of the structure.

## **1.5 Speed and Accuracy of Region Growing Algorithms**

### **1.5.1 Speed**

Traditional region growing algorithms are slowed down by the fact that they must cover every single voxel in the interior of the structure. When applied to larger structures, this can take an ever-increasing amount of time and memory.

### **1.5.2 Traditional Problems with the Region Growing Approach**

At the same time, region growing algorithms also suffer from “leakage”, which is the phenomenon resulting from a gap in the outer surface of the object. Upon reaching this gap, the region growing algorithm has a tendency to continue on its outward path

and “leak out” of the structure volume. In order to be effective, a region growing algorithm must address this problem in some manner.

Hence, the motivation for this work has been to develop segmentation strategies that overcome the limitations in speed and accuracy of existing methods.

## **1.6 Thesis Overview**

Chapter 1 has briefly presented the motivations and background for the development of ray-projecting segmentation methods. In Chapter 2, the kooshball algorithm is described in general terms, while in Chapter 3 the details are provided for the mechanics of generating a single ray and locating candidate edge points along that ray. Chapter 4 explains the recursive branching method of the algorithm, while Chapter 5 discusses the important technique of removing erroneous edge points from the generated cloud of surface points.

Chapter 6 lists the parameters in the kooshball algorithm that are under the control of the user. Chapter 7 provides the results of testing the developed algorithm on both geometric data and actual medical MRI data. Finally, chapter 8 describes future work and the ultimate applications and implementations of this work.

# Chapter 2

## Brief Description of the Kooshball Algorithm

### 2.1 Algorithmic Strategy

#### 2.1.1 Finding a Group of Edge Points: The Simple Approach

As mentioned before, the conventional region-growing algorithm begins at a point in the interior of a volume, and adds voxels to this point, forming a clump of voxels which continues to grow until it reaches what it perceives to be the surface of the structure. The kooshball algorithm follows the same concept, except that it does not add voxels to form a solid mass around the initial point. Instead, the algorithm extends a number of rays directly outward from the initial point in every direction until all the rays reach the edge of the image volume. The initial point with rays generating from it in all directions resembles a Kooshball<sup>TM</sup>; hence the name of the algorithm (see footnote in Chapter 1).

The algorithm performs a very simple edge detection in one dimension along each of the rays. In this manner, each ray contributes one surface point to the final array of surface points. Summarizing the initial concept of the kooshball algorithm: a point is chosen in the interior of the object, rays are projected out from it in all directions,

and a surface point is found on each ray using an edge detection technique.

### **2.1.2 Charting around Convex Curves: The Recursive Algorithm**

A problem occurs when the object is convex, because parts of the inside surface cannot be seen from the initial point. Some rays projected from the starting position may pierce the surface of the object and then travel back into the object at a later point, giving two or more surface points to consider. Only the nearest surface point is considered that of the target structure. The rest remain invisible to the kooshball algorithm.

In order to detect the “invisible” portion of a convex shape, the algorithm must be re-started at a point in that portion of the object. One way in which this could be achieved is to have the user provide more than one seed point from which to project rays. A second method for mapping out the unseen portion of the structure would be to have the algorithm restart itself in different places which are known to be within the interior of the object. Our kooshball algorithm takes this second approach.

Points are randomly chosen along different rays, between the initial point and the identified surface point on the ray. At each of these secondary points, a new kooshball of rays projecting in all directions from the point is formed. From these projected rays, more points are chosen, and more “kooshes” (rays projected uniformly in all directions) are born. The algorithm continues this process in a recursive manner until a sufficient number of surface points are found. Borrowing from the language of recursion, each koosh resulting from a point on the ray of a previous koosh will be called a “child koosh”, while the previous koosh from whose ray it was derived will be called the “parent koosh”. The set of parent and child kooshes form a recursive tree, whose depth and width may be determined by the user.

### **2.1.3 Screening Surface Points**

Due to noise, the fuzziness of object boundaries, and occasional ray leakage, a cloud of surface points results from the above procedure. Before tessellating the cloud of points into a surface, a screening algorithm has been developed to weed out incorrect surface points.

This screening algorithm first finds the center of mass for the cloud of proposed surface points. Each point is then compared to its neighbors in terms of its distance from the center of mass. Those points which are much farther or much closer to the center of mass than their neighbors do not pass the screening.

## **2.2 Advantages over Previous Techniques**

### **2.2.1 Advantages of a Three-Dimensional Algorithm**

The advantage of a three-dimensional algorithm, as opposed to a two dimensional algorithm which combines surfaces on different slices, is that a three dimensional algorithm makes fuller use of the three-dimensional data given. Because no one dimension is weighted any differently than any other dimension, the algorithm avoids unsmooth approximations in the z-direction which might be present if a two-dimensional algorithm were used.

### **2.2.2 Further Applications for Edge Detection**

The kooshball strategy is a very general method of locating the surface volumetric of a structure. It is versatile in that the one-dimensional edge detection can be carried out using any number of different convolution filters and edge detection methods, while preserving the structure of the kooshball algorithm. Once the algorithm is fully developed, different edge detection methods may easily be substituted, thus increasing the effectiveness of the algorithm on different types of image data.

### **2.2.3 Advantages of the Ray Approach vs. Marching Cubes Speed**

The most obvious advantage of the kooshball algorithm is its speed. The kooshball algorithm covers more space than the marching cubes approach, using fewer pixels. The kooshball algorithm is also computationally efficient by performing a one-dimensional edge detection as opposed to a complete three-dimensional analysis. For these reasons, the kooshball algorithm is expected to be much faster than the marching cubes region-growing algorithm, especially for large structures in which there is a large volume to be covered.

#### **Leakage**

In the kooshball approach, leakage is expected to be much more easily detectable than in the traditional region-growing approach, primarily because a ray that leaks out of the volume will most likely travel a fair distance before being stopped by the next surface. This means that a very simple filter will screen any leaked rays quite effectively based on the distance of the proposed surface points from the actual surface of the object.



## **Chapter 3**

# **The Algorithm - Creating a Single Koosh**

The following three chapters give a detailed presentation of the inner workings of the kooshball algorithm. A uniform distribution of rays is traced out from a single point, and the edges of the volumetric structure are determined along each ray by an edge detection convolution. This uniform cluster of rays emanating from the initial point is named a “koosh”. This chapter concerns itself with the formation of the koosh, and the subsequent edge detection procedure.

### **3.1 Determining a Uniform Angular Distribution of Rays**

The mathematical description of a uniform spherical distribution of rays around a point is not trivial. Such a uniform distribution is required so that from a given seed point we can project a given number of rays,  $n$ , such that the rays cover the volume around the seed point in a uniform manner;  $n$  will vary depending on how densely the surface points are to be located. Two methods of forming a uniform distribution were considered.

1. The first method discussed was the 'positive charge' method, involving the computation of the solution to a perfectly uniform distribution of rays around the sphere. This is best visualized as a set of mobile positive charges placed on the surface of the sphere. Each charge will end up as far away as possible from all the neighbouring charges. When one more charge is added, all the previous charges change their positions slightly. Mathematically, this solution was far too complex to use efficiently.
2. The 'globe model' was a second method considered. In this model, the intersections of the longitude (constant  $\theta$ ) and latitude (constant  $\phi$ ) define the outward direction of each ray. However, a problem exists in that the concentration of rays around the poles is significantly larger than around the equator, due to the increased density of longitude lines.
3. The third model considered, and the one which was actually used, is the same as the globe model, except that the  $\theta$  angles between the rays on each latitude line are not equal as in the globe model (See Figure 3-1). Instead, the horizontal arc length between the points is kept constant around every latitude in the sphere, thus putting large numbers of points on the long central latitudes while decreasing the number of points at the poles. In order to avoid the points from longitudinally lining up in a column, a wrap-around feature is used whereby the arc length between points on one latitude line overlaps onto the next latitude line.

Though this third method is still only an approximation to a uniform distribution, it is computationally simple, and allows a large number of rays to be calculated. It is also guaranteed not to leave any gaps in the ray distribution around the single point.

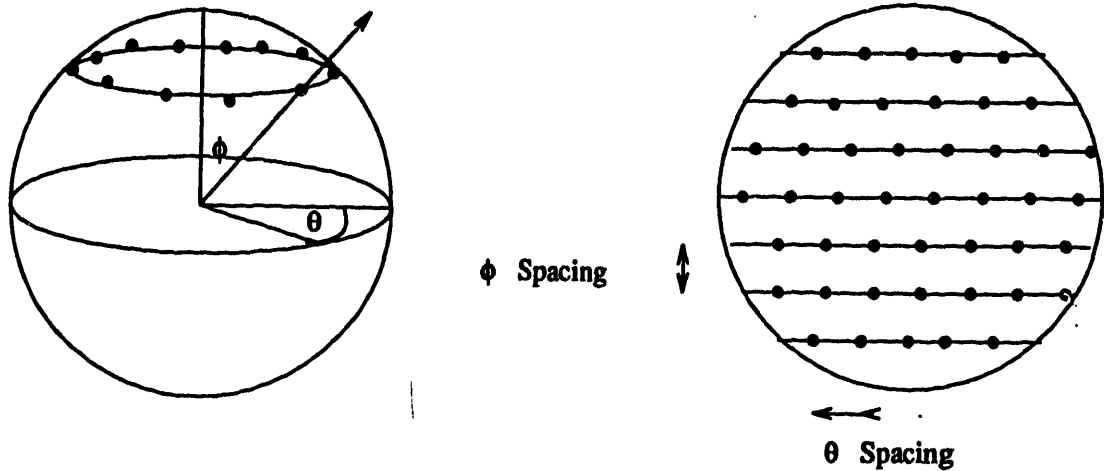


Figure 3-1: *Left: Phi and Theta Conventions Right: The Globe Model (Flat View)*

### 3.1.1 Determining the Number of Latitudes vs. the Number of Rays

The preceding section on ray distribution leads to a question about the relationship between the number of rays emanating from a single point, and the number of latitudes which should be put onto the “direction sphere”. (The “direction sphere” is a mental construct of a sphere centered around the initial point. Points on this sphere represent locations where the rays intersect the surface of the sphere. The model enables one to judge the distribution of the rays. See Figure 3-1, Right.) If too few latitudes are used, the rays miss large phi angles, while if too many latitudes are used, there is a danger of all the rays lining up vertically along one side of the sphere. In the ideal case, one would want the theta angles between points in the sphere approximately equal to the phi angles between the latitudes.

In order to analyze this problem, the two halves of the sphere were modeled as cones of radius  $r$  and height  $r$  (see Figure 3-2). If there are  $p$  surface points on a cone, and  $n$  evenly spaced latitudes, the distance between the latitudes on the surface of the cone (the phi-spacing, in the case of a sphere) will be  $\frac{r}{n} * \sqrt{2}$ . Because the circumference of the cone varies linearly with height, the average circumference is  $\pi r$ , making the sum of the circumferences  $n\pi r$ . This is the only dimension affected by

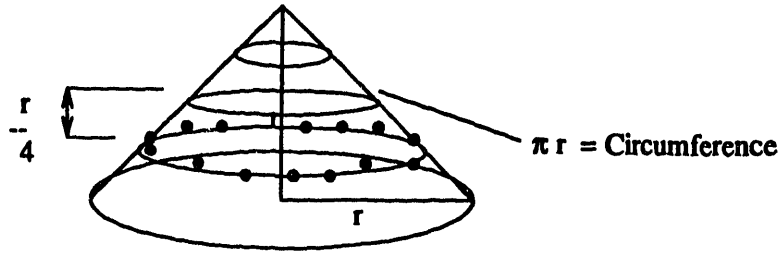


Figure 3-2: Conical Approximation of the Top Half of the Sphere.

the modeling of a cone as opposed to a sphere. Therefore, the distance between two points on the same latitude on the surface of the cone (the theta-spacing) becomes:

$$\frac{n\pi r}{p}$$

It is wished that the phi-spacing and the theta-spacing be equal in order to have a uniform distribution. Thus,

$$\frac{n\pi r}{p} = \frac{r}{n} * \sqrt{2}$$

Therefore,

$$n = \sqrt{\pi p \sqrt{2}}$$

Thus for a cone, the optimal number of latitude lines varies with the square root of the total number of surface points.

Based on these calculations, and neglecting the coefficient, it was decided that in order to ensure a good degree of uniformity in the distribution, the number of latitude lines on the sphere should be exactly equal to the square root of the number of rays emanating from the originating point.

## **3.2 Finding the Intersection of the Rays and the 3-D Image Surface**

In order to use the Bresenham algorithm (discussed in the next section) to its full extent, it is necessary to find where each ray reaches the edge of the data volume, given the theta and phi values calculated for each ray. Therefore, an algorithm was created to calculate the point at which a ray reaches the outside of the image volume, given the originating point inside that volume, and the polar coordinate angles theta and phi of the ray from that point. This algorithm first tests to determine which side of the data volume the ray will intersect, by comparing the angle pointing to the edge of the data volume to the angle of the ray. This is done for edges in all three dimensions. The function then performs another trigonometric calculation to determine the exact voxel on the side of the data volume which will mark the end of the ray. This is a simple trigonometric calculation.

## **3.3 Creating a Single Ray: The 3-D Bresenham Algorithm**

A three-dimensional version of the highly efficient Bresenham algorithm [3] was developed for the purpose of speed in tracing the ray from the koosh point to the surface of the given image volume. The Bresenham algorithm picks voxels along the ray, following a straight line. No voxel along the line touches more than two other voxels. Furthermore, the Bresenham method does not involve floating point computation, hence it is computationally efficient; this is an important consideration given the large number of rays that the kooshball algorithm typically processes.

There is an alternative to finding the ray endpoint and then using the Bresenham algorithm to find a ray. This would be to grow out the ray itself, using its theta and phi angles along with an altered Bresenham algorithm which checks for the surface of the volume as it goes along. In the future, this new method might be an advantageous

one to try in order to increase speed. For the present, however, finding the point at the edge of the data volume, and then performing the usual Bresenham selection of points is sufficient.

### **3.4 Surface Detection Along a Single Ray**

A one-dimensional spatial convolution is performed to find the edge of the structure of interest for each ray. Typical kernels are  $[-1,-1,0,1,1]$  or  $[1,1,0,-1,-1]$  for first derivative detection, and  $[1,-1,0,-1,1]$  or  $[-1,1,0,1,-1]$  for second derivative detection. The convolution algorithm is general enough so that it can select any number of possible edge points from a ray, given any length of kernel. However, the edge detection in the present algorithm only allows for one possible edge point on each ray. It also chooses only the strongest of all edges along the ray, and does not take into account the closeness of the edge to the initial point. In the case of two edges having exactly the same convolution value, the edge nearest to the point of origin is always selected. However, problems will still arise in the very common case where the edge of a far-off structure sends out stronger edge detection signals than the target structure's edge.

# Chapter 4

## Recursive Branching

### 4.1 Convex Curves and Child Kooshes

In order to find its way along convex curves, the kooshball algorithm chooses a number of rays from the primary koosh (we use the term “koosh” to describe a point with a uniform distribution of rays emanating from it). On each of these chosen rays, a new point of origin (or “koosh point”) is picked in a random fashion to begin a new koosh.

A recursive algorithm is readily applicable due to the nature of the problem. Each spawned child koosh depends on its parent koosh for a starting point. Each parent koosh, meanwhile, is expected to have one or more child kooshes. Since each koosh process proceeds in exactly the same manner as its parent koosh, and is also dependent on the parent koosh for a starting point, a recursive algorithm seems to be the most plausible. It is also very easy to make a flexible tree structure in a recursive setting as opposed to an iterative setting, which tends to be more rigid. Thus, a recursive algorithm was chosen due to both its simplicity and flexibility.

### 4.2 Set-up of the Recursive Tree

The algorithm proceeds in a recursive fashion; that is, each generation produces a set of rays, some of which are selected as seed points for the next generation of koosh points. This creates a tree of koosh points, with each koosh point located on a ray

of its predecessor. The recursive structure of the tree at each level is controlled by two variables: the total number of rays per koosh point, and the number of rays per koosh point used to spawn child koosh points. These two parameter arrays are called *raysperkoosh* and *branchfactor* respectively (see Chapter 7: Verification). Both have an effect on the depth vs. width ratio of the tree. If the number of rays per koosh point is large on a certain level, then a large number of surface points will result from that level. If the branch factor is large, the number of koosh points on the next level will be large, leading to a large number of surface points. Two constraints on the structure of the tree are: it can't narrow as its depth increases (i.e. *branchfactor* must be  $\geq 1$ ), and the *branchfactor* can only take on integer values.

The breadth and depth of such a tree are important considerations, because the structure of the tree can affect the probability of having a large number of false surface points. The situation proceeds as follows: If the tree is narrow and deep, then a false point located at the top level of the tree can lead to false points all the way down, thereby seriously affecting the quality of the segmentation. This is why it is good to have a very broad first level, whose rays definitely originate from inside the structure. However, if the tree is too broad, then it may not be able to reach down enough levels to maneuver itself around convex curves within the structure. A balance must be made.

At present, the algorithm defaults to a *raysperkoosh* of 100 and a *branchfactor* of 100. At every other level, *raysperkoosh* is 10 and *branchfactor* is 1. This means that there will be 100 rays on the first level, and 1000 rays on each level after that. We use the notation “(100,100),(10,1)...” to denote this tree structure (See Chapter 6 for a full explanation of this notation).

### 4.3 Tree Level Calculation

Given the total number of required surface points, the algorithm calculates the number of levels to include in the tree, and the number of rays which should be computed to find edge points in the last level. This calculation is complicated by the factors



outlined below.

The rays which are initially chosen as sites from which to choose the points of origin for the child kooshes are chosen in an even manner, i.e. the second out of every five rays. This, however, leads to a problem when all the points of origin have exactly the same type of uniform distribution, since generations of child koosh will be colinear with an initial ray, and all will find the same surface point that was originally found by that ray. In a deep tree, this could be lead to many repeated surface points.

This problem has been addressed in two separate ways. First of all, the edge points of any ray which is being used to spawn a child koosh are not added to the array of surface points, since it is assumed that the same edge point will be found by a ray traveling in the same direction in the next generation. If the number of rays emanating from a kooshball of the next generation is the same as for the original generation, then the same uniform distribution of rays around the point will be used, but the point detection along the original ray will only be performed once.

Secondly, given that the number of rays per origin point, as well as the *branchfactor* remain the same over two generations, the positions of rays chosen to support child kooshes are translated randomly, so that in one generation, the second ray out of every five is chosen, while in the next generation, the fourth ray out of every five is chosen. These two measures are used to avoid colinear sets of previous seed points.

## 4.4 Recursive Structure of the Code

In order to implement the tree-like kooshball function, the code is written in a recursive manner. This provides the simplest and most elegant code. The recursive structure of the algorithm is set up so that most of the variables used in the recursive algorithm are static, and are accessed and adjusted by non-recursive functions which are called by the recursive algorithm. The code is written in this manner in order to make the manipulation of important variables simple to comprehend. Only a small number of functions actually manipulate the global variables (these are the functions *TreeInfoStore()* and *EdgePointStore()*, both of which use static variables to achieve

their purpose).

# Chapter 5

## Screening the Cloud of Surface Points

The outcome of the previous steps is a cloud of points which must be tessellated into a surface. Between these two steps, however, a screening algorithm is used to weed out any points which are plainly not on the surface. The screening procedure outlined in this section is not designed to work on convex curves, unless all parts of the surface of the structure are visible from its center of mass.

Given a center point  $P$ , let  $r$  be the distance between  $P$  and a proposed surface point  $S_n$  (see Figure 5-1). In its ideal form, our screening strategy would take all points ( $S_1..S_n$ ) contained within a solid angle centered around the  $PS$  axis. The  $r$  value of the selected point would be judged against the  $r$  values of its neighbours,  $S_1..S_n$ , and accepted based on its deviation compared to the standard deviation of  $r_n$  in the set ( $S, S_1, S_2, S_3, ..S_n$ ).

In actuality, however, the calculation of a solid angle centered around each PS axis is computationally intensive as well as unnecessary. Instead, a method of linearly ordering all the points, in reference to their spherical coordinate angles ( $\theta, \phi$ ) from point  $P$  is utilized. The points are ordered such that they are surrounded by their closest neighbours (by angle). Points are then compared to neighbouring points in the linear array.

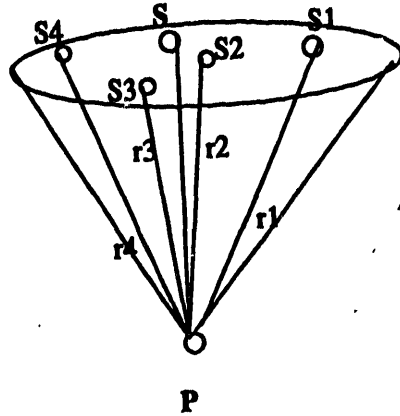


Figure 5-1: Solid Angle Method for Comparing  $r$  Values

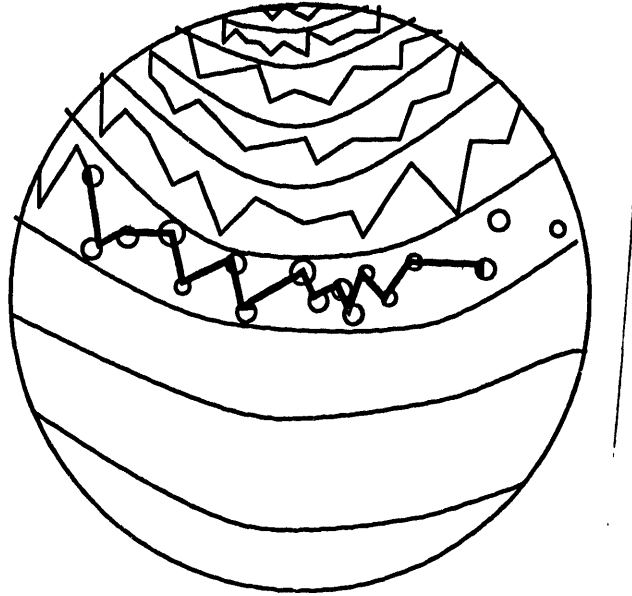
## 5.1 Choosing a Center Point

The center point  $P$  is taken as the center of mass for the cloud of surface points. It is assumed that in cases where the screening is done, the center of mass will occur inside the object, and all parts of the surface will be visible from that center point (i.e. no ray from that point will pass through the surface of the object more than once). In chapter 8, the possibility of having more than one center point from which to judge the surface points will be discussed.

## 5.2 Ordering the Surface Points in a One Dimensional Array

The objective in ordering the proposed surface points into a one-dimensional array is to ensure that each point is bordered on both sides by those points nearest to it (where "nearest" refers to the angle between two points. One method by which to achieve this objective is to chart a spiral path along the surface of a sphere drawn around point  $P$  (see Figure 5-2).

Each point along the spiral path is then ordered on one criterion: how far along the spiral path it occurs. The points zigzag along the path, never having a phi spacing of more than the path width. If the path is widened, then the phi-spacing of the



**Figure 5-2: Charting a Spiral Path Down the Sphere**

ordered points will increase, while the theta-spacing decreases (See Figure 5-2). It is important to choose the correct path width in order to keep the average phi and theta spacing between the points equal.

The implementation of this concept uses a step-wise spiral, with a large number of steps per rotation. In Chapter 3, a simple approximation was made to equalize the average phi-spacing and the average theta-spacing by taking the square root of the total number of rays, and slicing the sphere into that number of horizontal slices. The same approximation can be made to determine the path-width and number of revolutions of the spiral.

In this case, however, the manner in which the points are analyzed plays an important part in the determination of the path width. Since all neighbouring points are given equal weight in the screening analysis, it is optimal if the average phi-spacing of the points compared in the screening is equal to their average theta-spacing. In the algorithm, the number of points compared with the point being tested in each screen is stored in the constant SCREENWIDTH. If SCREENWIDTH is 5, then each point  $S$  is being compared with four of its neighbours ( $S_1, S_2, S_3, S_4$ ): two from each side.

Therefore, the average theta-spacing from point S is twice the average theta-spacing between any two points. Doubling the average phi-spacing, i.e. doubling the path width, corrects this problem. The implementation is shown in the *DoScreening()* procedure:

```
numslices = (int)(sqrt((double)numsurfacepoints)*(2/(SCREENWIDTH-1)));
```

At the current time, the sorting of the surface points is done using an insertion sort. It is therefore one of the time-limiting factors of the algorithm, especially when the number of points is raised, since the sort routine is of the order n-squared. The changing of this sorting procedure to a QuickSort will be mentioned in Chapter 8, Future Work.

### 5.3 Performing the Screening

The actual screening is performed as follows: each point is compared to its neighboring points in the one-dimensional ordered array. Including the point which is to be screened, the  $r$  distance from SCREENWIDTH points are passed to the *ScreenPoint()* algorithm. The average and standard deviation of these points are found. If the point being screened deviates from the average by more than the standard deviation multiplied by a given factor, called the *sdfactor*, then the point is not accepted.

Starting at a value of 0.1, the *sdfactor* is raised at 0.1 intervals until the percentage of points which pass through the screen is equal to that specified by the user.

The screen procedure returns a boolean value which is 1 (pass) or 0 (fail) to each point in the *surfacepoints* array.

# Chapter 6

## User Parameters

Apart from the image to be segmented, the user has a number of parameters which can be adjusted for the specific data volume, or specific structure to be segmented. These parameters can be separated into three classes: those dealing with the surface points, those dealing with the recursive method, and those dealing with the screening of the surface points. Many of these parameters are present for testing purposes, so that the optimum values can be determined for the regular use of the algorithm.

### 6.1 Surface Point Parameters

The most important restriction in the program occurs in the number of surface points which it is asked to find. This will affect the speed of the algorithm as well as its eventual accuracy in surface determination. The surface points parameter determines the total number of surface points which are proposed, which is a greater number than the total number of surface points considered after screening.

Another parameter is the starting point, which will eventually be inputted by clicking the mouse at the proper point in the three-dimensional data volume. At present, the coordinates for this parameter must be inputted manually. The starting point must be located inside the volume of the structure, at least two voxels away from the surface, depending on the length of the convolution used for the edge detection algorithm.

The maximum ray length is a parameter whose intent is to localize the length of any particular ray. If the ray length is set to the longest path-length which exists within the structure, it ensures that the rays do not stray too far from the structure, while at the same time making sure that a ray is not halted in the interior of the structure if it was begun in the interior of the structure.

## 6.2 Recursion Parameters

The recursion parameters specify the structure of the recursive tree, as well as the number of surface points found at each level of the tree. The structure of the tree is determined by the branching factor, a factor whose value is the number of child kooshes resulting from all the rays of a parent koosh. The number of surface points found on each level is dependent on the rays projected from each seed point, i.e. the number of projected rays that make up one koosh.

Therefore, in order to make a tree with 100 points in the first level, 1000 on the second level, and 1000 on every level afterwards, one could have 100 rays per koosh on the first level, with a branching factor of 10. On the second level, one would also want 100 rays per koosh, to make 1000 points in all (there will be 10 kooshes on the second level since the branching factor for the first level was 10). For the third level, the branching factor could be 5, and the rays per koosh could be 2 (this would also make a thousand surface points). Afterwards, the branching factor would have to be decreased to 1, and the rays per koosh would have to remain at 2.

The program possesses no ability to narrow the tree as it goes further down - each branch of the tree is considered in the same manner. Therefore, either all kooshes on one level have a child koosh, or none of them do. This rule is only broken on the last and second to last levels, due to the restriction placed on the tree by the number of surface points parameter.

Once again, these parameters have been placed in the program mainly for testing purposes. In the final version of the algorithm, the user-adjusted parameters will not go into such detail for the structure of the recursive tree. Rather, an average optimum



tree structure will be used for all cases, perhaps with a width vs. depth variable.

### **6.3 Screening Parameters**

A parameter is used to control the stringency of the screening process. This screening parameter is given by the user as a percent of the total points which must pass through the filter. If the value is set at 100, the points are not screened at all. Otherwise, the algorithm goes through and screens all the points at decreasing stringency values, until the percentage of points which pass through the filter is higher than that specified by the screening parameter. This allows the user to adjust the stringency of the screening depending on the effectiveness of the edge detection.

# Chapter 7

## Verification

### 7.1 Test Data Sets

#### 7.1.1 The Three Sets of Test Data

In order to test the algorithm, a driver program was developed which creates four test shapes of any dimensions: a rectangular prism, an ellipsoid, a hollow ellipsoid, and a three-dimensional "L" shape. All of our tests were done on cubic volumes, and the three shapes are referred to as the cube, sphere, and hollow sphere. The outer radius of the hollow sphere is twice the inner radius.

The sphere and cube are used for timing tests, while the "L" is used to verify the orientation of the image, and to ensure that the algorithm can function when the surface of a structure is not closed. The hollow sphere is used to test the recursive functioning of the algorithm: a starting point is picked in the annulus of the sphere, and the algorithm works its way along, detecting the surface on both sides of the annulus.

The driver program also contains a timing feature which measures the CPU time and the actual running time of a process. This is used in timed trials of the algorithm. All trials were run on a SPARCstation 1 computer at the MIT Whitaker College Biomedical Imaging and Computational Laboratory, a machine rated at approximately 1 MFLOP.

### **7.1.2 Accuracy**

The test cases contained only those structures being tested, without noise or any other structures present. As expected, the accuracy of the algorithm was very good, being within two pixels from the surface for all points.

### **7.1.3 Varying the Screening Stringency**

To investigate the effect of varying the screening stringency, the algorithm was tested on a solid sphere of radius 29, in a volume of 64x64x64 pixels. The results for the test are shown on Table 7.1 (timing) and Figure 7-1. The Sd Factor is a measure of the total number of iterations the algorithm had run before the required percentage of points could be accepted by the screening procedure.

Figure 7-1 shows the sequence of outlines of the structure which has been identified by the algorithm. From the unscreened outlines, it is clear that the initial cloud of surface points defines the sphere, with small outcroppings on the top right, and on the left side. Both of these artifacts disappear as the stringency of the screening is increased, as can also be seen in the figure. This is positive evidence that the screening procedure is working correctly.

The vast difference in times between the unscreened run (28.9s) and the fastest screened run (292.7s) is a result of the current sorting algorithm used by the program, which is an insertion sort. An appropriate replacement for this sort is discussed in Section 8, Future Work.

### **7.1.4 Tree Structure Tests on a Hollow Sphere**

To investigate the effect of changing the recursive tree structure of the algorithm, it was tested on a hollow sphere, with the starting point located in the annulus of the hollow sphere. Six trials were then made, with an increasing depth to width ratio for each trial. It was expected that as the tree became more deep and less wide, the algorithm would travel further on average before finding the surface points. This should lead to less of a concentration of points directly around the starting point, as

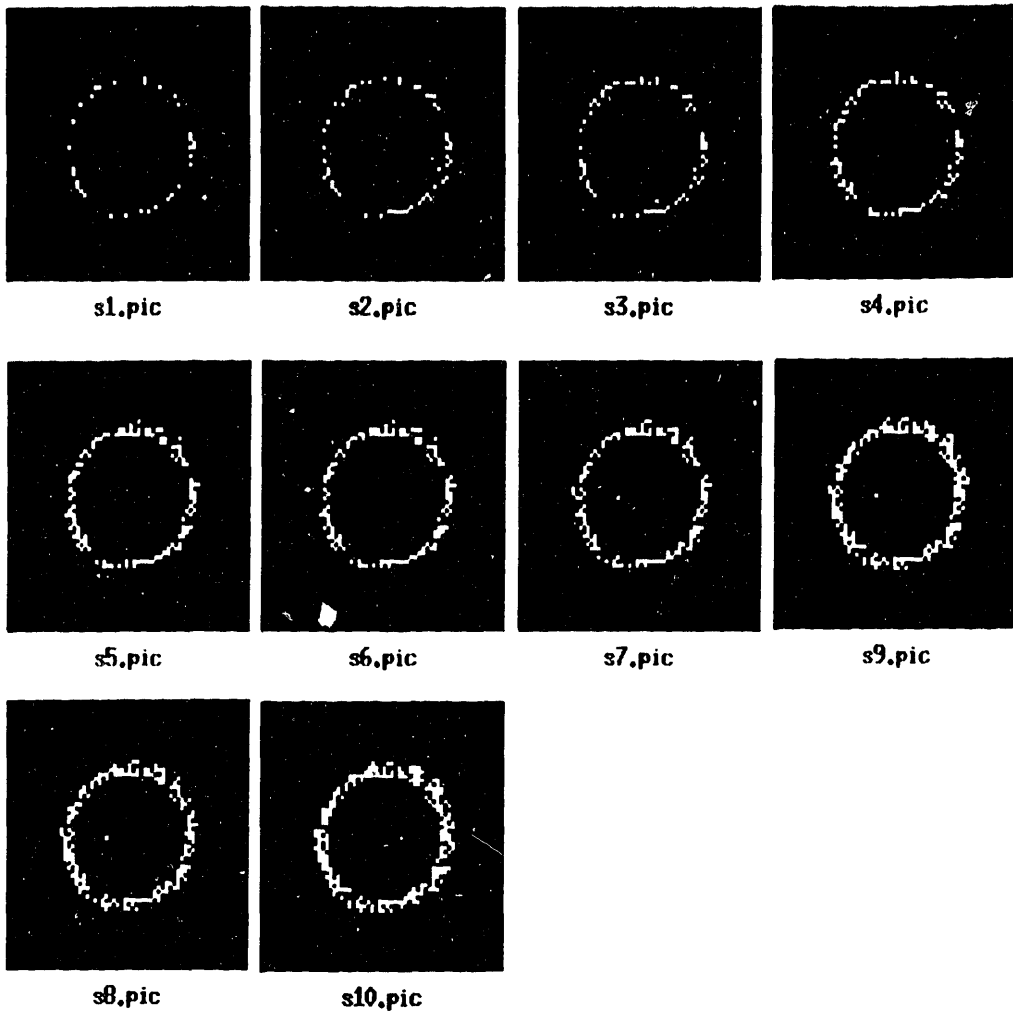


Figure 7-1: Solid Sphere (10,000 Points) 10%, 20% ... 100% Points Passed through Screen

% of Points Allowed Through	Time (CPU seconds)	Sd Factor (0.1 minimum)
10%	292.7	0.4
20%	300.6	0.9
30%	302.5	1.1
40%	305.2	1.3
50%	312.4	1.7
60%	312.3	1.8
70%	318.8	2.2
80%	324.2	2.6
90%	335.1	4.0
NO Screen	28.9	0
Data Volume 64x64x64	Sphere Radius 29 pixels	Starting Point (32,32,32)

Table 7.1: Screening 10,000 Surface Points from the Solid Sphere

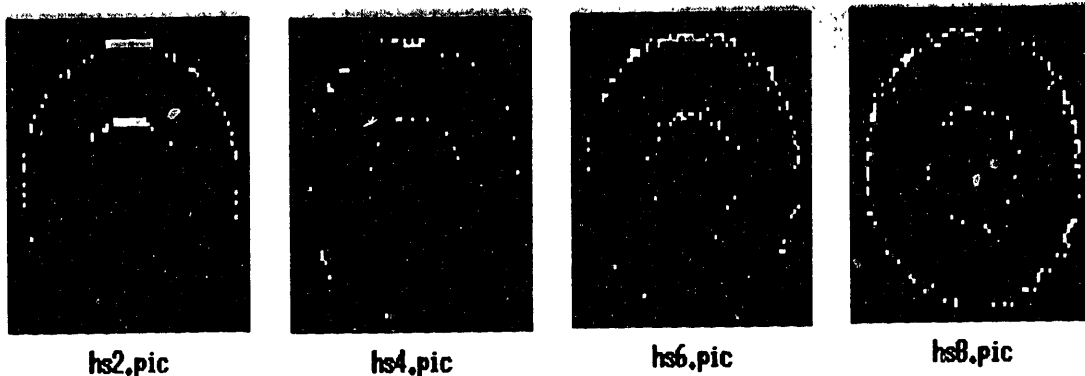


Figure 7-2: Results on Hollow Sphere for Varying Tree Depths

the surface points travel farther away from the initial koosh point.

The tree structure is given in *raysperkoosh/branchfactor* pairs (r,b). Therefore, (100,100) means that there will be 100 rays per koosh, and all 100 of these rays will be used to start child kooshes. If the last level of the tree structure just keeps repeating, (i.e. (100,100),(10,1),(10,1),(10,1)...), it is written as follows: “(100,100), (10,1)...”. Table 7.2 gives the information for the runs shown in Figure 7-2.

Figure 7-2 shows the sequence of trials as the depth of the tree is increased. The starting point in this figure is at (32,5,32), and the slice which is shown is slice 29. The heavy concentration of surface points around the starting point can be seen in

Trial #	Tree Structure	Bottom Level
1	(10000,10),(10000,10)...	0
2	(1000,1000),(5,1)...	2
3	(100,100),(10,1)...	10
4	(10,10),(10,1)...	100

Table 7.2: Legend for the Tree Structures Tested

Dimensions	8x8x8	16x16x16	32x32x32	64x64x64
CPU Time(s)	4.9	7.7	13.6	26.6

Table 7.3: Varying Data Volume Dimensions and CPU Run Times, 10,000 Point Trial on a Solid Sphere. The data shows a linear relation between the length of the cube and the CPU run time of the algorithm, suggesting that kooshball algorithm reacts very favorably to large data volumes.

the first picture, as shown by the two white bars near the top of the figure. The almost even distribution of points around the sphere in the last figure can be seen equally clearly.

### 7.1.5 Size of the Image

Trials were run to test the effect of data volume size on the run time of the algorithm. The results shown in Table 7.3 and in suggest an approximately linear relationship between the radial dimension of the sphere, and the CPU time necessary to find 10,000 points. This demonstrates the scaling capability of the kooshball algorithm for large structures.

### 7.1.6 Number of Surface Points

Table 7.4 shows the number of surface points versus time for the cube test. This trial was done without screening, with a tree structure of (100, 10), (10, 1).... The relation is linear, as can be seen in Figure 7-3.

Number of Points	1250	2500	5000	7500	10000	20000	40000
CPU Time (s)	3.2	4.8	7.6	10.5	13.1	24.3	48.3

Table 7.4: Number of Surface Points vs. CPU Time for a 64x64x64 Data Volume Cube Test (Cube Length = 52)

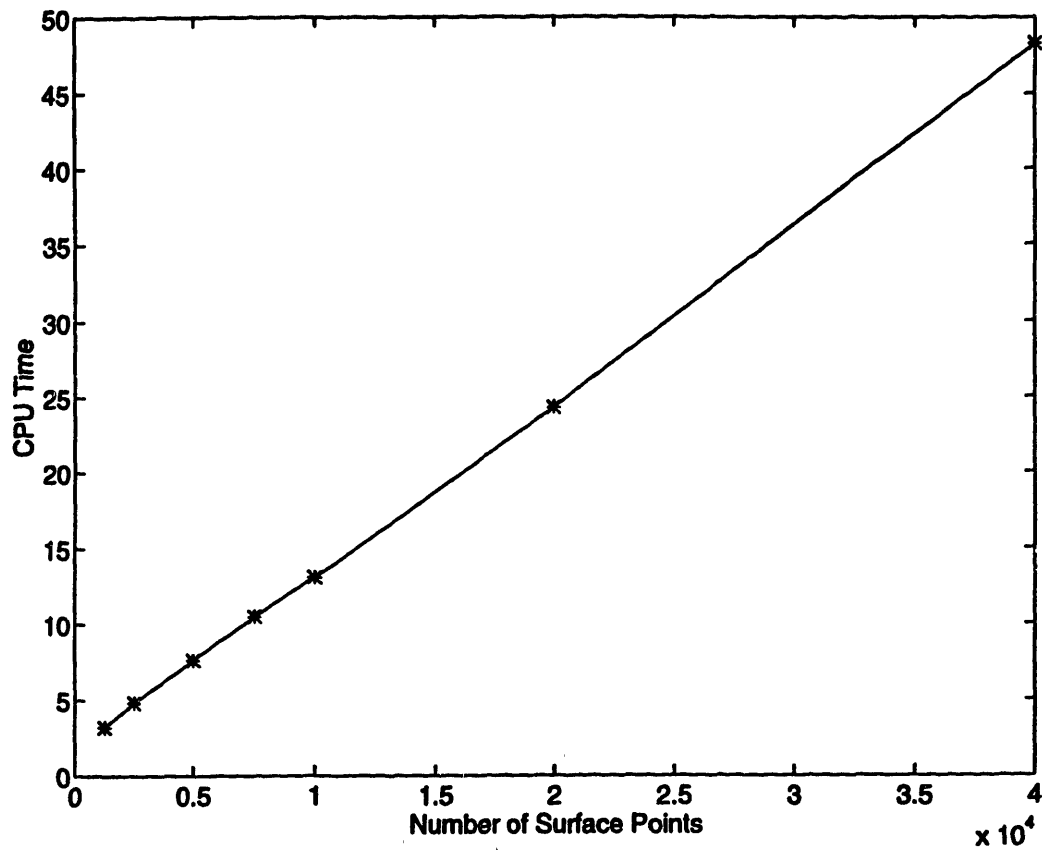


Figure 7-3: Graph of Results for Time vs. Number of Surface Points Trial

## **7.2 Medical Data Sets**

### **7.2.1 The Test Image**

The kooshball algorithm was also tested on a ventricle from an MRI image of the brain. The ventricle was chosen partly because it is the most defined structure in the brain, and would thus yield to the edge detection methods present in the algorithm at this time. It was also chosen because it is convoluted, and makes a good test of the recursive part of the algorithm, that part which allows it to maneuver its way around convex curves.

The data set went through two preprocessing procedures before it was used to test the algorithm. First, a cubic volume of the brain image containing the ventricle was cut out and made into its own data set. This prevented the algorithm from getting distracted by the strong contrast regions at the edge of the skull and on the edges of other structures. Our justification for this preprocessing procedure is that the kooshball algorithm will eventually be used in a semi-automated environment, where the user will define the rough region containing the structure of interest. The kooshball algorithm will then be limited to that data located in a box surrounding the region of interest.

The data was also smoothed in 3D by a low pass filter which removed much of the random noise, thereby taking away another component of the data volume which might distract the edge detection mechanism of the algorithm. Images of four slices of the ROI and smoothed data set are shown in Figure 7-4.

### **7.2.2 Potential Problems for the Kooshball Algorithm**

The main problem which the algorithm meets when processing the medical data occurs because of its simple edge detection mechanism. At this point, the algorithm does not take into account the distance of the edge from the initial point: only the strength of the edge. Therefore, if a ray passes through two surfaces, and the second has a stronger edge, the second, rather than the first surface will be detected.



Figure	Image	Number of Surface Points	Tree Structure	Slice
7-4 Test Data Set (Ventricle)	a	-	-	12
	b	-	-	16
	c	-	-	23
	d	-	-	29
7-5 Segmentation Results	a	40,000	Level 40	12
	b			16
	c			23
	d			29
7-6 Tree Structure Trials	a	20,000	Level 0	29
	b		Level 1	29
	c		Level 2	29
	d		Level 20	29

Table 7.5: Information on Medical Data Trials

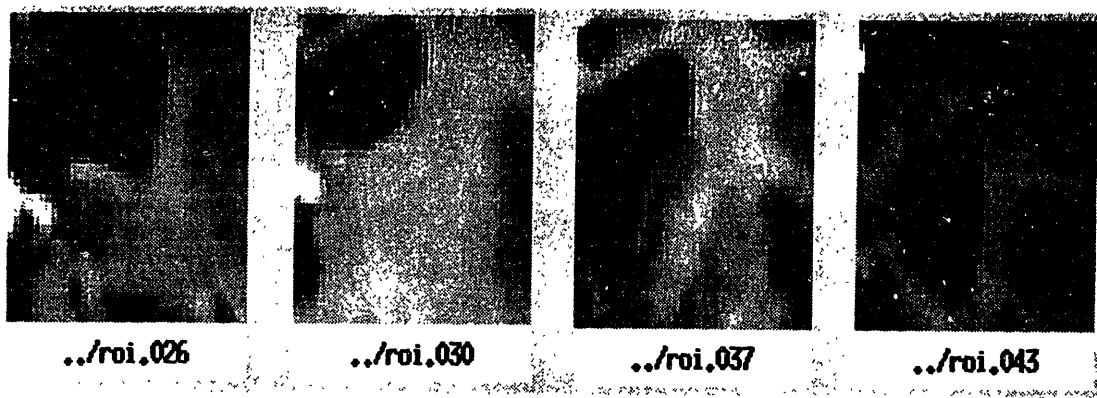


Figure 7-4: Slices of the Smoothed Ventricle Image: slices 13,16,23 and 29

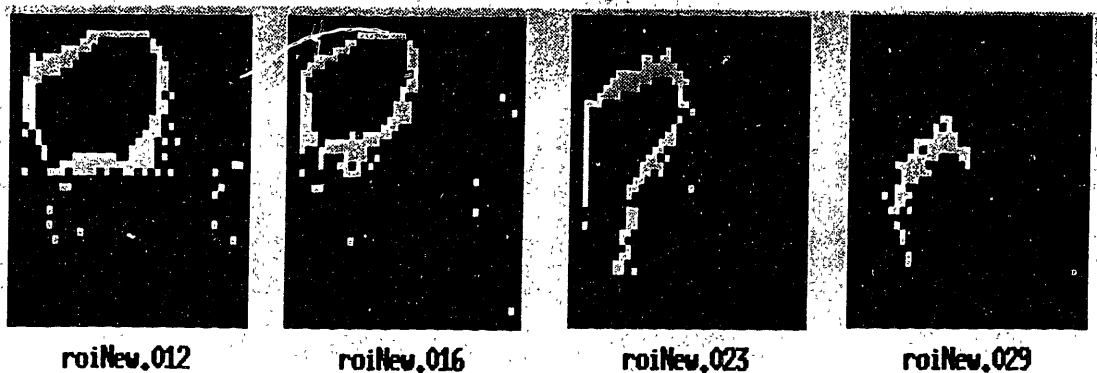


Figure 7-5: 40,000 Point Trial of the kooshball algorithm on the medical test data. Here, slices 13,16,23 and 29 are shown. The starting point is on slice 17. Note that the algorithm does quite well, especially for those slices close to the starting slice.

A second problem occurs because of the contrast between different MRI slices. This contrast in gradient between the slices is sometimes picked up more strongly than the edge of the ventricle itself. The algorithm sees this as a strong edge. Fortunately, however, the edge of the ventricle in this case is strong enough to override this problem. If in future versions of the kooshball algorithm the slice contrast creates difficulty, it can be eliminated by normalizing along the data slices.

### **7.2.3 Accuracy**

The accuracy of the algorithm can be seen in Figure 7-5, by comparing it to the original data in Figure 7-4. Data concerning this run is located in Table 7.5. Though there is some noise in the 40,000 point run, the ventricle is shown very clearly along most of its length. Because the ventricle is convoluted, the algorithm does not perform very well on slice 29. The reason for this is investigated in the tree recursion portion of this section. What Figure 7-4 clearly shows is that the algorithm can perform successfully on medical data, with good accuracy.

### **7.2.4 Changing the Recursive Tree Structure for Better Results**

In an attempt to achieve better results for the algorithm, the tree structure was varied, as shown in the runs on Table 7.5. The results from these runs demonstrate the balance between too wide a tree and too deep a tree in a very clear manner. All these runs were performed with 20,000 surface points.

The first run is flat, with all 20,000 points coming from rays emanating from a single koosh. The fact that it outlines some of the surface in slice 29 shown in Figure 7-6 shows that at least some of that surface is visible from the starting point. The second run, which goes down two levels, with 100 kooshes giving 200 surface points each, outlines the surface on slice 29 adequately. This is the optimal tree setting for this slice. When the tree is deepened in trial 3, less of the surface is shown. The fact that the tree is deeper means that there is more chance of leakage affecting the

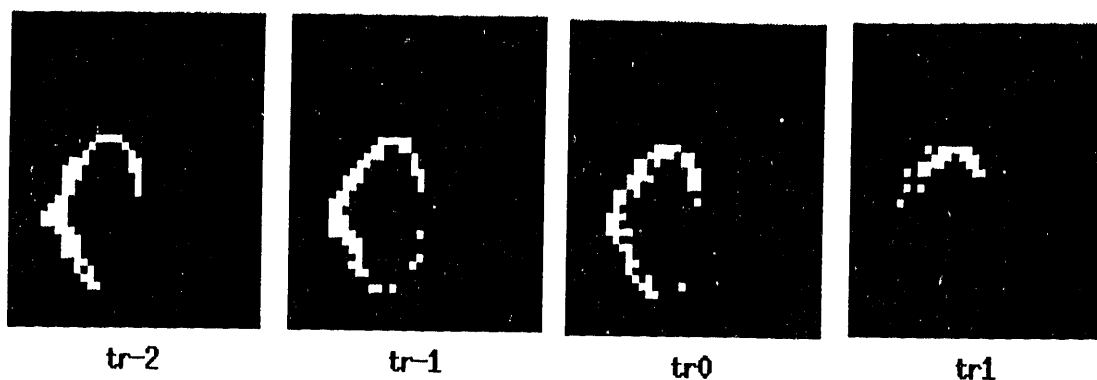


Figure 7-6: Depth of Tree Trials: Slice 29 of a Brain Ventricle Data Set. The depth of the tree in the trials from left to right was 0,1,2,20. The trial with only one tree level yields the best results.

outcome. Further trials showed varying degrees of success in finding the surface on slice 29, but had increased noise. The increase in noise most likely results from the increased effect of one leaked ray, all of whose child kooshes could start from the wrong part of the data volume.

### 7.2.5 Testing the Screening Procedure On a Medical Data Set

As mentioned before, the screening algorithm was not expected to perform well on a convoluted structure such as the ventricle. This proved not to be the case. Though it has trouble with groups of points, the algorithm did quite well getting rid of individual points which were far away from the surface of the structure. However, many of the surface points were erased before this occurred.

## 7.3 Summary of Results

Using four test shapes, we performed qualitative tests on the effectiveness of the screening procedure, and on the effect of varying the tree structure of the kooshball algorithm. The screening procedure works well on the test shapes, producing a more defined result on a solid sphere as the stringency is increased (Figure 7-1). The tree structure was tested on the annulus of a hollow sphere, and it was found that it did

travel around the annulus when the depth of the tree was increased(see Figure 7-2).

Timing trials were performed while first varying the size of the data volume, and then while varying the number of surface points. In the data volume trial on a solid cube, it was found that the CPU time used for the algorithm to complete its task varied linearly with the length of one side of the cube (see Table 7.3). In the surface point trials, the CPU time vs. number of points is also linear (Table 7.4, and Figure 7-3).

A medical data set containing the ventricle of a brain was also used to test the algorithm(Figures 7-4 and 7-5). The algorithm outlined the volume accurately on the whole, though this accuracy dropped off at slices far from the starting point. Tests made by changing the tree structure yielded some better results (Figure 7-6).

The medical test data does show us, however, that the algorithm can outline a real medical structure with a fair bit of accuracy.

# Chapter 8

## Future Work

A number of improvements are scheduled to be made on the kooshball and screening algorithms. The purpose of this initial version of the algorithm was solely to observe its ability to effectively segment structures in MRI images. Now that this ability has been established, further work will enable the algorithm to work faster, perform on a broader range of structures and test cases, and allow us to investigate its potential to a further extent.

### 8.1 Choosing Surface Points Closer to the Starting Point

The most necessary change, which will allow other structures in the brain to be segmented, will take place in the edge detection method. A weighting factor which takes into account the distance between the proposed edge and the origin of each ray is essential for segmenting structures whose edges are not as strong as the ventricle. Though this would be a simple change, work needs to be done on the optimal weighting scheme which would detect the closest edge without being distracted by noise or slight contrast variations close to the initial point.

## **8.2 Creating and Processing Multiple Edge Points on a Single Ray**

A more complicated edge detection scheme, involving the use of more than one proposed edge point per ray is also expected to be pursued. In this scheme, the model would choose the optimal of a set of three edge points from each ray, depending on the strength of each edge, its distance from the center of mass, and its deviation from the strongest edge points given by its nearest neighbours. An entirely different method of screening the edge points will have to be developed in order to accomplish this task.

## **8.3 Replacing the Insertion Sort**

At the present time, the insertion sort which is used in the screening process consumes the vast majority of CPU time in running the algorithm. This is because the sort is on the order of  $n$ -squared, and therefore does not perform well when many surface points are being found. A suitable alternative would be a quicksort: its order of  $\log(n)$  would be much more suitable for the purpose of sorting a large amount of surface point information.

## **8.4 Screening Surface Points on Convex Surfaces**

The screening procedure must be moderated so that it can handle convex shapes; in its present form, it is incompatible with the the recursive part of the algorithm. Modifying the screening procedure might be achieved by taking the center of masses of different sections of the structure, and doing a number of comparisons for each surface point, involving a distance comparison between the surface point and each of these center of masses. In this way, a portion of the structure which was invisible to one center of mass could be screened from the center of mass for the surface points in its section of the structure. An investigation must be made into the efficacy of this

approach.

## **8.5 Time Optimization**

After the algorithm is finalized, the code will be optimized so that it will run at a much faster rate. The optimization of C-code has not been performed at this point because of the changes pending in the algorithm structure.

## **8.6 Incorporation into the WCBICL 3-D Viewing System**

When it is completed, the algorithm, complete with starting seed points and parameters, will be incorporated in existing MIT Whitaker College Biomedical Imaging and Computational Laboratory Software. This software, currently under development, is based on [5] and has been modified by several researchers to serve as a general purpose visualization and segmentation system. The output from the algorithm will be tessellated into a surface, and its volume will be automatically calculated. It is hoped that eventually the algorithm will be used by clinicians to determine the volume and location of structures of interest in medical data in a fast and accurate manner.

# Bibliography

- [1] Tianhorng Chang and C.-C. Jay Kuo. Texture analysis and classification with tree-structured wavelet transform. *IEEE Transactions on Image Processing*, 2(4):429–441, October 1993.
- [2] D. Louis Collins, Terry M. Peters, Weiqian Dai, and Alan C. Evans. Model based segmentation of individual brain structures from MRI data. *Visualization in Biomedical Computing*, 1808:10–24, 1992.
- [3] James D. Foley and Andries Van Dam. *Fundamentals of interactive computer graphics*. The Systems programming series. Addison-Wesley, Reading, Massachusetts, second edition, 1984.
- [4] D.J. Peck, J.P. Windham, H. Soltanian-Zadeh, and J.R. Roebuck. A fast and accurate algorithm for volume determination in MRI. *Med. Phys.*, 19(3):599–605, 1992.
- [5] Joseph T. Samosky. Sectionview: A system for interactively specifying and visualizing sections through three-dimensional medical image data. Master's project, Massachusetts Institute of Technology, Cambridge, MA, May 1993.
- [6] H. Sekiguchi, K. Sano, and T. Yokoyama. Interactive 3-dimensional segmentation method based on region-growing method. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J76D-II(2):350–358, February 1993.



- [7] M.R. Stytz, G. Frieder, and O. Frieder. Three-dimensional medical imaging: Algorithms and computer systems. *ACM Computing Surveys*, 23(4):421–499, December 1991.
- [8] J. Yu, X.; Yla-Jaaski. A new algorithm for image segmentation based on region growing and edge detection. *1991 IEEE International Symposium on Circuits and Systems*, 48(1):516–519, June 1991.