FASTER RECONSTRUCTION ALGORITHM FOR VOLUME ANIMATION

BY KUNDAN SEN

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Deborah Silver

and approved by

____

____

____

New Brunswick, New Jersey

October, 2001

ABSTRACT OF THE THESIS

Faster Reconstruction Algorithm for Volume Animation

by Kundan Sen

Thesis Director: Professor Deborah Silver

Volume graphics refers to the branch of computer graphics that deals with the realm of volumes, or 3-dimensional objects, such as the data produced from MRI, CT and ultrasound imaging. The objective of volume animation is deforming such datasets to produce a sequence of datasets that imbibe into it the sense of motion.

Unlike conventional computer graphics, the realm of volume animation is still in the development phase. This thesis describes the volume animation pipeline developed mostly in Vizlab. Two important stages of the pipeline are discussed in detail - the reconstruction process and the generation of a proper color mapping chart - which were developed and enhanced as a part of this thesis.

Acknowledgements

I would like to thank my research advisor, Prof. Deborah Silver, for her guidance and support throughout my study as a graduate student. My special thanks are due to Nikhil Gagvani, Arindam Bhattacharya, and Tilottama Roy for their useful suggestions to my work. Last but not the least, I would like to acknowledge my colleagues at Vizlab and friends at Rutgers for their love and support during my studies.

Table of Contents

Abstract ii

Acknowledgements iii

List of Tables viii

List of Figures ix

1. Introduction 1

1.1. Motivation 1

1.2. Previous Work 2

1.3. Overview of Material 4

2. The Volume Animation Pipeline 5

2.1. Creating the Volume 6

2.2. Skeletonization 7

2.3. Connecting the Skeleton 8

2.4. Adding Motion Capture 9

2.5. Post-Motion Capture Steps 10

2.6. Reconstruction 11

2.6.1. Sampled Reconstruction 11

2.7. File Formats 12

2.8. Use of the Volume Animation Pipeline 13

3. Enhancements to the Pipeline 26

3.1. Faster Reconstruction 26

3.1.1. Current Approach 26

3.1.2. Disadvantages of the Approach 28

3.1.3. New Reconstruction Algorithm 29

3.1.4. Scan-Filling a Sphere Using Double Bresenham's Algorithm 30

3.1.5. Parametric Equations to Reduce Matrix Multiplication 32

3.1.6. Faster Relaxation Rule 32

3.1.7. Code Level Optimizations 33

3.1.8. Timing and Profiling: Results of the Optimizations 33

4. Additional Functionality 36

4.1. Creating the Colormap 36

4.2. Sorting the Colormap 37

4.3. Segmentation 40

4.4. Reconstruction Modes 41

4.4.1. Binary Reconstruction 41

4.4.2. HotSpot Reconstruction 42

4.4.3. Mottled Reconstruction 42

4.4.4. Sampled Reconstruction 42

5. Results 49

5.1. Execution Time Improvements 49

5.2. Example 1: The Visible Human Dataset 50

5.3. Example 2: The Colon Dataset 51

5.4. Example 3: The Sample Cube Volume 52

5.4.1. Testing Aliasing Effects 53

5.4.2. Anti-Aliasing by a Gaussian Kernel 53

6. Conclusions and Future work 64

Appendix A. User Manual 65

A.1. AddHeader.sh 65

A.2. AddSkel2Vol 65

A.3. AlphaFactor 66
A.4. Bob 66
A.5. combineToBinaryVol.sh 66
A.6. connect2identitytskel 67
A.7. connect2tskel 67
A.8. Convert2Binary 68
A.9. convertall.sh 68
A.10.Count 69
A.11.CropVolume 70
A.12.euclidskel 70
A.13.Fill 71
A.14.FillSection 71
A.15.FitVolume 72
A.16.flip 72
A.17.float2uchar 73
A.18.GenerateHeader 74
A.19.getskel 74
A.20.getVoxelCount 74
A.21.Icol 75
A.22.InterpolateAlpha 75
A.23.InvCMap 76
A.24.ivbbox 76
A.25.ivskeladjust 76
A.26.meltMan 77
A.27.MergeMaps 78
A.28.Merger 78
A.29.multiVolumeMelt 79
A.30.NewSort 80
A.31.OneVolume 80

A.32.Peek 81
A.33.polyr 81
A.34.Poke 81
A.35.Reconstruct1 82
A.36.ReconstructEuclid 83
A.37.remapVolume 84
A.38.ReverseData 84
A.39.rmhdr 84
A.40.Skeleton.tcl 85
A.41.Skelselect 86
A.41.1.Selecting the Articulate Skeleton 87
A.41.2.Selecting the Root Node 88
A.41.3.Selecting the Skeleton Points to Join the Selected Root Node. 89
A.42.SortTskel 90
A.43.teleportMan 91
A.44.TiffToIfl 91
A.45.TightBounds 91
A.46.Toolbar.tcl 92
A.47.View.tcl 93
A.48.Vol2Vtk 93
A.49.VolumeSuperDiff 94
A.50.Vtk2Vol 94
A.51.Zap 95
Appendix B. Troubleshooting 96
References 100

List of Tables

2.1. Comparison of Reconstruction Accuracy With Identity Transform 12

2.2. Comparison of Reconstruction Accuracy With Frame12 of Jump Sequence . 12

3.1. Code Profiling: Comparison of Various Algorithms and Optimizations . . . 35

5.1. Properties of the Colon Dataset 52

5.2. Properties of the Aliascube Dataset 53

List of Figures

2.1. The Volume Animation Pipeline 18

2.2. The Skeleton Points, Shown Inside the Surface Hull of the Visible Human Volume, in 3 Different Thinness Values- 1.8, 1.5, and 1.3 19

2.3. Root Nodes Selected In The Proper Order, Skelselect Started Up With The Skeleton Points And The Bone Skeleton, Partially Connected Skeleton . . . 20

2.4. Final Connectivity Information Shown At 3 Different Thinness Values Using Skelselect 21

2.5. Partial Connection of a Foot Node in Skelselect 22

2.6. Inventor Files Converted From The Motion Capture Dxf Files 23

2.7. Sequence After Extracting The Articulated Skeleton Points 23

2.8. Sequence After Connecting Skeleton Points And Reconstructing Spheres . . 23

2.9. Splatter of The Various Forms of Post-Animation Representation 24

2.10. The Volume Animation Pipeline 25

3.1. Scan-filling One Line from Another, Using Parametric Equations 32

4.1. The Volume Animation Pipeline 44

4.2. Butchershop.gif - Showing Range of Colors in the Dataset 45

4.3. The Original Colormap Extracted from butchershop.gif 46

4.4. Colormap Sorted by Gradient with Penalty. Note the Discontinuities. . . . 47

4.5. Final Results from newSort 48

5.1. Reconstruction Times for the Left Leg of the Visible Human 50

5.2. Reconstruction Times for the Sample Cube Dataset 50

5.3. Reconstruction Times for the Half-Sized Visible Human 51

5.4. Average Speedup with the New Reconstruction Program 52

5.5. Cumulative Speedups Obtained by the Various Optimizations to the Reconstruction Program 53
5.6. Visible Human Dataset, a Frame From the Jog Sequence 54
5.7. Visible Human Dataset, with Hands Stretched Outwards 55
5.8. Visible Human Dataset, a Frame from the Wave Sequence 56
5.9. Internal Details in Visible Human Animation Volumes 57
5.10. The Dataset, as Obtained, After Removing Blank Envelope 58
5.11. Surface of the Colon Dataset, Extracted After Segmentation 58
5.12. Articulated Skeleton and a Few Skeleton Points of the Colon 59
5.13. Final Connectivity Information of the Colon Skeleton 60
5.14. The Colon Dataset, After Uncoiling with the Animation Pipeline 60
5.15. Cut-away View of the Uncoiled Colon 61
5.16. Cube Volume to Test Aliasing Results, Viewed from Front Top Right. Volume size is 128x128x128 61
5.17. Euclidean Skeletons of the Aliascube, at Thinness Levels of 1.50, 1.35 and 1.10 61
5.18. The Transformed Reconstruction of the Sample Cube 62
5.19. Slices from the Transformed Reconstruction of the Sample Cube Dataset . . 62
5.20. Slices in the Y-Z Planes, different X-values, from the Cube Reconstruction, Using Smoothing Kernel of 1 (No Smoothing), 3, and 5 62
5.21. Subsections of Figure 5.20, Magnified 8 Times. Note that Some of the Minor Artifacts Were Caused by the Magnification in **xv** 63
A.1. Skeleton.tcl - User Interface for the Old Volume Animation Pipeline 86
A.2. Toolbar.tcl - User Interface for Volume to tiff Converter 92

Chapter 1
Introduction

The goal of computer graphics is to produce 'realistic images' using a computer, i.e. create a synthetic model, 'render' that model, and somtimes simulate the environment around that model. Animation is the art of producing dymanic images. Animators take synthetic 'computer models' and move them around, creating 'animations' or movies.

Volume graphics refers to the branch of computer graphics that deals with the realm of volumes, or 3-dimensional objects. Unlike conventional computer graphics, which generally limits itself to surface or 'polygonal' models of 3-dimensional objects, volume graphics deals with an entire 3D object, the outside and the inside. Examples of volume models include datasets from MRI, CT, and ultrasound imaging. Animating these models, i.e. volume animation, is the focus of this thesis.

1.1 Motivation

Volume animation follows the mainstream computer graphics animation pipeline and includes modeling, manipulation and rendering. Modeling refers to the process of creating the data and the design of adequate data structures to hold the data. Manipulation refers to altering the model in order to instill into it the sense of motion. Rendering refers to the process of taking in the manipulated data and creating the final 2-dimensional image out of the information. Many 2D images can then be put into a movie format.

Conventional computer graphics has developed into a very standardized process where every step exists as a well defined sub-process. However, the field of volume animation has seen little work to date. There has been isolated work in one or more sub-processes yet there is an absence of a well defined pipeline guiding the user through all the necessary steps from aquiring a volume dataset to the final objective of producing a volumetric animation

sequence. The reasons for this include:

1. Dataset sizes are immense,
2. Fast rendering algorithms are relatively recent (1980's), and
3. Sampling devices to collect the volume data are expensive.

In this thesis, we present the volume animation pipeline developed in the laboratory for Visiomertics and Modeling at the CAIP Center. Our focus will be on one component, reconstruction, and the enhancements to the algorithm for this component.

1.2 Previous Work

The goal of a volume animation pipeline is to generate a sequence of 2D images that show motion of a 3D volume, starting from the dataset of the 3D volume and motion capture sequences. As mentioned earlier, the goal can be broken down into the following steps:

- Modeling: create the 3D volume from 2D sliced data, and design appropriate data structures
- Manipulation: Alter the data structure so as to exhibit a sense of motion
- Reconstruction and rendering: Re-generate the volume in the new deformed position and render the final environment into a 2D image.

Although publications on an entire volume animation pipeline are not available, a lot of work has been done on the individual steps that make up the pipeline. Some of the concepts that are relevant to this thesis are described below:

- **Octree Encoding** [2] This work describes the octree data structure that is designed for storing and working with volumes. Arbitrary 3-D objects can be represented to the desired resolution in a hierarchial tree structure where each node has eight children. The octree is particularly suitable for sparse volumes, but when the volume size becomes huge, and is sufficiently filled, the memory requirement of this representation surpasses by far that of the array based approach. Moreover, the octree does not provide as fast access to data in neighboring cells as the array approach.

- **Marching cubes algorithm** [4]: An isosurface is a collection of polygons that make up the surface hull of a volume. The marching cubes algorithm is a key algorithm in the generation of isosurfaces. The isosurface plays an important role in the modeling phase - it's by comparing this isosurface to the cloud of skeleton points that we can make the best possible selection of points for the articulated skeleton. The articulated skeleton is the basis for all the transformations that the volume has to undergo in the motion sequence. The Marching Cubes algorithm allows us to build this isosurface from a binary segmented dataset by converting each 'cell', comprising of a 2x2x2 grid of adjacent voxels, that lies on the boundary of the sampled volume, into one or many triangles that make the closest representation of the boundary.

- **Skeletonization process** [15][19] This work explains the research and development involved in creation of the process of skeletonization that our volume animation pipeline relies heavily on. Skeletonization, as defined by the work, is the process of thinning a volume down to a number of key points with associated 'weights', or 'Distance Transforms'. The quality of skeletonization, i.e. the density of the skeleton, dominates in determining the accuracy of the reconstruction process. The articulate skeleton, the stick figure to which the motion capture data is applied, is a subset of these skeleton points.

- **Volume Animation** [16] [23] These publications describe in detail the pipeline designed and implemented at our lab. The central idea of using a skeleton model to perform volume animation is discussed. This work also lays down the basic theory of reconstruction as applicable to our work.

- **Collision detection** [22] This paper illustrates the theory of collision detection based on our skeleton model of volumes. The animations illustrated in this work have been produced by the reconstruction process discussed and improved upon in this thesis.

- **Reconstruction Filters** [11] The research conducted by this group was aimed at obtaining a density function that defines the entire volume, given a discrete set of samples to start from. The definition of reconstruction, as used by the authors, is the process capable of regenerating the volume by interpolating between the discrete

sample points. Note that this definition is different from our use of the term. The authors discuss several filters that generate volumes with smoother sampling, such as cubic, trilinear, pass-band, windowed sinc, and a host of others. Properties of good filters, and possible errors in sampling, are discussed. This paper gave us the insight to include basic smoothing kernels into our reconstruction process.

- **Keyframe animation system** [21] This work presents a keyframe animation system where a 3-D world containing the volume of interest as well as its surrounding environment is represented as a voxel model and is animated. By keyframe animation, the authors refer to the process of creating an animation sequence by hand-crafting a few frames that define the range of motion, and interpolating the rest of the frames. The application developed, Sirpi (Sculpting Interface for Rapid PrototypIng), is a voxel based approach to Computational Solid Geometry (CSG) sculpting. The authors have shown implementation of basic volume sculpting tools, namely the cuboid, cylinder, cone and sphere, and of arbitrary shaped tools formed by the use of one or more of the fundamental tools on a volume, by use of Minkowski operators. They illustrate the use of stack-of-bits approach and the reference count approach, which is somewhat on the lines of our distance transform stencil approach, in determining the correct sampling lookup for reconstruction of a particular voxel.

1.3 Overview of Material

This thesis continues with a description of the current working pipeline that was developed in Vizlab. Chapter 2 describes the existing volume animation pipeline developed in Vizlab. Chapter 3 describes the details of the major additions to the pipeline that were carried out as a part of our research. Chapter 4 describes in detail some specific implementation issues in the new pipeline. Chapter 5 presents some of the results of the improved pipeline.

A comprehensive user manual of all the utilities and programs developed and/or used during this research has been provided in Appendix A. Some common troubleshooting hints are mentioned in Appendix B, to help the new user get along with using our pipeline.

Chapter 2

The Volume Animation Pipeline

The volume animation pipeline developed in Vizlab has several steps that have to be carried out in order to produce a successful and appealing animation sequence. Figure 2.1 shows the flow diagram of the pipeline. The major stages of the pipeline are as follows:

- **Pre-processing the dataset:** The 3D Datasets we worked with were typically received as a set of 2D images, each of which contain the data at a particular height of the 3D object. In the first step of the pipeline, we process the dataset from its raw format to the format suitable for our animation process, a 'volume'. Any scaling and/or filtering of the data is done here. The colormap for the volume is extracted for sampled datasets.

- **Skeletonization:** The next step involves thinning the volume to reduce the voxel count. Only the voxels that are most important to the shape are retained. A small number of these points are then selected as the 'bone skeleton', a stick figure representation of the shape of the volume.

- **Connectivity:** Every skeleton point in the cloud of points obtained above is connected to a segment ('bone') of the bone skeleton. This is carried out such that when a bone moves, the points attached to it move along with it.

- **Animation:** The bone skeleton is handed over to the animator, who uses conventional animation packages like Maya to apply motion capture to our skeleton. Once the motion has been applied, the animator exports the transformations that each segment of the bone skeleton undergoes, in each frame of the animation. The bone skeleton, after being given transformations for each frame, forms the 'articulated skeleton'.

- **Reconstruction:** Now that we have the articulated skeleton, we pass the series of skeletons to the reconstruction program, which generates a series of volumes that represent the motion capture sequence.

- **Rendering:** In this last step of the volume animation pipeline, the series of new volumes generated by the reconstruction program are rendered into 2D images. These images are then made into a movie.

We shall now look into each of these stages, and sub-stages, in detail.

2.1 Creating the Volume

Volumetric datasets can be acquired in many ways, for example, from MRI, CT, and ultrasound imaging, or from a sculpting program, like Carpeaux [17].

One famous dataset is the visible human dataset, from the National Library of Medicine [1]. This dataset will be used in this thesis as an example. The visible human dataset is a set of 24-bit color tiff images, from cryogenic slices of a male. These images were at a resolution of 1740x1012, with 1879 slices covering the height of the man, from head to foot. These slices can be combined, by stacking up vertically, in order to create a 3D volume. For the coordinate system we followed throughout our animations with this volume, the origin was the top left front corner, the x-, y- and z- axes being left-to-right, front-to-back, and top-to-bottom, respectively. Note that the 'left' of the volume, by convention, implies the section of the volume to the left of the viewer, and not with respect to the position of the visible man in the volume. In our case, the visible man is facing the viewer, so the left of the volume is the right hand of the man.

It is clear from the size of the images that the volume that was created upon stacking was roughly a cube, the width being comparable to the height. This is very unlike the dimensions of a human body. This discrepancy arose because the sampling of the dataset is much higher in x- and y- dimensions, i.e. within the cross sections, since these are high resolution photographs of the slices. The slices themselves could not be made as thin as the resolution along them, so the resolution in z-axis was limited to 1879. This meant that the

volume had to be re-sampled in x- and y-dimensions, in order to make it look like a human. Because of the large size, we subsampled the dataset to a resolution of 580x337x1879.

2.2 Skeletonization

Skeletonization is a thinning process that reduces a volume to a simpler shape while preserving the essential features of the original volume. The thinned version of the volume is called the skeleton. Skeletonization of a binary-segmented dataset (containing only 0's and 1's) produces a cloud of points. Details of this process are mentioned in prior work carried out in Vizlab, [16], [15], [19], and [23].

Each of these points has an associated radius (distance transform, DT) and thinness value. If a sphere of radius equal to the DT is scan-filled at each skeleton point, the original shape is retrieved. However, this process is lossy, and the accuracy is directly proportional to the number of skeleton points, which in turn is determined by the thinness parameter [15] in the skeletonization process. Details of the skeletonization program, '**Skelselect**', are discussed in chapter 4 of this thesis, and in [24].

A skeleton of a volume can be made very thin (lesser number of points) or very thick (huge number of points). The thinner the skeleton, the less information it retains, and so the poorer the reconstruction. The thinness parameter determines the thickness of the skeleton.

Current skeletonization programs in Vizlab produce a skeleton of multiple resolutions, allowing the user to choose how loss-less the process of skeletonization should be. Figure 2.2 shows how the surface model of the volume is matched up with the skeleton in order to assist in selection of points for the bone skeleton. Figure 2.3 has 3 parts: (a) the display after **skelselect** is loaded up with the skeleton points and the bone skeleton, (b) the bone skeleton after the root traversal order has been selected, in the *Select Root* mode of the tool, and (c) a partially connected skeleton.

2.3 Connecting the Skeleton

The points generated by the skeletonization program described in the previous section are unconnected. For animation, a connected set of nodes is needed. Establishment of a connection between the points helps us move groups of points together. For example, in the visible human dataset, all the skeleton points of the hand have to move together, since the hand is one single segment. For standard animation programs, like Maya or Character Studio, 'joint' values must be chosen. Our 'bone skeleton' translates to the joint framework required by these programs. The joints are also placed in a hierarchy, so that transformations of the joints higher up in the dependency graph are transferred to all the joints that depend on them. In simpler terms, moving the shoulder would move the upper arm, which in turn would cause the lower arm and the fingers to move along with it. The combination of the joints and the hierarchy help us keep the skeleton connected, irrespective of the transformations applied by the motion capture data.

The multiple level-of-detail skeleton, even at the lowest resolution, is too huge to be worked upon in standard animation packages and add motion capture to. For this purpose, we select a set of points that are crucial to defining movement in the skeleton and form an 'bone skeleton'. For the visible human project, these points are chosen as the joints in the body, in correspondence with the placement of sensors in the motion capture data we used for the project. To ensure proper correspondence of the points and the joints in the body, the cloud of skeleton points was placed within the isosurface hull of the visible man, using **skelselect**.

Once the bone skeleton has been formed, each skeleton point in the multi-resolution skeleton has to be connected to one and only one segment ('bone') of the skeleton. This is also done using **skelselect**. An approximate automatic connectivity can be accomplished by the minimal spanning tree algorithm, since each skeleton point should be connected to the bone that is the nearest to it, so long as no boundary is crossed in the path from the point to the bone. The automatic connectivity works fine for geometric objects, but for complicated shapes like the visible man, division of the body is better done with visual verification than by a minimal spanning tree method. The automatic method may be used

for obtaining a general connectivity, but areas where the bone skeleton forks - shoulders, neck and pelvis - require visual fine-tuning for best results.

Our file handling convention names the bone skeleton as a '*.connections' file, and the final output of the skelselect program (containing the bones and a list of points corresponding to each bone) as a '*.xxx.connect' file, xxx's being filled in by the thinness value at which the connections were saved.

The connection of the skeleton points to the articulated skeleton is a complicated, time-consuming process. The segments of the articulated skeleton are selected one at a time, and all the skeleton points in the skeleton cloud that might belong to this segment are connected to this 'bone'. The selection pointer of **skelselect** allows selection in the form of an axis-aligned cube. Most of the segments of the visible human can not be exactly contained within a cube of any size. This makes it necessary to perform the selection of the arbitrary selection by repeated selection of smaller cube sections.

Unless special attention is paid to connecting the shoulders, pelvic joints, and the knee, these regions are susceptible to breakage. This is because our reconstruction algorithm does not compensate for stretching of the skin, and the rotations of the limbs are faked by rotation about the joints. Thus, a large degree of rotation will lead to exposure of the flesh within, since the skin will fail to cover up the rotation. In many cases, there will appear a visible breakage in the reconstructed volume - due to excessive rotation, improper connectivity, or poor choice of points for the articulate skeleton. Each such case is different, and has to be handled differently by the user.

Figure 2.4 shows the final connectivity information for the skeleton at three different thinness parameter values. Figure 2.5 shows partial connectivity of one foot of the skeleton, using the cube selection cursor of the program.

2.4 Adding Motion Capture

At this point of the pipeline, the articulated skeleton is passed down to the animator. The motion capture sequences we have used are typically based on a polygonal abstraction of the human figure, with each segment being represented as a box. The animator attaches each

joint in the articulated skeleton to its corresponding joint in the motion capture mannequin and applies the motion to this new structure. At each frame of the sequence, the new coordinates of the joints of the articulated skeleton are stored (as a *.dxf file).

2.5 Post-Motion Capture Steps

Each of the frames produced by the motion capture data have the coordinates for all the points in the articulated skeleton for that particular frame. These coordinates are extracted, compared to the original points, and the transformation matrix that will replicate this position change is back computed. This transformation matrix is stored in a '*.trans' file. It may be necessary to apply a global transformation to the entire sequence, in order to match up the motion capture coordinate system to the skeletonization/reconstruction coordinate system. The script 'convertall.sh', discussed in Appendix A.9, takes care of the entire process of converting the frames to tskel files. Section 2.7 describes the file formats used in this work.

Figure 2.6 shows the motion capture sequence as extracted from the dxf files that were exported by the animation software. Figure 2.7 shows the same sequence after the articulated skeleton points were extracted, and each segment length adjusted to be the same as the original articulated skeleton. The differences in segment lengths may occur because of compression or stretching of segments in the motion capture sequence, which we do not accomodate for in the current version of our animation process. Figure 2.8 shows the surface reconstruction of the same sequence, after every skeleton point attached to each of the segments was blown up into its corresponding sphere model. Figure 2.9 shows, for a single frame, how the different stages of post-reconstruction change the amount of information stored in the file.

At this point, the transformation matrix at each bone and the list of points attached to each bone have been established. We shall now discuss the process of reconstruction.

2.6 Reconstruction

Reconstruction refers to the process of re-building the original shape in the new pose. In the case at hand, this means the visible human has to be transformed from its default pose to each of the poses in the motion capture sequence. There are several modes of reconstruction, namely binary, mottled, hotspot and sampled, each having its own purpose. The focus of this thesis shall be on sampled reconstruction. The other modes are discussed in detail in Chapter 4.

2.6.1 Sampled Reconstruction

The ultimate objective of the reconstruction process is to obtain the sampled human in the deformed pose. This implies that not only should the new volume retain the shape information from the original volume, but it should also retain all the color information as well. To further elaborate, all of the interior voxels must be maintained. Sampled reconstruction produces this required objective.

Sampled reconstruction requires the original volume as a lookup and the *.tskel file (see next section for file formats) for the bones, skeleton points and the transformations. The tskel file contains the list of bones in the bone skeleton, the transformations associated to each of these bones, and the skeleton points attached to each of the bones. For each skeleton point present in the tskel file, the reconstruction program reads in the point coordinates and the DT value. The coordinates are transformed by the transformation matrix associated with the bone to which this point belongs. A sphere is located in the destination volume with these transformed coordinates as the center, and the DT value as the radius. This sphere is now scan-filled: each voxel encountered in the scan-fill is multiplied by the inverse of the transformation the skeleton point in hand underwent. This gives us the coordinates of the corresponding voxel in the original, undeformed (lookup) volume. The data from the lookup voxel is then copied to the reconstruction voxel.

Voxels may fall within boundaries of multiple spheres. When this occurs, we apply a rule of relaxation - the voxel is assigned to the skeleton point that is closest to it. Since the closest skeleton point is not known at the start of the program, we compute the distance

Table 2.1: Comparison of Reconstruction Accuracy With Identity Transform

Metric	Thinness Value				
	1.03	1.40	1.74	2.22	2.48
Number of skel points	29777	13068	5706	1282	378
No. of voxels in reconstructed volume	1621724	1578988	1536805	1434874	1302138
% voxel loss from 1.03 reconstruction	0	2.63	5.23	11.52	19.70

Table 2.2: Comparison of Reconstruction Accuracy With Frame12 of Jump Sequence

Metric	Thinness Value				
	1.03	1.40	1.74	2.22	2.48
Number of skel points	29777	13068	5706	1282	378
Reconstruction time on cyan	48m 51.10s	24m 42.85s	12m 5.61s	3m 10.88s	1m 8.98s
Number of voxels in reconstructed volume	1610541	1603810	1591507	1537901	1434522
% voxel loss from 1.03 reconstruction	0	0.42	1.19	4.52	10.93
"Hotspots" - maximum redundancy in reconstruction	34	32	25	18	13

between the voxel and the skeleton point, for every sphere the voxel belongs to. The voxel is filled with the new lookup value only if this distance is less than the distance from the voxel to the previous skeleton point that filled it. At the start of the program, all distances are set to an arbitrarily large value. This ensures that the first sphere that touches a voxel always fills it. Spheres that touch the same voxel later in the execution of the program have to conform to the relaxation policy. Implementation details are dealt with in the next chapter.

2.7 File Formats

Numerous file formats are used in the volume animation pipeline. These include:

- ***.vol:** Volume file. Contains the raw data in binary format. Data at each voxel is stored as an unsigned char value, in the range [0-255]. The voxel data is stored in increasing order of x, then y, and then z. Both the original volume and the animated volumes are present in this format.

- ***.skel:** Skeleton file. Contains the cloud of points created by the skeletonization process. The points are unconnected. Each dataset typically has a few skel files, at different resolutions.

- ***.mskel:** Multi-Skeleton file. Same as skel file, with an additional column of thinness value included for each skeleton point. Each dataset has a single mskel file, created at a very low thinness (i.e. with as much detail as possible).

- ***.connections:** Bones file. Contains the list of the bone segments defined for the bone skeleton. Also defines the hierarchy of these bones. Each dataset typically has a single connections file.

- ***.connect:** Connected skeleton. Contains the bone skeleton, along with a list of skeleton points for each bone. Each skeleton resolution produces one connect file. Usually, for a particular animation, the skeleton resolution is fixed, so we deal with a single connect file.

- ***.trans:** Transformations file. Contains the matrix transformations the bone skeleton undergoes in order to form the frames of the animation. Each frame of the animation has its own trans file.

- ***.tskel** Tskel file. Contains a merge of the information contained in the trans and the connect file - i.e., has every bone segment, the transformation associated to this bone segment, and the list of skeleton points attached to this bone segment. As with the trans file, each frame of the animation has its own *.tskel file.

2.8 Use of the Volume Animation Pipeline

The programs that make up the volume animation pipeline, and the process flow through the use of these programs, is shown in Figure 2.10. In this section, we shall briefly go over the steps required to produce an animation sequence. Syntax details of the use of these programs are mentioned in the user manual in the appendix of this thesis.

The pipeline will be discussed with the Visible Human Dataset as the example dataset. Details of the dataset have been discussed in Section 5.2 of this thesis.

1. **combineToGrayVolume.sh** The Visible Human Dataset is obtained as a series of images, each defining one plane in the Z-axis. This script takes each of these images, converts them into 255-color grayscale images using the SGI utility **convert**. These grayscale images are them stacked end-to-end to form a single volume file. Each voxel in the volume file has data in the range of 0-255, and can be read in as an unsigned character.

2. **MapImages** This stream of the pipeline is used to extract the color information from the original volume, and create a best match of 255 colors that will be used later as the color table for rendering the volumes. This program scans through a sample image, such as Figure 4.2, and extracts the best fitting set of 255 colors.

3. **InvCMap** InvCMap takes in the 255 colors generated by MapImages, all the colors present in the mapping image (in this example, Figure 4.2), and creates a mapping table that maps each of the 24-bit colors present in the mapping image to it's best fit color in the 255-color table.

4. **newSort** NewSort takes in the 255 colors generated by MapImages, and sorts the colors in a visually smooth gradient. This is required to avoid unpleasant color bleeding in the final rendered image. Details of implementation of this program have been discussed in Section 4.2. At the end of this step, we have the final colormap that is passed on to the renderer.

5. **Convert2BinaryVolume**

   Skeletonization requires a segmented volume, with all the points that are considered 'inside' the volume marked as 1, and all the points that are 'outside' as 0. **Convert2BinaryVolume** takes in the grayscale volume generated earlier, and a threshold value passed as a parameter. All voxels that have a value greater than, or equal to, the threshold value are considered to be inside the volume, and marked as 1 in the new binary volume created. All the other voxels are given a value of 0.

6. **Fill**

The binary volume generated in the step above may contain cavities - areas that have voxel values as 0, even though they are contained within the volume. This situation arises when the original volume has values within it that are less than, or equal to, the value of the blank region. For instance, if the air around the volume has a value of 0, the air in the body cavities such as the lungs will also have a value of 0. Presence of such cavities will lead to incorrect skeletonization. The **Fill** program does a convex hull filling of these blank spaces. The program takes in a bounding box, and fills up any cavity that is completely surrounded by filled (i.e. voxel value 1) regions.

7. **euclidskel**

   This program generates the skeleton of a binary volume. Details of the implemetation of this program are discussed in [15, 16, 19, 23]. This program takes the binary volume and it's bounds passed as parameters, and generates an **mskel** file.

8. **sort**

   The mskel file generated above is not sorted in any order. We use the Unix shell sort utility to sort the mskel file in decreasing order of thinness. This way, by extracting a desired set of lines from the top of the file, we obtain a thinner skeleton - since the points that have thinness values lesser than desired are culled from the bottom.

9. **skelselect**

   This is a graphical user interface to allow the user to select a bone skeleton (consisting of key joints within the skeleton file generated above), and establish connectivity of the cloud of skeleton points to this bone skeleton. Details of use of this program is mentioned in the user manual in the appendix of this thesis. Implementation details of this program can be found in [23]. Details of a new version of the tool that has been developed can be found in [24]. The output of this program is a bone skeleton and a **connect** file containing the mapping of the cloud of skeleton points to the bone skeleton.

10. Animation Process

This step involves handing over the bone skeleton to the animator, who applies motion capture to the skeleton using a conventional animation package like Maya. Each frame of the motion capture sequence is now exported from the package as a dxf file.

11. **dxf2iv**

    This program takes in the dxf files generated by the animation package above, and converts them into Inventor file format. This file format is required by the programs that follow to extract the points of the bone skeleton from the motion sequence.

12. **boxes2iv**

    This program takes the Inventor file generated above, and extracts the positions of the joints from our original bone skeleton in each of the frames. This gives us the new coordinates of each joint in each frame of the motion capture sequence, in the form of a series of articulated skeletons.

13. **ivskeladjust**

    Our pipeline currently does not accomodate stretching of bone segments. However, most motion capture sequences have some amount of stretching. **ivskeladjust** compares the length of each bone in the new series of articulated skeletons and corrects the lengths of the segments that were stretched.

14. **ivskelgetrot**

    This program takes in the series of articulated skeletons, and computes the transformation matrices for each joint in the bone skeleton for each frame of the animation sequence. The output of this program is a **trans** file, containing the list of bone points and their transformation matrices, for each frame of the animatio sequence.

15. **connect2tskel**

    This program takes the **trans** file generated above, and the **connect** file generated by the skelselect program, and merges the information. The **tskel** file thus produced for each frame of the animation contains the list of bone points, their transformation matrices, and the list of skeleton points that are attached to each of these bone points.

16. **ReconstructEuclid**

    This program is the main focus of this thesis. Impelmentation details of this program are discussed in Chapter 3 of this thesis. **ReconstructEuclid** takes in each **tskel** file generated above, and the original volume to be used for sampling the correct voxel values, and generates a new volume in the animated pose of the frame being reconstructed.

17. **vtkRender**

    Since volumes are data files and can not be visually seen, we need a rendering tool to generate an image of the volume in the new pose. **vtkRender** takes in volume files in **vtk** format and generates image files after rendering the volumes. Camera angles, cropping distances, colormaps, and other rendering variables are passed as command line parameters.

This thesis will focus on the improvements carried out to the reconstruction phase of the volume animation pipeline. Chapter 3 will elaborate on this topic. In order to improve the performance of the entire pipeline, additional enhancements were carried out, which will be described in Chapter 4.

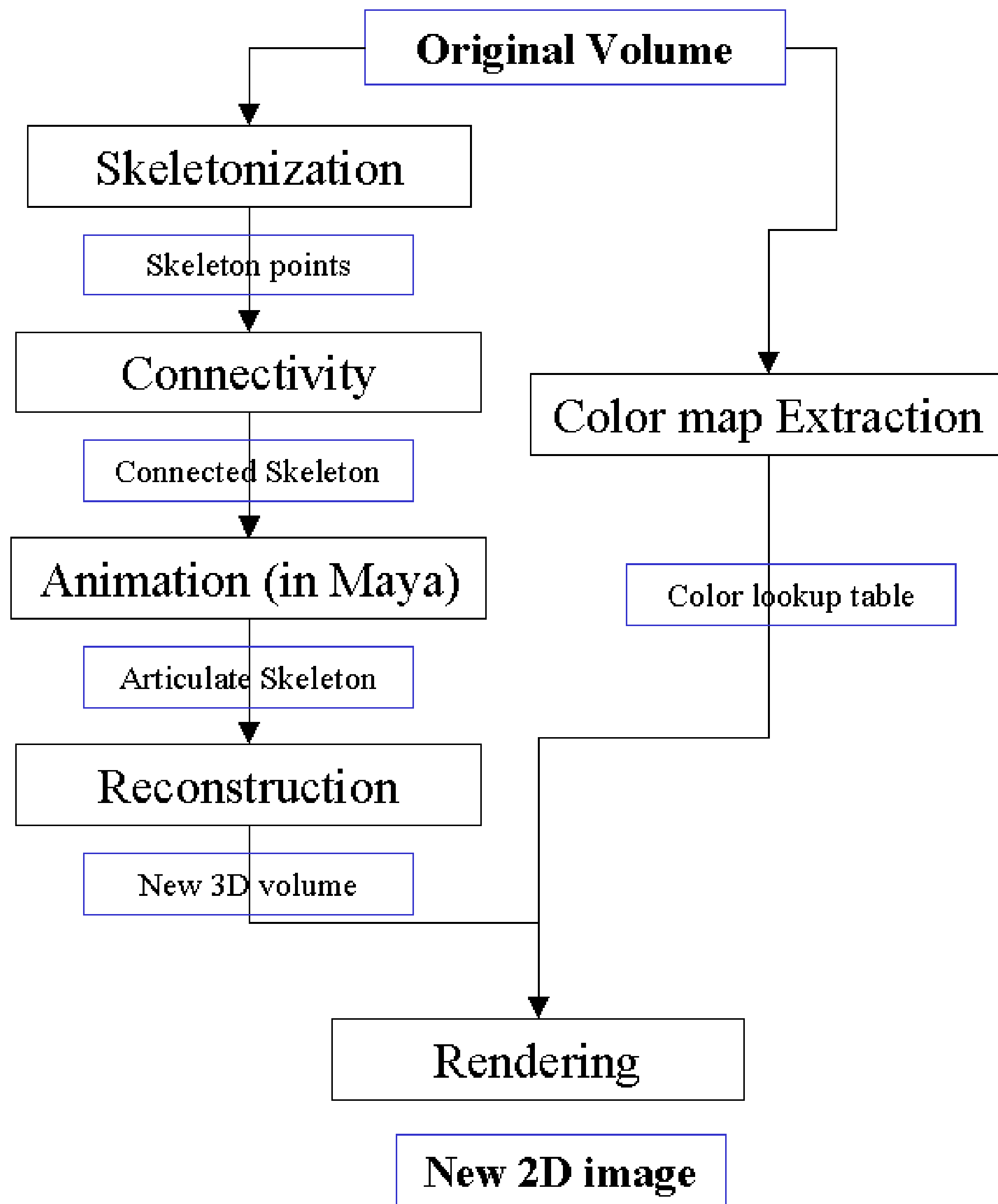

Figure 2.1: The Volume Animation Pipeline

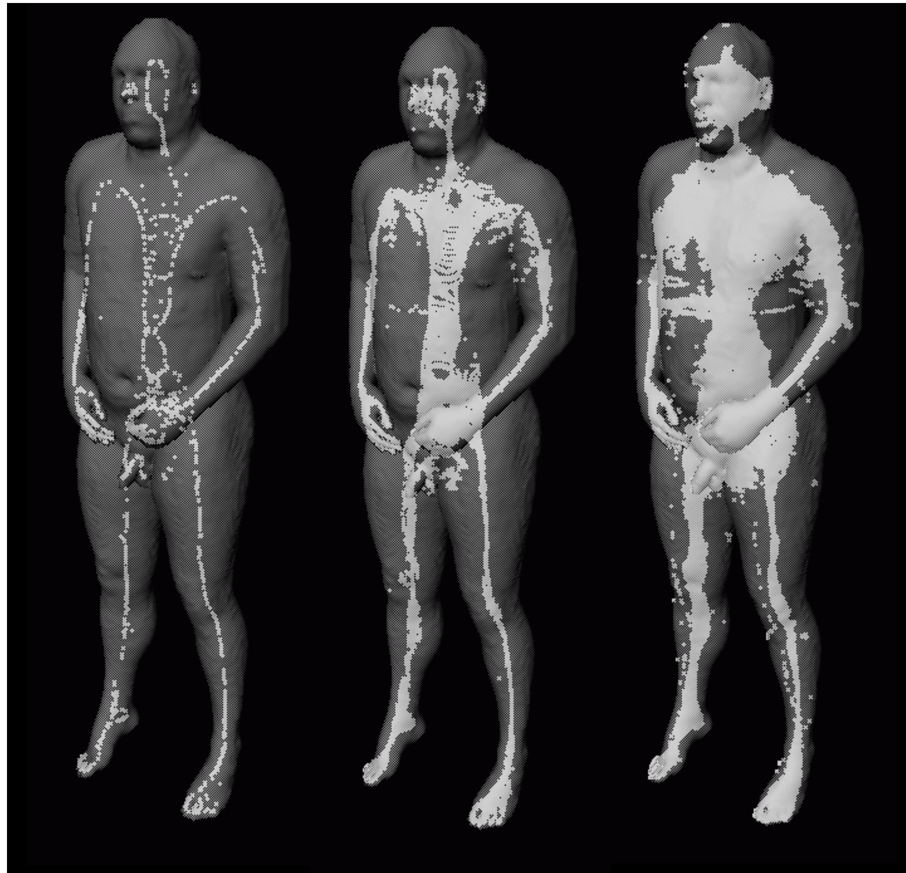

Figure 2.2: The Skeleton Points, Shown Inside the Surface Hull of the Visible Human Volume, in 3 Different Thinness Values- 1.8, 1.5, and 1.3

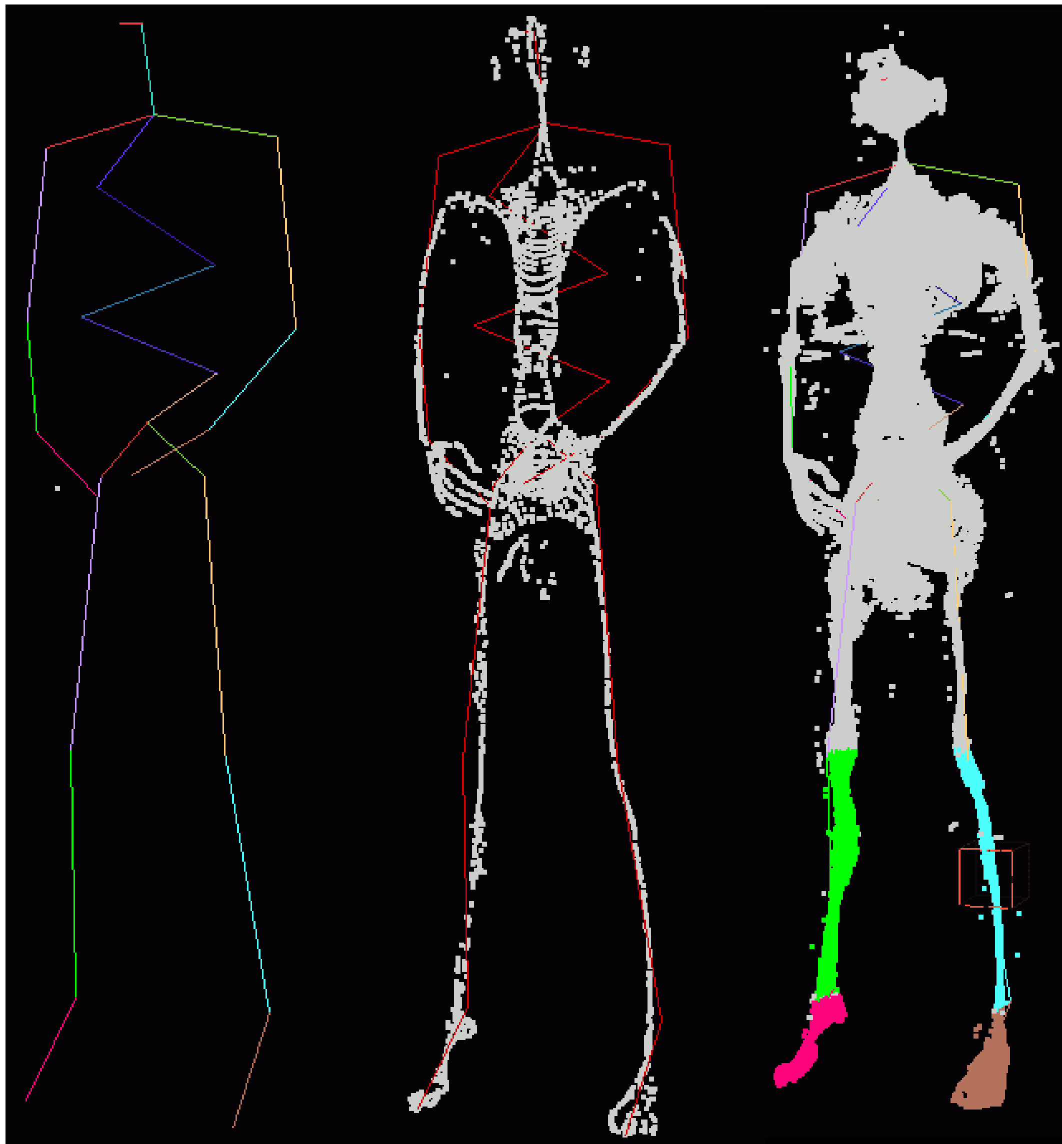

Figure 2.3: Root Nodes Selected In The Proper Order, Skelselect Started Up With The Skeleton Points And The Bone Skeleton, Partially Connected Skeleton

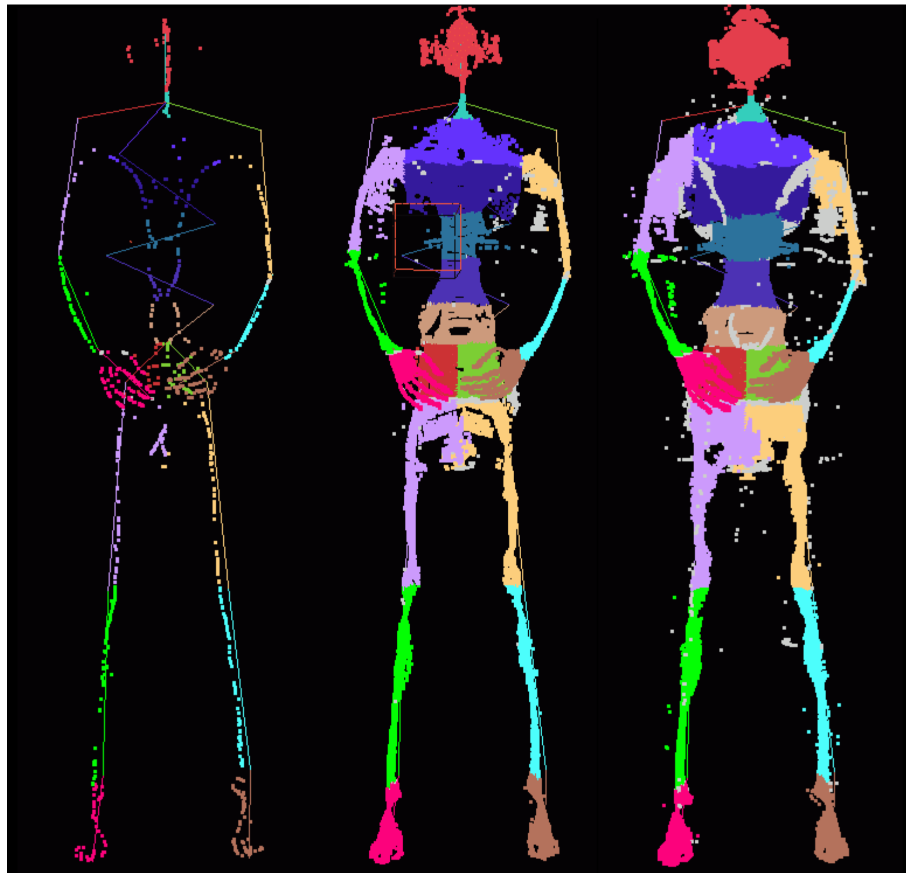

Figure 2.4: Final Connectivity Information Shown At 3 Different Thinness Values Using Skelselect

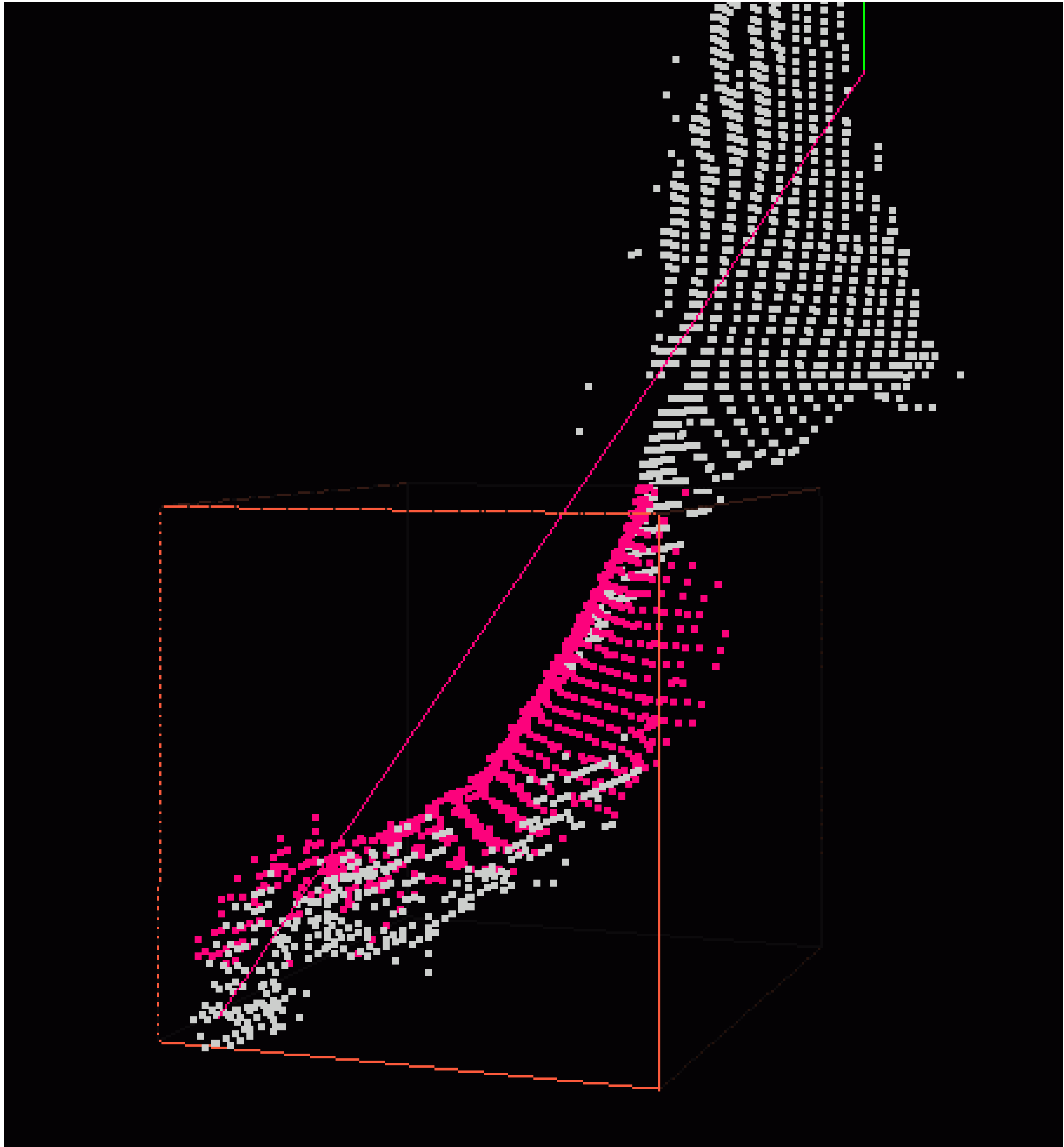

Figure 2.5: Partial Connection of a Foot Node in Skelselect

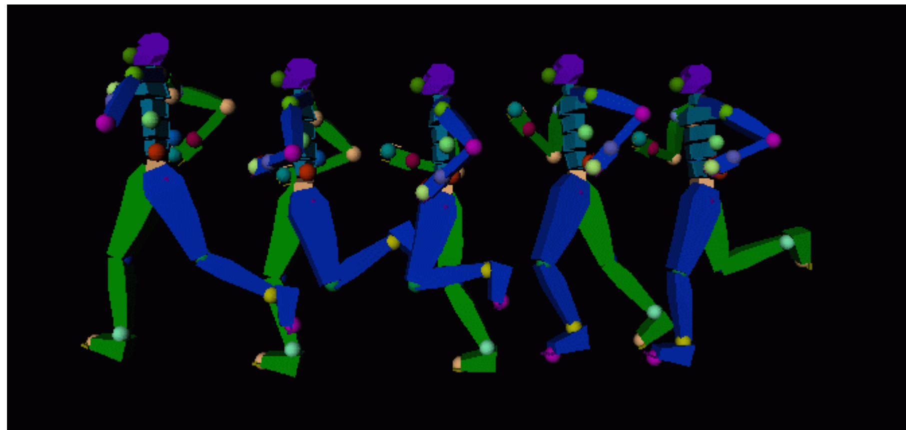

Figure 2.6: Inventor Files Converted From The Motion Capture Dxf Files

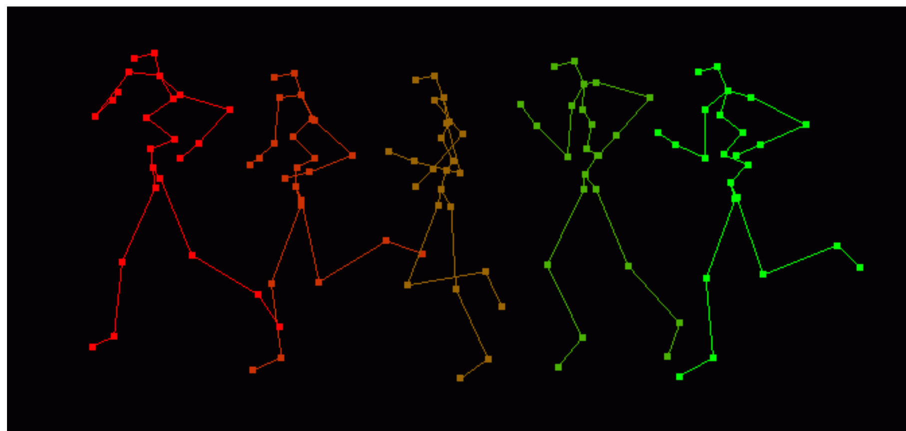

Figure 2.7: Sequence After Extracting The Articulated Skeleton Points

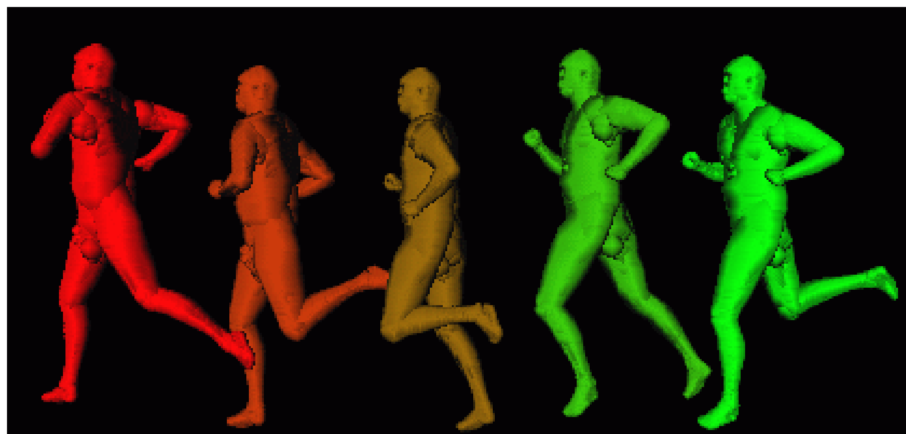

Figure 2.8: Sequence After Connecting Skeleton Points And Reconstructing Spheres

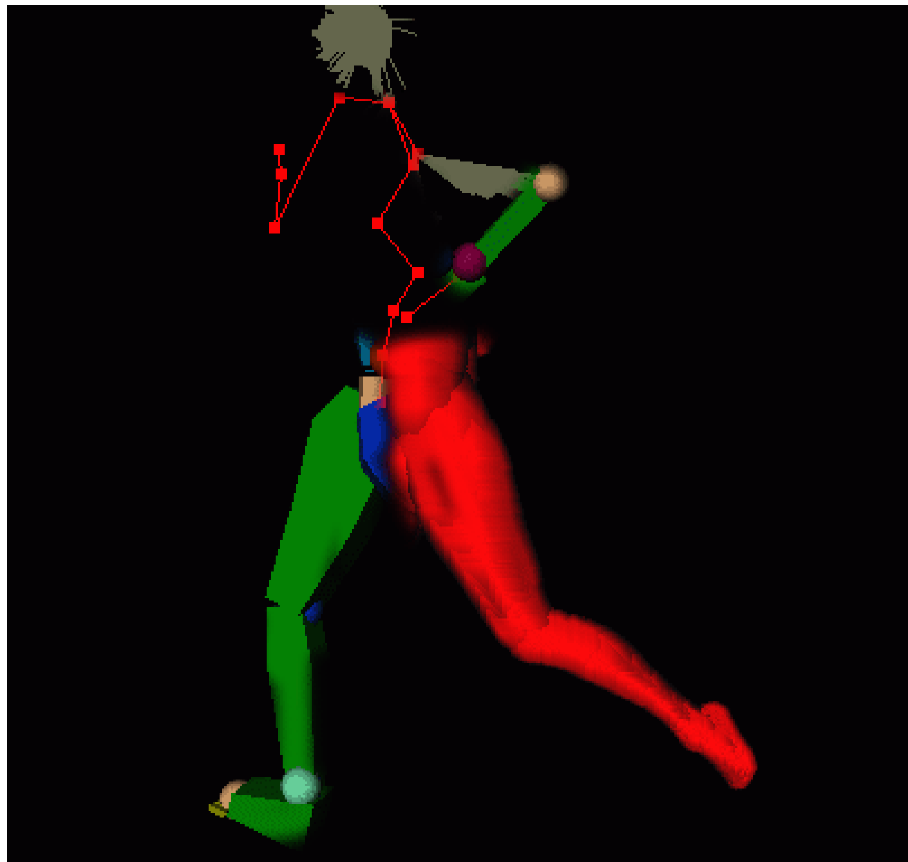

Figure 2.9: Splatter of The Various Forms of Post-Animation Representation

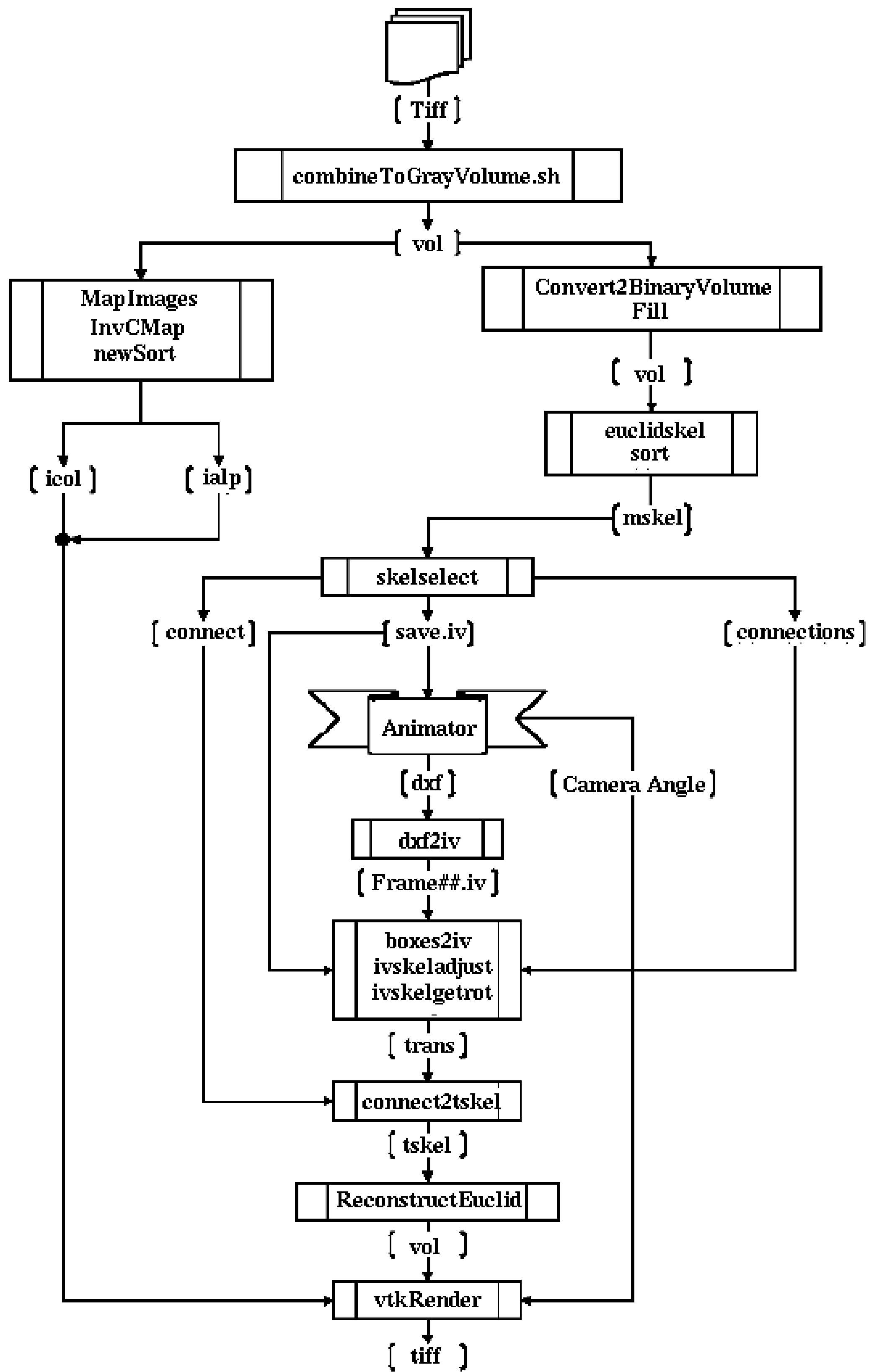

Figure 2.10: The Volume Animation Pipeline

Chapter 3

Enhancements to the Pipeline

3.1 Faster Reconstruction

A brief overview of the volume animation pipeline has been discussed in the previous chapter. Here we shall concentrate on one particular phase of the pipeline, the reconstruction process. This is the most time-consuming process in the pipeline. Hence, it's important to optimize this program as much as possible, in order to have a faster turn-around time for volume animations.

3.1.1 Current Approach

The process of reconstruction is computationally intensive. This is primarily due to the intense floating point computations- for Euclidean distance transforms, and martix multiplication and inverse operations. Details of distance transform computations in the 3-4-5 metric and the Euclidean metric are discussed in [22].

The reconstruction code scanfills a sphere about each skeleton point. In case of binary, or pseudo-binary, modes of reconstruction, the scan-fill is simple - all points within the sphere boundary have to be filled in with the constant data value. In the case of sampled reconstruction, this is somewhat more intensive - each voxel that lies within the boundary of the sphere has to be filled in with the data value corresponding to that point in the original volume. This involves matrix operations that are very expensive in terms of computation time.

The original method of sampled value reconstruction [15, 19, 22, 23] had the following steps:

1. The original, undeformed volume is loaded into memory to act as the lookup volume.

2. Memory is allocated for buffer for the final volume. The size of the final volume is passed as a parameter to the program.

3. The *.tskel file containing the skeleton information for the frame in hand is scanned for a root node (bone). This is marked by a line of format "Root: [x1] [y1] [z1] :: [x2] [y2] [z2]" in the text of the tskel file.

4. Once a root has been found, the transformation matrix associated to this bone is read in. This information is contained in the 4 lines immediately following the "Root" line in the tskel file. The matrix is stored in the file as a 4x4 transformation, in floating point precision. The inverse of this matrix is computed, and both the matrix and its inverse are stored in memory.

5. The lines that follow in the tskel file contain the centers and the radii of all the skeleton points that are associated to the current root segment, and thus are subject to the same transformations. These lines are read in one at a time, each containing data in the form of the sphere center (x, y, z), and the DT of the sphere.

6. For each sphere, a cube of size corresponding to the DT of the current sphere is scanned. Each point of the cube is checked for inclusion in the sphere - the voxel is inside the sphere if its distance from the sphere center, measured in the 3-4-5 metric, is less than the DT of the sphere.

7. For voxels that are found to lie inside the sphere, we use a 'stencil buffer' to determine if the value at this voxel is going to be filled in by this sphere, or by some other sphere that also includes this voxel. A stencil buffer is an array of the same size as the destination volume, and stores, at each voxel, the distance to the center of the last sphere that filled this voxel. All values are set to an arbitrarily large number to start with. The rule of relaxation states that a sphere will fill in a voxel only if the distance mentioned above is less than the value of the stencil buffer at that voxel.

8. If the voxel satisfies the relaxation rule above, the coordinates are multiplied by the inverse transform matrix to obtain the corresponding point in the lookup (original) volume.

9. The data in the lookup volume voxel thus obtained is filled in to the point.

10. The steps 7-9 are repeated for all voxels within the bounds of the cube in step 6.

11. Steps 5-10 are repeated for all spheres included under the root node in step 4.

12. Steps 4-11 are repeated for all root nodes in the tskel file.

3.1.2 Disadvantages of the Approach

The following are the most obvious disadvantages of the previous approach:

1. The scan-fill of the sphere has to start from the enclosing cube. Every point in the cube is tested for inclusion into the sphere. Since the spheres themselves overlap each other, and the cubes are larger than the spheres, this results in a huge number of redundant computations.

2. In order to get the mapping correct, each voxel is multiplied by the inverse of the transform the skeleton point underwent. This is the transformation associated with the sphere the voxel belongs to. This is an operation involving 4x4 matrices, and even when using fast matrix multiplication libraries, reconstruction easily becomes the bottleneck factor in the animation pipeline.

3. The relaxation rule says a voxel is to be considered included in a sphere if the distance to the center of this sphere is less than the distance to the center of the last sphere that filled this voxel, or the sphere is the first one to touch this voxel. This involves computation of distances for every voxel that is within the bounds of the sphere under consideration. When the algorithm is ported to the Euclidean metric, this involves floating point square root computation, one of the most expensive computations in terms of time required.

4. Similarly, in the Euclidean metric, the computation of distance from a voxel to the center of the sphere, to check for inclusion in the sphere, becomes a slow computation.

5. In the 3-4-5 metric, for small volumes (with maximum DT less than 255), the stencil buffer can be of type unsigned characters, 1 Bytes per voxel. For volumes having

maximum DT greater than 255, we require each stencil voxel to be at least an integer. In the Euclidean metric, the stencil buffer has to be of floating point precision. This means a 4- fold or higher increase in the size of the buffer.

The idea is to optimize the algorithm so that the number of floating point operations, particularly square root computations and matrix operations, are minimized, while maintaining the general idea of filling the volume from the lookup and conforming to the relaxation policy. Several optimizations were carried out, as discussed below.

3.1.3 New Reconstruction Algorithm

The new reconstruction algorithm is as follows:

1. The original, undeformed volume is loaded into memory to act as the lookup volume.
2. Memory is allocated for the final volume and the stencil for storing the distance of each voxel to the center of the nearest sphere that it belongs to. The buffers are initially filled in with zeros.
3. Each line of the tskel file is read in. If the line is detected to be a "Root" information, the matrix following the line is read in, the inverse of the same computed, and both matrices stored in memory.
4. If the line read in is a skeleton point, the center of the sphere and its distance transform are read in. The sphere center is multiplied with the transformation matrix to obtain the new center in the transformed volume.
5. The new center is passed on to a Bresenham circle-drawing algorithm that generates the circle made by the intersection of the sphere with the Y-Z plane. This gives us a series of z-values and the radius of the circle in the XY plane corresponding to each z-value.
6. Each (z-value, radius) pair generated in step 5 is fed into another Bresenham circle drawing algorithm that generates the end points of the circle in the XY plane.

7. The end points generated above are multiplied by the inverse transformation matrix computed in step 3, to obtain the corresponding points in the original undeformed volume.

8. We now have a line in the transformed volume, aligned in X, and a corresponding line in the lookup volume. The former is traversed along the x- direction, one step at a time, and the data in the corresponding point in the lookup line is copied to this voxel, so long as the relaxation rule is satisfied. The relaxation rule is checked by computing the distance from each voxel to the center of the sphere being currently reconstructed, and comparing this distance to the distance stored in the stencil buffer for the same voxel. If the new distance is smaller than the previously stored value, the data at this voxel is updated from the lookup volume, and the stencil updated to the new nearest distance computed.

9. Steps 7-8 are computed for each pair of points generated in step 6.

10. Steps 6-9 are computed for each pair of points generated in step 5.

11. Steps 5-10 are computed for each skeleton point read in step 4.

3.1.4 Scan-Filling a Sphere Using Double Bresenham's Algorithm

A major time-consuming operation in the original reconstruction algorithm was detection of points that are inside a given sphere. The easiest way to do this would be to scan the axis-aligned cube that contains this sphere, and compare the distance of each voxel in this cube to the center of the sphere with the radius of the sphere. This is extremely redundant and time consuming, since all we really need is the two end points that each scan line intersects the sphere at. In other words, we need a list of points that make up the boundary of the sphere.

To this effect, we use two instances of Bresenham's circle drawing algorithm. The basic algorithm is as follows:

```
while (newY<= newZ)
{
```

```
        ScanXYCircle(newY, newZ);
        ScanXYCircle(newY, -newZ);
        ScanXYCircle(newZ, newY);
        ScanXYCircle(newZ, -newY);

        if (diff < 0)
        {
            diff += 4*newY + 6;
        }
        else
        {
            diff += 4*(newY-newZ) + 10;
            newZ--;
        }
        newY++;
    }
```

In the code example above, each ScanFillX (X, Y, Z) scan-fills a line from (-X, Y, Z) to (X, Y, Z).

The first call to the routine draws a circle in the Y-Z plane. This is the circle that the sphere would make on the YZ plane, had it been centered at the origin. In other words, we obtain, for each value of Z in the sphere, the projection of the sphere at that z-value on the X axis. This gives us the radius of the circle in the XY plane for each z-value within the sphere.

The second call to the routine takes this radius and gives us the boundary points of the circle in the XY plane. Since the second routine is called for every z-value in the first routine, the boundary points of the entire sphere are thus obtained.

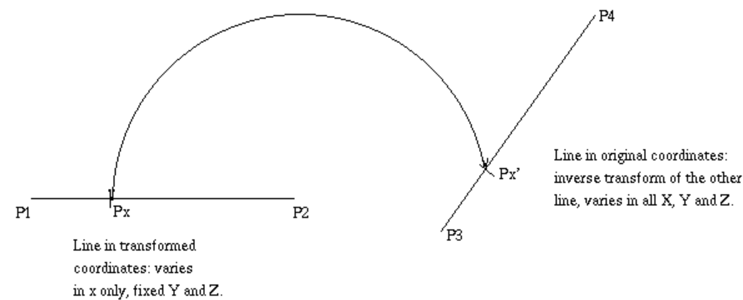

Figure 3.1: Scan-filling One Line from Another, Using Parametric Equations

3.1.5 Parametric Equations to Reduce Matrix Multiplication

The use of parametric equations to minimize the number of matrix multiplications while scan filling a line is illustrated in Figure 3.1. As given in this figure, we scan across the line P1-P2. For each point on P1-P2, we compute the corresponding point on the line P3-P4, based on the ratio of the point on P1-P2 to the total length of the line. Thus, P1 corresponds to P3, P2 does to P4, and every point in between on P1-P2 finds it's match on the other line.

3.1.6 Faster Relaxation Rule

The rule of relaxation we had laid down earlier says that a voxel is filled during the scan fill of a particular sphere only if the distance from the center of the sphere to this voxel is less than the distance from the center of the last sphere that filled the same voxel. This must be done to account for overlapping spheres incorrectly overwriting values after animation. As mentioned earlier, at the start of the program, the entire stencil is set to an arbitrarily large value.

This essentially means computation of the distance from the voxel to the center of the sphere, carried out order-N times over the volume. This is a substantial task in Euclidean coordinates, as each distance computation involves a square-root operation. Moreover, the stencil has to be in floating point precision - considering the super huge volume sizes, even when storing the volume as an array of unsigned integers, having a buffer of this size with floating point precision requires a tremendously large amount of memory.

A better method- both in terms of memory requirement and in terms of speed - is to

maintain the stencil buffer in the 3-4-5 metric. This will allow us to maintain the stencil as unsigned integers. Moreover, the 3-4-5 metric does away with the expensive computation of the square root. Leaving the distances squared leads to expensive memory allocation in order to store the large (squared) distances.

3.1.7 Code Level Optimizations

Our basis for optimizations was to **make the common case faster**. In the reconstruction process, this refers to the line filling algorithm, since this is called for every pair of points generated for every circle drawn for every z-value in the sphere that is built around each of the skeleton points undergoing reconstruction. If the number of skeleton points is of O(N), the number of times the line scan-fill routine is called is of approximate $O(N^3)$.

Some common methods of code level optimization are the following:

• Common Subexpression Elimination • Strength reduction • Copy Propagation and Dead Store Elimination • Global variables vs. parameter passing • Inlining Functions • Use of Compiler Optimization Flags

3.1.8 Timing and Profiling: Results of the Optimizations

Several optimization solutions have been mentioned above. The general approach to optimization is to make the common case faster. For this, we need to know which code section of the program is taking the largest share of the execution time.

A profiling of our code, using the SGI utility 'pixie', gave us this information, as can be seen from the last column of Table 3.1. As can be seen from the table, the old method of reconstruction had the following 'hotspots', or time-taking routines:

- the scan-fill routine for the spheres,
- the distance computations for the stencil buffer, and
- the Inventor routines for matrix multiplication.

The optimization solutions discussed in the preceeding sections were applied to the old code to remove these hotspots. The results of the various methods of optimization are

shown in Table 3.1.

Table 3.1: Code Profiling: Comparison of Various Algorithms and Optimizations

Description	# of instructions	Execution time (sec)	Average cycles per instruction	Function execution times (% of total time)
dv_reconsetskel	20,060,490,172	101.375	0.985	`growSphere` 75.9% `sqrt` 10.7% `multVecMatrix` 6.7% `rint` 6.6%
ReconstructEuclid (Scan fill)	12,497,858,353	58.497	0.913	`GrowYLine` 20.9% `GetSquareDistance` 20.2% `floor` 19.2% `IndexOf` 18.8% `SampledLookup` 8.5%
ReconstructEuclid (stencil modified to 345 metric)	10,421,646,375	53.710	1.005	-
ReconstructEuclid (some functions made inline)	7,926,050,618	33.257	0.818	`GrowYLine` 50.1% `Get345Distance` 22.1% `SampledLookup` 19.4% `multVecMatrix` 2.4%
ReconstructEuclid (binary recon only-with above version of the sampled recon)	2,515,130,670	12.534	0.972	`GrowYLine` 84.2% `powf` 4.6% `exp` 2.9% `log` 2.8%
ReconstructEuclid (powf changed to sqrt, O3-mips4optimized)	7,625,832,381	31.950	0.817	`GrowYLine` 52.1% `Get345Distance` 23.0% `SampledLookup` 20.2% `multVecMatrix` 2.5%
ReconstructEuclid with Bresenham's scanfill, binary mode.	1,499,866,709	10.943	1.423	`ScanFillX` 97.6%
With Bresenham, after inlining and speeding up the 345 distance computation, binary data.	1431634714	10.719	1.460	`ScanFillX` 97.3%

Chapter 4
Additional Functionality

The various programs that make up the volume animation pipeline are shown in Figure 4.1.

4.1 Creating the Colormap

The original visible human volume is in 24-bit color. The size of the dataset is about 1.1 GB (GigaBytes). Note that this size is just for the man in the standing position - when motion capture is applied to a volume, its final size can be as much as 10 times or more of the original pose. Since the number of unique colors in the original volume was not large (the colors were mostly shades of red / tan), we mapped the colors to 255 unique best-fit colors in order to save on volume size (by saving as unsigned characters).

The steps needed include:

1. Determining 255 unique colors in the volume.
2. Map colors to these 255.
3. Create a new colormap.

We took slices from various regions of the volume to create a collage, shown in Figure 4.2, that had most of the colors in the volume. This image was then reduced from 24-bit color to 8-bit color, i.e. 255 colors. These 255 colors would form the colormap of the final volume. The extracted colormap is shown in Figure 4.3. We used the freely available program, **InvCMap**, to map these colors to the volume. **InvCMap** creates a Voronoi diagram of the desired colormap on a 24-bit colorspace. The output of **InvCMap** is a volume, *colormap.vol*, that maps the 24-bit colorspace to an unsigned char value, which maps to the 8-bit colormap in *butchershop.gif*. Any voxel, at coordinates [R, G, B] in *colormap.vol*, has

an unsigned char data value that maps the color [R, G, B] to its nearest match in the 8-bit colormap provided by *butchershop.gif.*

We now run **mapImages** on each slice of the 24-bit color volume (stored as a .ppm file), and obtain a new set of .ppm files that are in 8-bit color of the desired colormap. These .ppm files are re-combined to give us an 8-bit color volume.

4.2 Sorting the Colormap

The 8-bit color volume we produced in the step above works fine in theory - we have a volume, where each voxel has a data value that points to a colormap, and each slice looks fine in 255 colors. However, when viewed as a volume, using volume visualization packages like **Bob** or **Volumizer**, we face a problem - while rendering, each voxel is anti-aliased with the colors of its neighboring voxels. In our case, though the neighboring voxels have colors that are visually similar, the lookup indices into the colormap are totally different, since we created the colormap without sorting the colors in any way. As an example, suppose indices 100 and 200 are both very similar shades of dark red. When anti-aliasing takes place, the voxels will have a value somewhat between the two, say 135. Now, index 135 in the colormap can very well be green, or blue, or any other shade. This results in a very speckled volume, each color seeping into the next in a randomized colormap, and is not recognizable as a human shape. What is needed is a "sorted" colormap with smooth transitions in color range.

The problem is thus one of sorting a series of values based on their [X, Y, Z] coordinates, the analogy to the [R, G, B] values that we actually have, and create the smoothest possible gradient. Some possible sorting methods may perform:

- Sorting first by B, then G, then R values, using a sort like radix sorting.
- Sorting by intensity, where intensity is the sum of (R + G + B) for a color cell.
- Cluster sorting - insert a few seeds into the colormap at adjustable intervals, assign each color cell to the seed nearest to it, and then sort all cells assigned to a particular seed by their absolute distance from the seed.

However, the results from these sorting methods did not satisfy the requirement - the

sorted colormap was not as smooth as is required to get a good rendering, specially when the transparency value (alpha) is kept low. A low transparency implies that a greater number of voxels contribute to the color of one single pixel in the final redered image, and hence it is all the more required to render with a smooth colormap to avoid color blending artifacts.

One method of sorting tried out was sorting by gradient. Like other sorting methods, we compute the distance of all the free voxels from the last voxel that has already been added to the sorted result. In addition to this, we add a penalty factor to this distance. The penalty occurs when any of the gradients - in R, G or B - undergo a change. For instance, suppose the last 2 points that were added create a gradient of increasing R, decreasing G, and increasing B, and that the penalty parameter has been set to ‘P’. Suppose the closest point by distance changes the gradient to decreasing R and decreasing B, while continuing to increase the G value, and that it's distance from the last added voxel is ‘D’. The effective distance of this voxel is then computed as ‘D + 2P’. Another voxel may be at a distance greater than D, but following the gradient - due to zero penalty, this point will have a lesser effective distance. The voxel that has the least effective distance from the last added voxel will be inserted into the sorted colormap. By adjusting the penalty factor, we can get a smooth transition from darker to lighter colors. Once the end of a gradient is reached, i.e. there are no other points that are in the same gradient direction, the voxel with the least effective distance will reverse the gradient in at least one axis, and thereby create a new gradient to follow.

This sorting yielded results that were much better than the previous methods discussed, and did produce a smooth colormap. However, there are sharp breaks where the gradient suddenly reverses, as can be seen from Figure 4.4.

After adjusting the parameters of the weighted gradient method, and combining this with the other methods discussed earlier, it was decided that the position of each color voxel has to have an influence on its neighboring cells. The influence of the surrounding voxels will result in a smoother overall transition. However, local variations must be permitted, otherwise a very smooth curve will end in an abrupt change of direction to another smooth curve in the opposite gradient direction. It was because the penalty method was preventing these local variations that it was ending up with a number of small gradients after it was

done with the primary gradient, and these small gradients lead to an unacceptable tailing end to an otherwise smooth colormap.

To this effect, we resorted to the following hybrid method of sorting, the results of which can be seen in Figure 4.5.

```
void SuperGradientSort() {
  newR = 2*theArray[1][R] - theArray[0][R];
  newG = 2*theArray[1][G] - theArray[0][G];
  newB = 2*theArray[1][B] - theArray[0][B];

  for (i=5; i<MAXCOLORS-1; i++) {
      minDist = 9999;
      minIndex = i;
      searchR = searchG = searchB = 0;
      baseR = theArray[i-1][R];
      baseG = theArray[i-1][G];
      baseB = theArray[i-1][B];
      rangeR = newR - baseR;
      rangeG = newG - baseG;
      rangeB = newB - baseB;
      deltaR = rangeR/20;
      deltaG = rangeG/20;
      deltaB = rangeB/20;
      for (counter=0; counter<=20; counter++) {
          searchR = baseR + deltaR*counter;
          searchG = baseG + deltaG*counter;
          searchB = baseB + deltaB*counter;
          closestIndex = GetClosestFuturePoint(searchR, searchG, searchB, i);
          tempDist = deltaR*counter + abs(theArray[closestIndex][R] - searchR);
          tempDist += deltaG*counter + abs(theArray[closestIndex][G] - searchG);
          tempDist += deltaB*counter + abs(theArray[closestIndex][B] - searchB);
```

```
            if (tempDist < minDist) {
                minDist = tempDist;
                minIndex = closestIndex;
            }
        }
        SwapRows (i, minIndex);
        newR = 2*theArray[i][R] - 0.5 *theArray[i-1][R] - 0.5 * theArray[i-2][R];
        newG = 2*theArray[i][G] - 0.5 *theArray[i-1][G] - 0.5 * theArray[i-2][G];
        newB = 2*theArray[i][B] - 0.5 *theArray[i-1][B] - 0.5 * theArray[i-2][B];
      }
}
```

The best results were obtained when the colormap was first sorted in increasing order of intensity (R + G + B) and then subjected to the sorting method discussed above.

4.3 Segmentation

We start the volume animation process from a sampled dataset, such as the visible human dataset. Each data-point (voxel) in this dataset has a value between 0 and 255 (inclusive). This 256-color index was formed by reducing the photo dataset from true (24-bit) color to 256 unique colors, as described in the previous section.

It is necessary to segment out the required volume from the background in order to animate the volume. This is performed by passing the volume through a band pass filter. The filter program takes in a lower and upper threshold. Any data value within the range is changed to 1, while all values falling outside the range are changed to 0.

It is important to verify that the segmented volume does not have 'holes' within it - a result of regions of the sampled volume falling below the threshold of segmentation, and melding with the background. If formation of holes is unavoidable, for instance, if the background has excessive noise of the same data value as these regions in the volume,

a region growing program can be used, or any computer vision operator can be used, to manually fill in the holes prior to the next step (skeletonization). The **Fill** program, discussed in the appendix, is one such program that does a convex hull filling of the volume.

4.4 Reconstruction Modes

The reconstruction program can be run in one of several modes. The most common mode is the one that has been referred to so far in the thesis, 'sampled reconstruction'. Here we shall discuss some of the other modes of the program, and their purposes.

4.4.1 Binary Reconstruction

Binary reconstruction produces a binary representation of the shape in the new pose. All voxels belonging to the shape are marked with a single non-zero value, typically either 1 or 255, and the rest of the voxels are set to zero. This is the fastest mode of reconstruction, and is used for verification of proper reconstruction, detection of breakage at joints, and a general idea of how the volume is going to look.

The reconstruct program processes one bone of the articulated skeleton at a time. The bone, the transformation it has to undergo, and the list of skeleton points attached to it are read in from the '*.tskel' file. Each skeleton point now has to be re-grown into the sphere it represents, the radius of the sphere being the DT associated withthe skeleton point, at the correct position. The skeleton points are multiplied by the transformation matrix to obtain the new center of the reconstruction sphere, and all the points in the sphere are filled with the pre-decided data value. Since all the points in the sphere have the same data value, rotation of the sphere about itself does not matter. This means that the filling of the sphere is straightforward, without the involvement of further matrix multiplication. This is why binary reconstruction is an order of magnitude faster than sampled reconstruction (discussed later).

4.4.2 HotSpot Reconstruction

For evaluation of the efficiency of reconstruction, we often need to know exactly how many times a voxel was visited, or filled in. If the volume is reconstructed with all transformations set to unity matrices, each voxel will be visited at most once in the entire reconstruction phase. However, when motion is applied to the shape, the skeleton points move about, dragging their sphere boundaries along with them. This changes distances between the spheres, and brings up regions of boundary interference, at which each voxel is visited by all the spheres within which it now lies. Multiple visits to a voxel indicate inefficiency in reconstruction.

The HotSpot mode of reconstruction gives us a clear picture of exactly how many times every voxel was visited. Instead of filling in the pre-decided data value in binary reconstruction, this maintains a 'fill count' at each voxel, and increments the count every time the voxel is visited. Important performance parameters that are obtained from this mode are the maximum and average number of times voxels were visited.

4.4.3 Mottled Reconstruction

This is yet another mode of pseudo-binary reconstruction, where each sphere is filled with a different data value. The reconstruction thus consists of multi colored spheres, and allow us to look into connectivity issues that come up when the shape undergoes transformations. For instance, it may be observed that the region that was expected to be a part of the upper arm is being reconstructed from a sphere of high distance transform centered in the chest cavity. In such a case, adequate changes are incorporated to the connectivity, by moving more points to the upper arm from the chest spheres, to compensate for this error.

4.4.4 Sampled Reconstruction

This is the most desired mode of reconstruction, and gives us the deformed volume with the color information retained from the original volume. This mode has been discussed in detail in chapter 3.

In chapter 3, we saw the enhancements carried out on our major issue, the reconstruction

phase of the pipeline. In this chapter, we have seen the enhancements carried out to some of the other stages of the pipeline. In chapter 5, we shall see the results of all the optimizations and enahncements discussed so far.

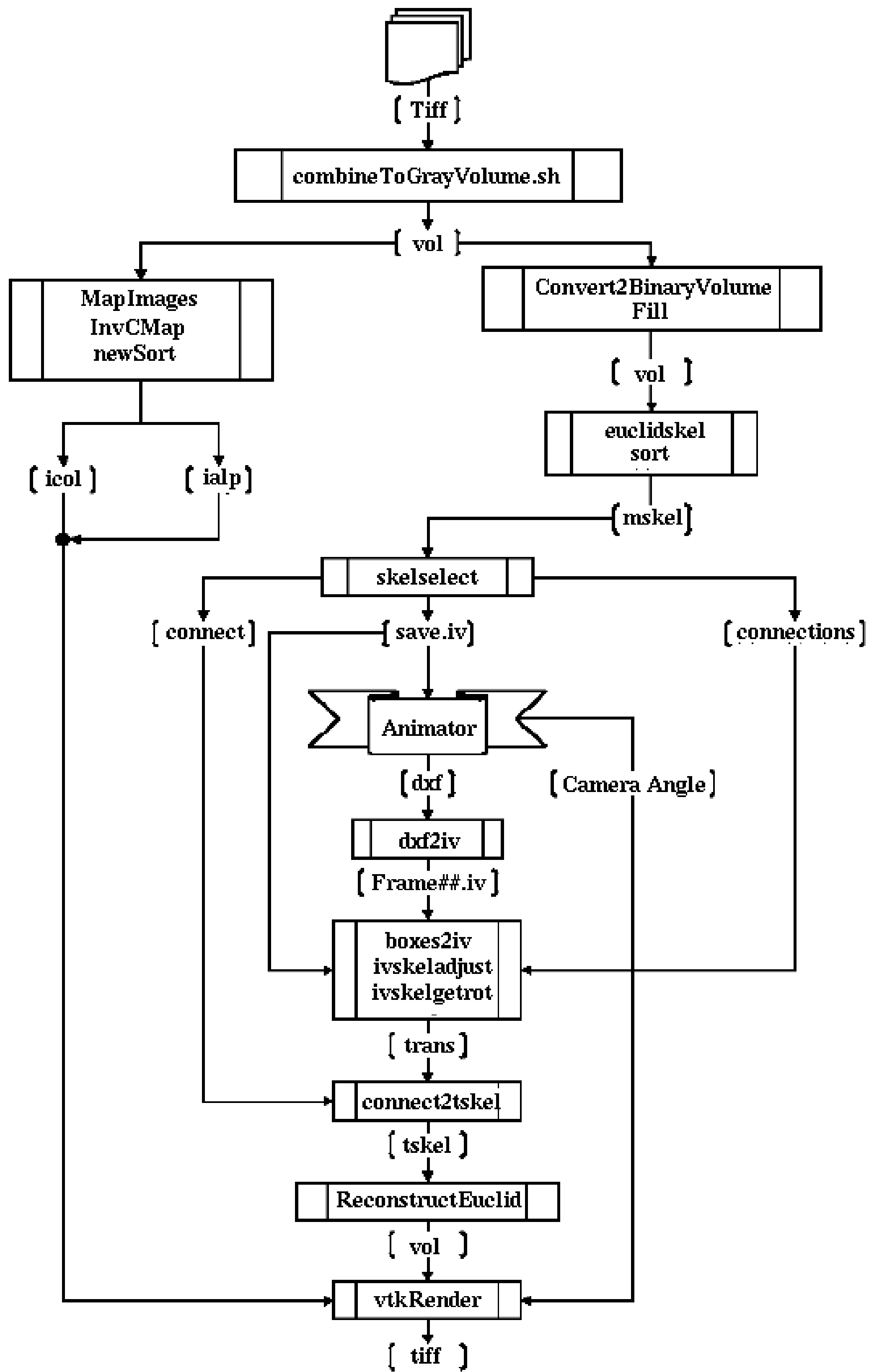

Figure 4.1: The Volume Animation Pipeline

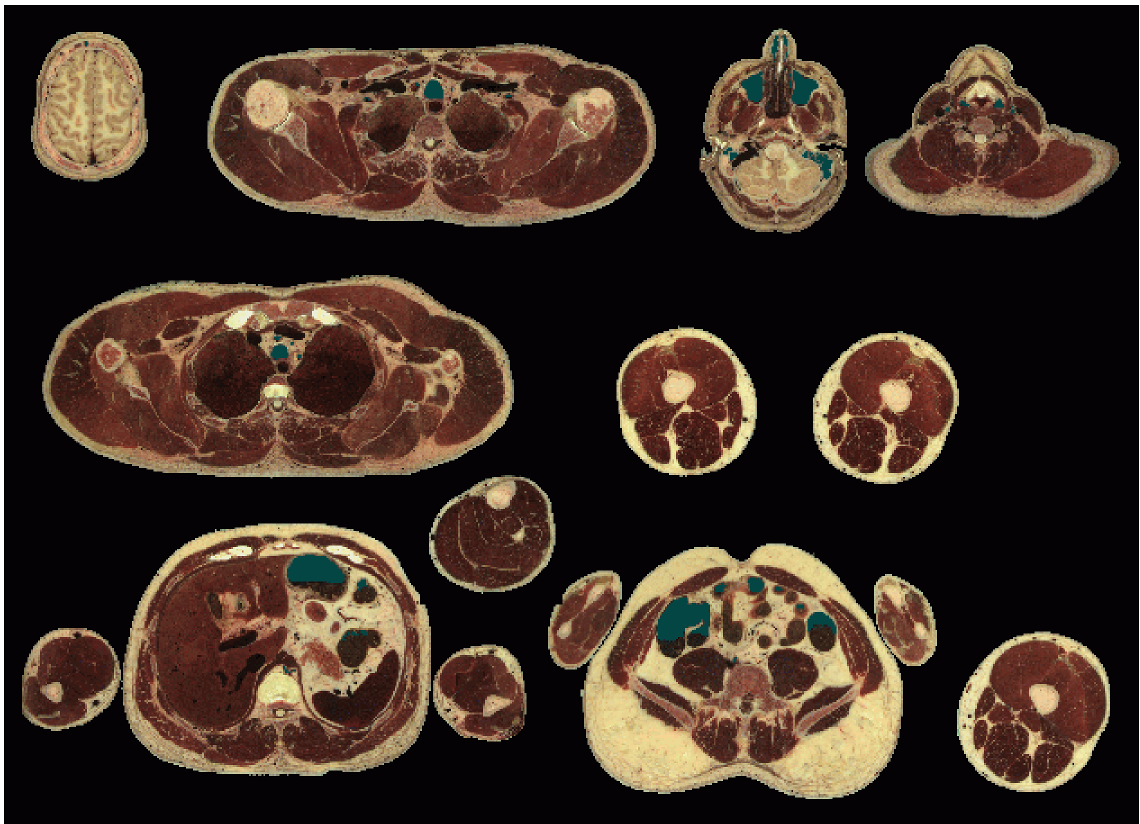

Figure 4.2: Butchershop.gif - Showing Range of Colors in the Dataset

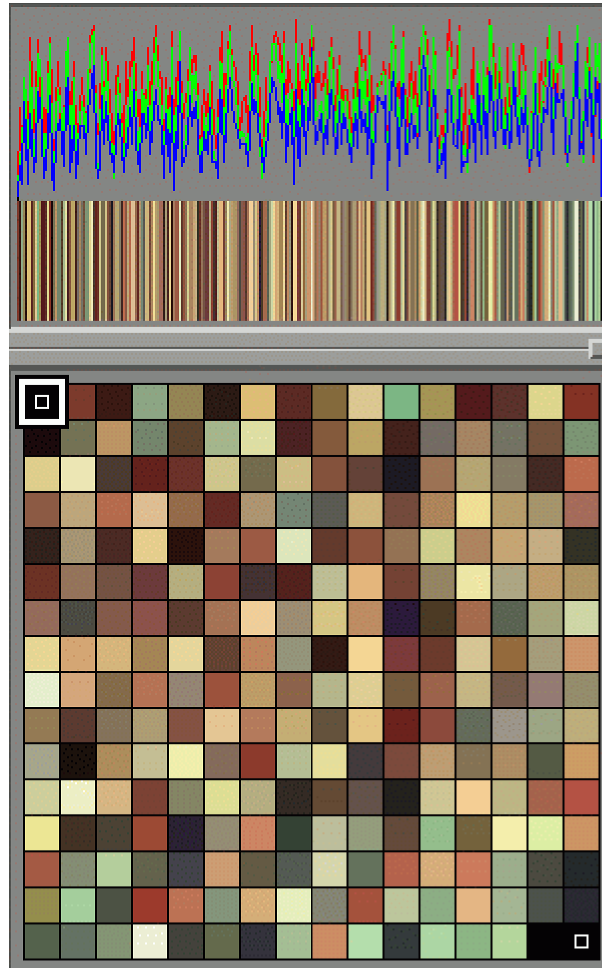

Figure 4.3: The Original Colormap Extracted from butchershop.gif

Figure 4.4: Colormap Sorted by Gradient with Penalty. Note the Discontinuities.

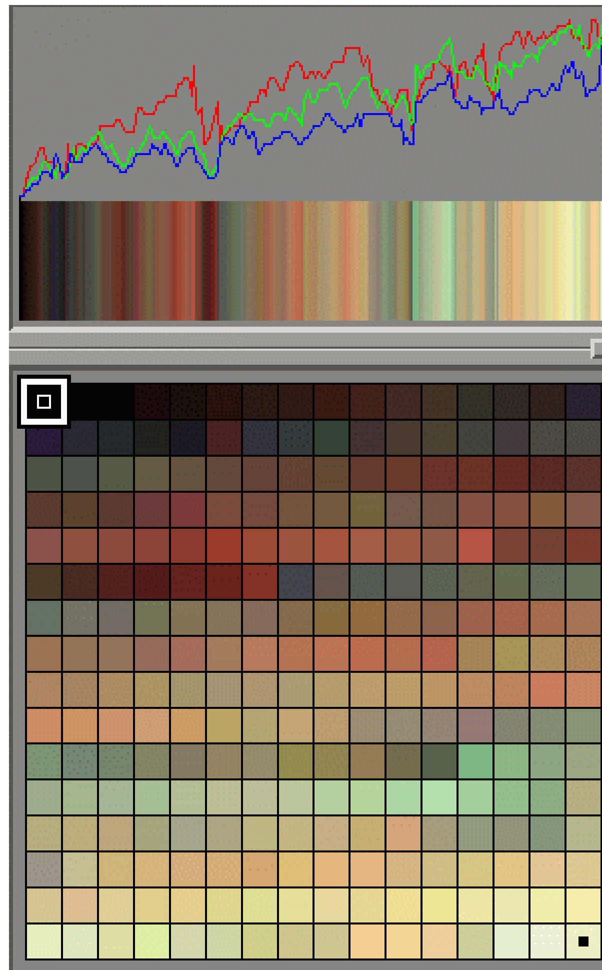

Figure 4.5: Final Results from newSort

Chapter 5

Results

Some of the improvements to the execution time presented by this thesis work have been presented in the preceeding chapters. Here we present a few more results that were targeted to bring out the key features of the enhanced volume animation pipeline.

5.1 Execution Time Improvements

The key focus of this thesis work was to improve upon the execution time of the reconstruction program. The results obtained are seen in the following figures.

Figure 5.1 shows the run time for reconstruction of the left leg of the visible human. This dataset has a resolution of 103x120x540 voxels, a size of 6.7MB, in the default pose. The skeleton was untransformed in position-1, and rotated by 45 degrees in position-2. The increased time in the second reconstruction is mainly due to the increased size of the resulting volume, resulting in increased disk write time.

Figure 5.2 shows the speedup for the sample cube dataset. This dataset is of resolution 128x128x128 voxels, a size of 2MB. The skeleton was rotated by 45 degrees for this reconstruction.

Figure 5.3 compares the run times for four different skeletons of the half-sized visible human. In the undeformed pose, this volume has a resolution of 290x169x940 voxels, a size of 46MB. The four frames were taken from 4 different animations. As we can see from the execution times, for a given dataset, the program takes roughly the same time to reconstruct, irrespective of the transformations the skeleton undergoes. This is because the time required for reconstruction depends on the number of skeleton points present in the original volume, and not on the transformations they undergo. The minor differences in the execution times are mainly due to the cost of writing larger file sizes to the disk. The file

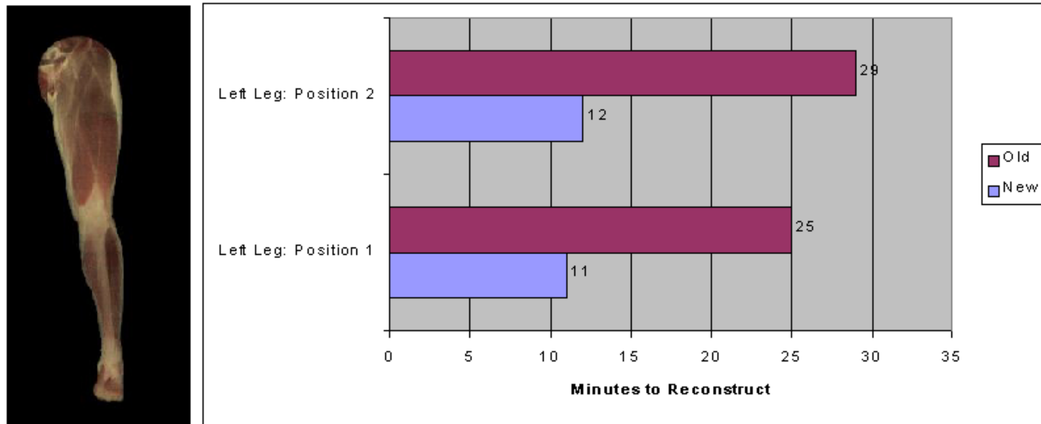

Figure 5.1: Reconstruction Times for the Left Leg of the Visible Human

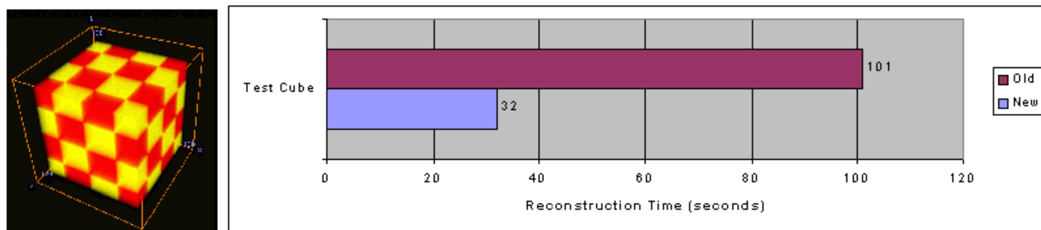

Figure 5.2: Reconstruction Times for the Sample Cube Dataset

size is determined by the bounding box of the transformed skeleton, and is thus dependent on the transformations of the frame at hand.

Figure 5.4 presents the average speedup seen by the new reconstruction program, as compared to the old program.

Figure 5.5 shows the cumulative speedup obtained by the different optimizations applied to the old program. The speedup after applying all the optimizations was 89.2% over the old program. As can be seen from this figure, the bulk of the speedup was obtained by modifying the scan-fill method, followed by the change to Bresenham's algorithm for the generation of the points on the hull of the reconstruction spheres.

5.2 Example 1: The Visible Human Dataset

The visible human dataset, in it's full resolution, has a size of 580x337x1879 voxels. Most of the animations we have produced deal with the half-size sampled dataset, having a resolution of 290x169x940 voxels, a size of about 46MB. The dataset was obtained as a series of TIFF

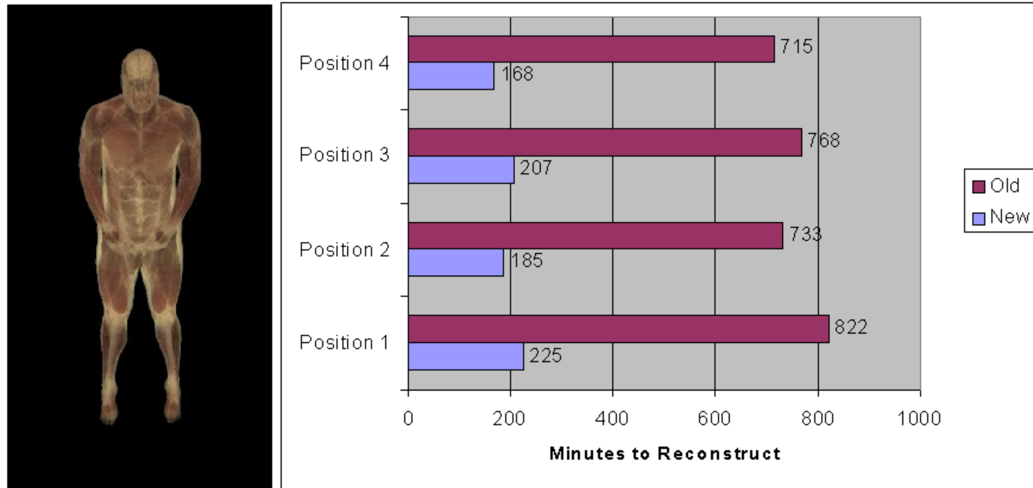

Figure 5.3: Reconstruction Times for the Half-Sized Visible Human

images, a result of the National Library of Medicine's Visible Human Project, [1]. The dataset was then reduced to 255 unique colors, and scaled to a size that was easy to use, while maintaining the expected proportions of a human being.

A few sample images from the animations that have been generated are shown in Figure 5.6, Figure 5.7, and Figure 5.8. These were generated without the use of any smoothing kernels. Figure 5.9 shows a rendering of a frame with the near clipping plane someway into the volume, so the interior details of the volume are made visible.

5.3 Example 2: The Colon Dataset

The original colon volume is shown in Figure 5.10. This dataset is of size 212x150x260 voxels. The objective was to display the colon in a virtually uncoiled state. This was achieved by skeletonizing the dataset, straightening out the skeleton points, and performing reconstruction of the new skeleton.

Figure 5.11 shows the isosurface of the segmented binary version of the colon dataset, in its undeformed state. Figure 5.12 shows the articulated skeleton and a very high thinness version of the skeleton points of the dataset. Figure 5.13 shows the final low-thinness (dense) connected skeleton of the colon that was used for the reconstruction process. Figure 5.14 shows the uncoiled version of the colon, reconstructed from the manipulated skeleton. Figure 5.15 shows the same volume, cut open from the center to show the data that is contained

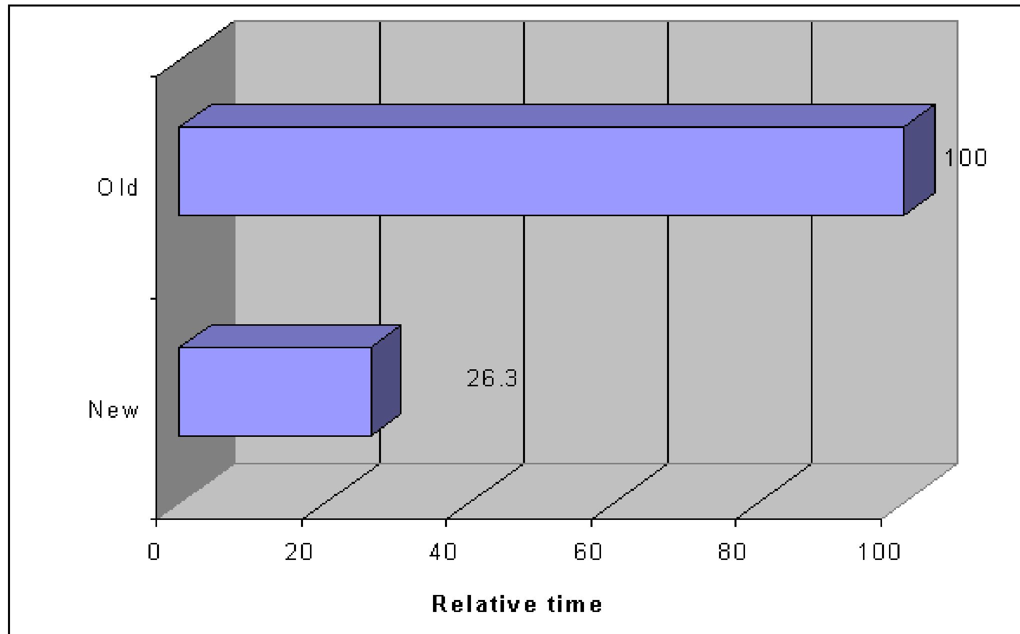

Figure 5.4: Average Speedup with the New Reconstruction Program

Table 5.1: Properties of the Colon Dataset

Property	Undeformed state	Deformed State
Size	205x133x261	991x90x94
Zero voxels	6458533	7810403
Non-zero voxels	657632	573457
Percent filled	9.24	6.84
Percent Voxel Loss	-	12.79

within. Table 5.1 gives us a few of the properties of the colon dataset.

5.4 Example 3: The Sample Cube Volume

The cube volume dataset was created to give a visual indication of the accuracy of the reconstruction program. The volume in its undeformed state is shown in Figure 5.16. This dataset has a size of 128x128x128 bytes. As seen from the figure, the volume is a 3-dimensional checkerboard pattern, filled with alternating voxel cubes of values 50 and 250, shown as darker and lighter squares. The background is filled with zero voxels. The properties of the volume are shown in Table 5.2. The eucledian skeleton of the volume is shown in Figure 5.17.

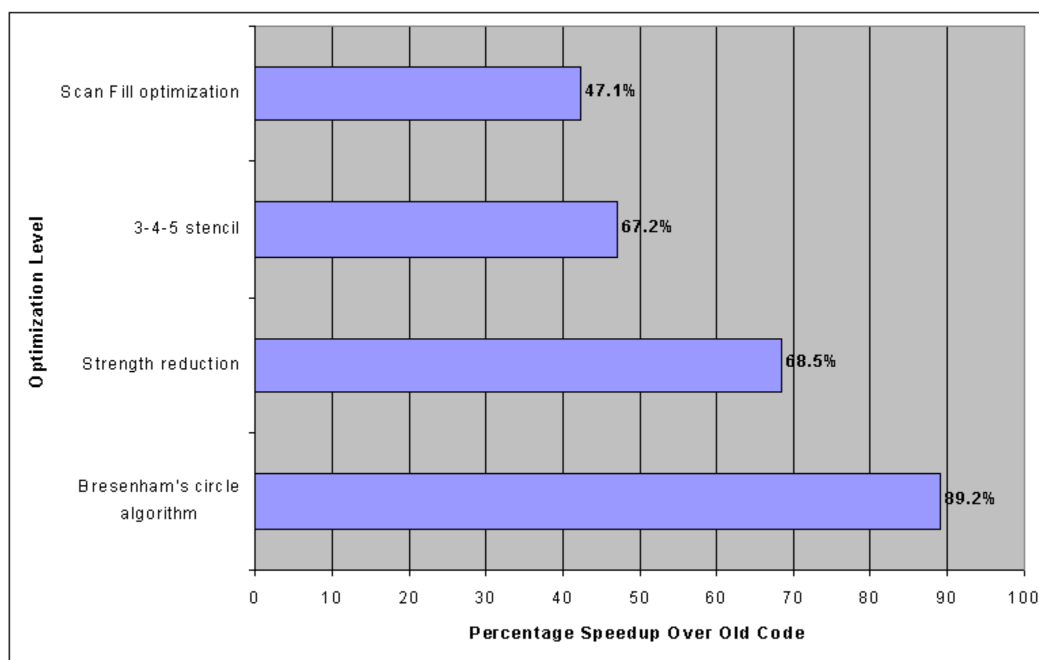

Figure 5.5: Cumulative Speedups Obtained by the Various Optimizations to the Reconstruction Program

Table 5.2: Properties of the Aliascube Dataset

Size:	128x128x128
Zero voxels:	872109
Non-zero voxels:	1225043
Percent filled:	58.41
Voxels of value 50:	612521
Voxels of value 250:	612522

5.4.1 Testing Aliasing Effects

We now rotate the skeleton by 45 degrees, and reconstruct into the volume shown in Figure 5.18. Figure 5.19 shows cross sections of the same, at various depths. Note that the squares have remained sharp, and the edges have not been deformed by the transformations. Such operations test the ability of the reconstruction program to handle aliasing effects and floating point precision loss errors.

5.4.2 Anti-Aliasing by a Gaussian Kernel

In vector graphics, a straight line stays as a line even when rotated about arbitrary axes. However, in rastor graphics, since the line exists as a series of distinct points, rotations

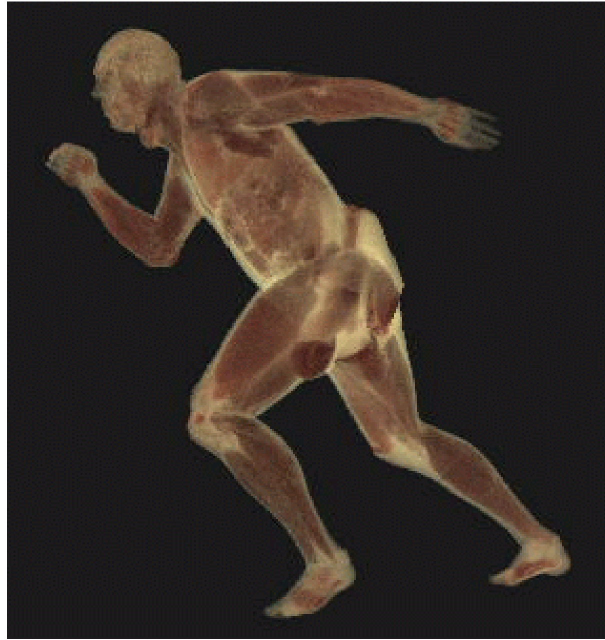

Figure 5.6: Visible Human Dataset, a Frame From the Jog Sequence

may cause creation of jagged edges, called 'jaggies'. Volume graphics is similar to rastor graphics - objects are represented as a series of voxels, and not by their shape. Thus, when volumes are rotated or otherwise transformed, there is a chance that we shall get jaggies on our resulting volume. Presence of such artifacts is highly undesirable. Anti-aliasing refers to the process of smoothing out such jagged edges so that they are not as obvious to the eye.

As discussed in the preceeding chapters, **ReconstructEuclid** has a provision for using a Gaussian smoothing kernel to reduce the effect of 'jaggies'. The code can be compiled with the kernel set to 1, 3 or 5 voxels. Setting the kernel to 1 implies direct reconstruction without smoothing.

In Figure 5.20 we present results from reconstructions of the sample cube using the same rotation as in the last section, but with added kernel smoothing effects. A subsection of the same images, magnified 8 times over, is shown in Figure 5.21.

Figure 5.7: Visible Human Dataset, with Hands Stretched Outwards

Figure 5.8: Visible Human Dataset, a Frame from the Wave Sequence

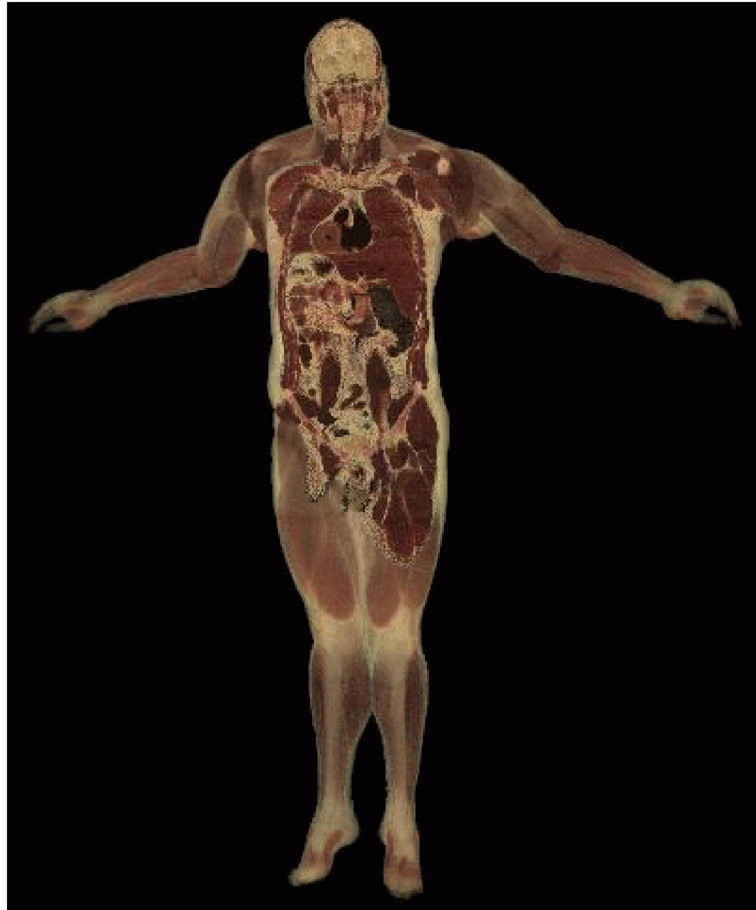

Figure 5.9: Internal Details in Visible Human Animation Volumes

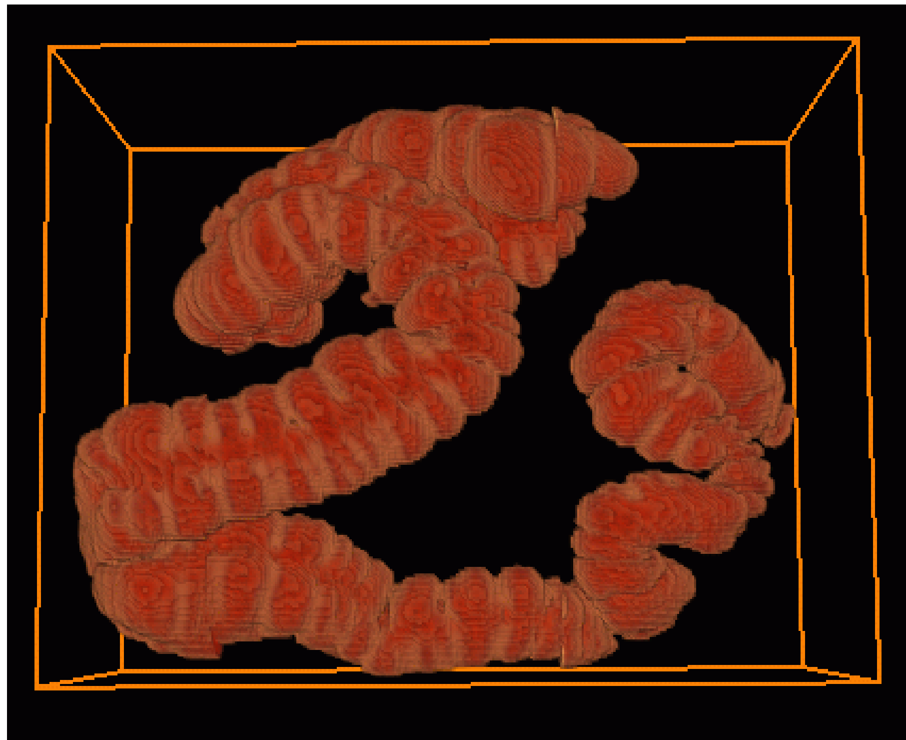

Figure 5.10: The Dataset, as Obtained, After Removing Blank Envelope

Figure 5.11: Surface of the Colon Dataset, Extracted After Segmentation

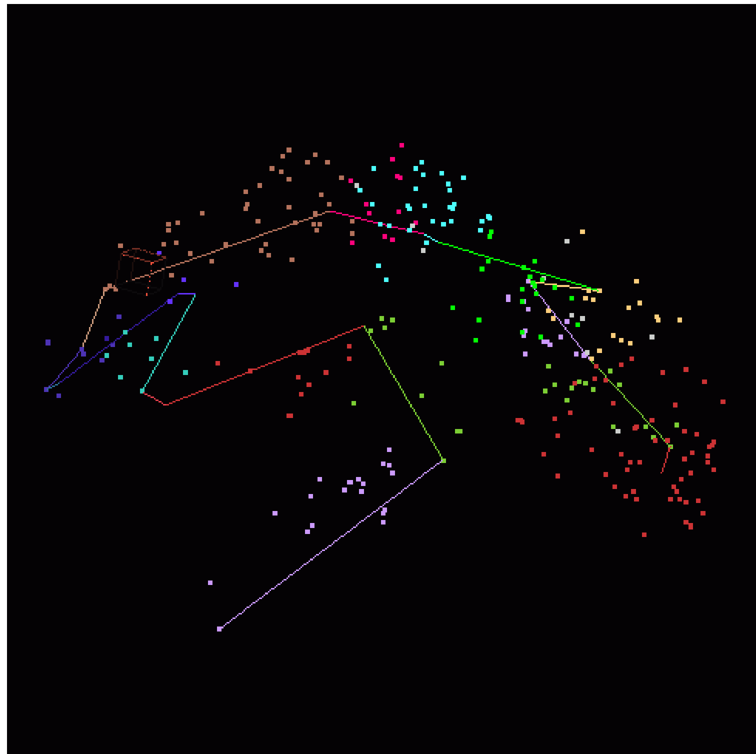

Figure 5.12: Articulated Skeleton and a Few Skeleton Points of the Colon

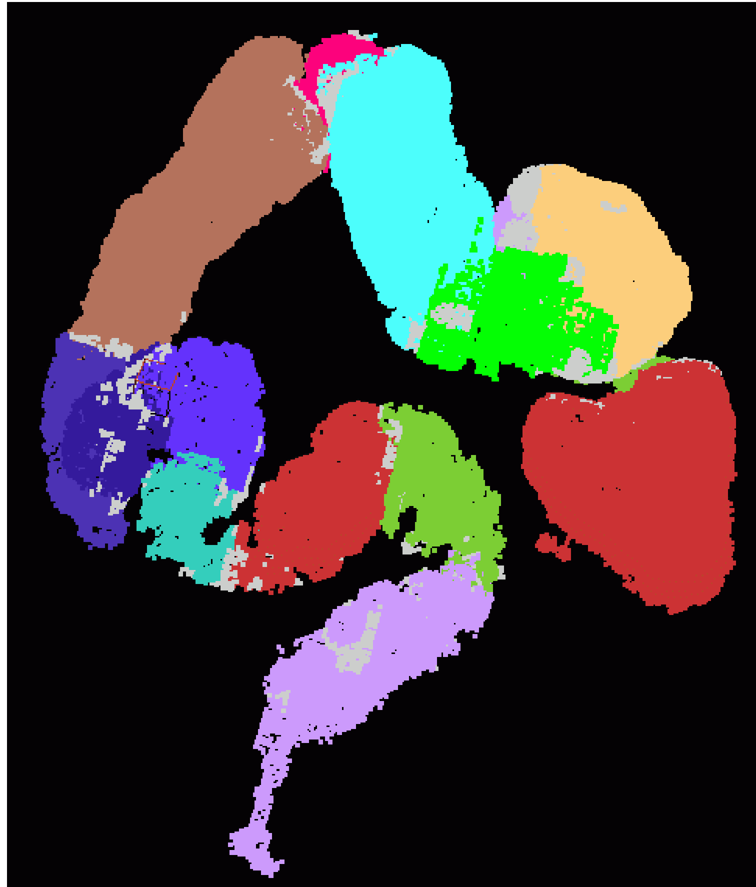

Figure 5.13: Final Connectivity Information of the Colon Skeleton

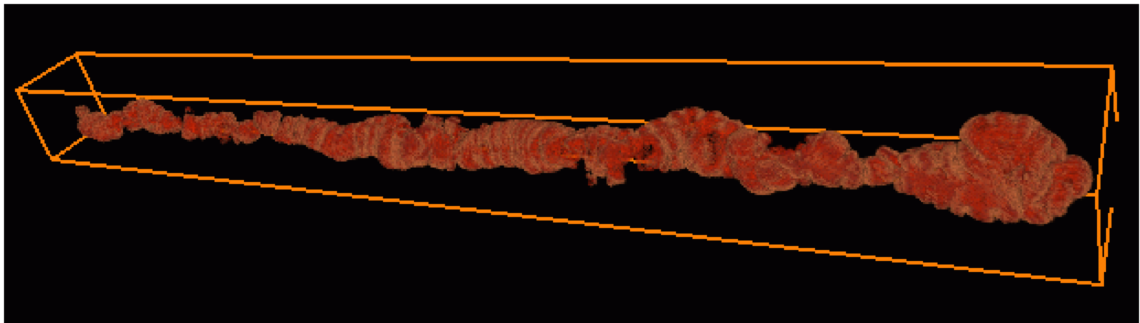

Figure 5.14: The Colon Dataset, After Uncoiling with the Animation Pipeline

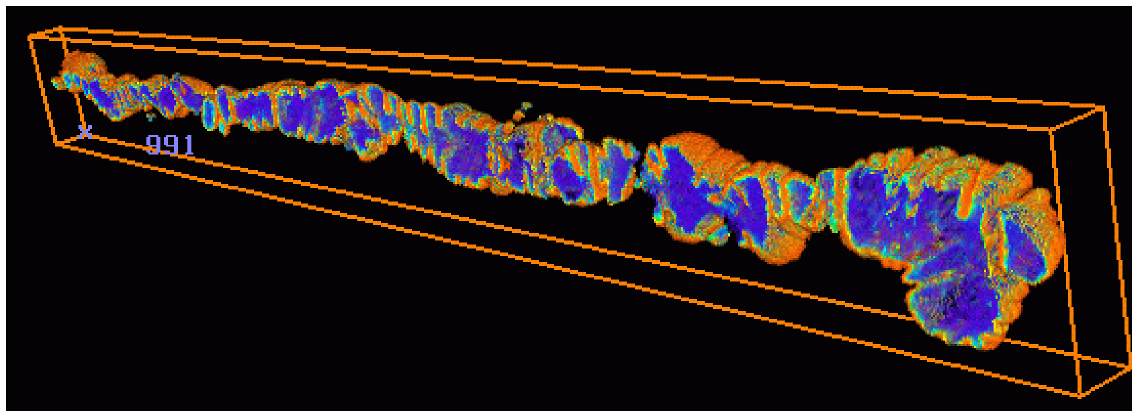

Figure 5.15: Cut-away View of the Uncoiled Colon

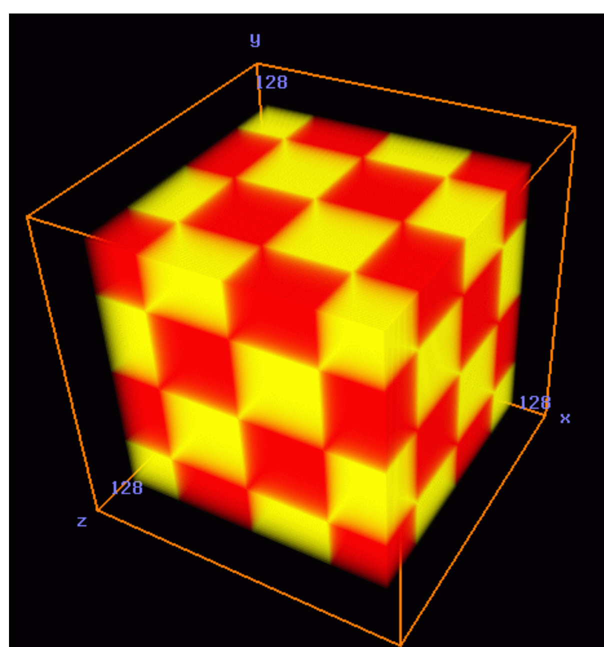

Figure 5.16: Cube Volume to Test Aliasing Results, Viewed from Front Top Right. Volume size is 128x128x128

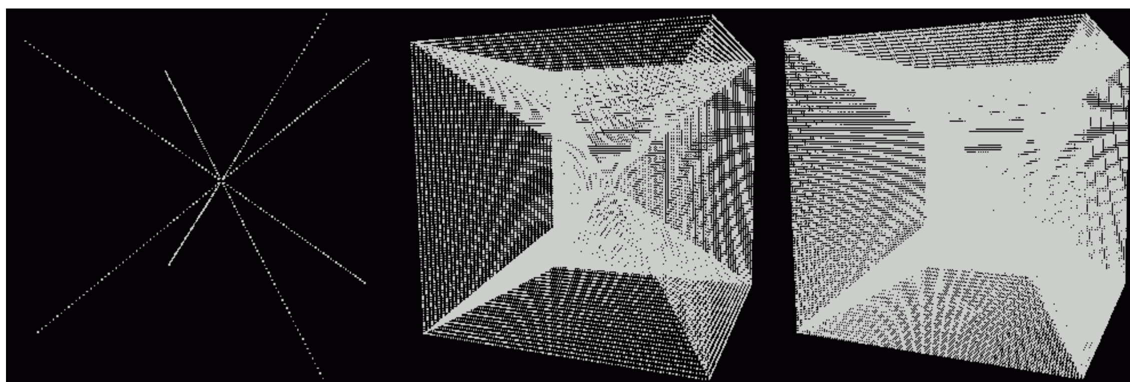

Figure 5.17: Euclidean Skeletons of the Aliascube, at Thinness Levels of 1.50, 1.35 and 1.10

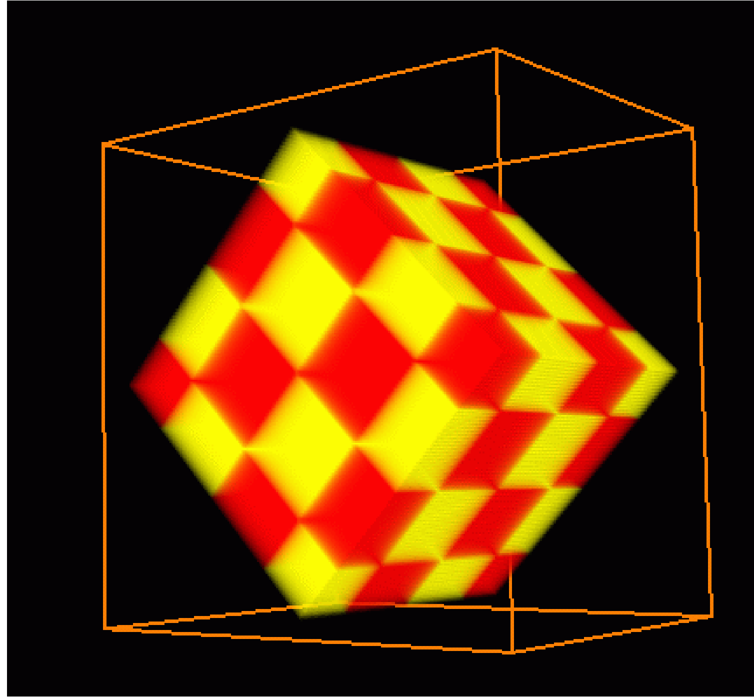

Figure 5.18: The Transformed Reconstruction of the Sample Cube

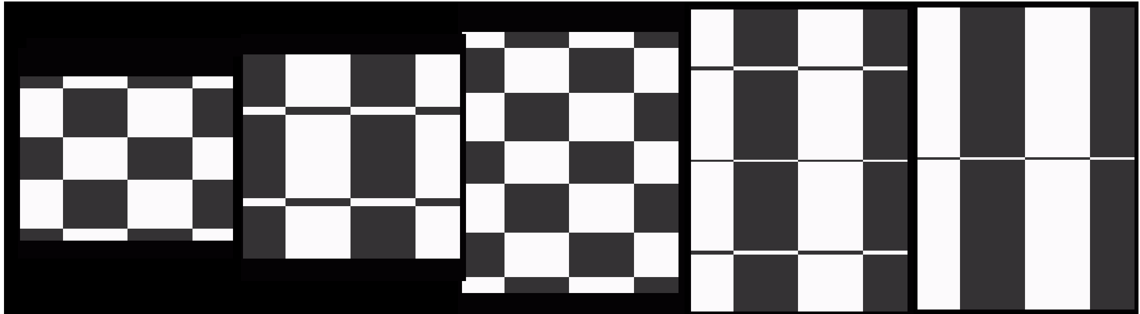

Figure 5.19: Slices from the Transformed Reconstruction of the Sample Cube Dataset

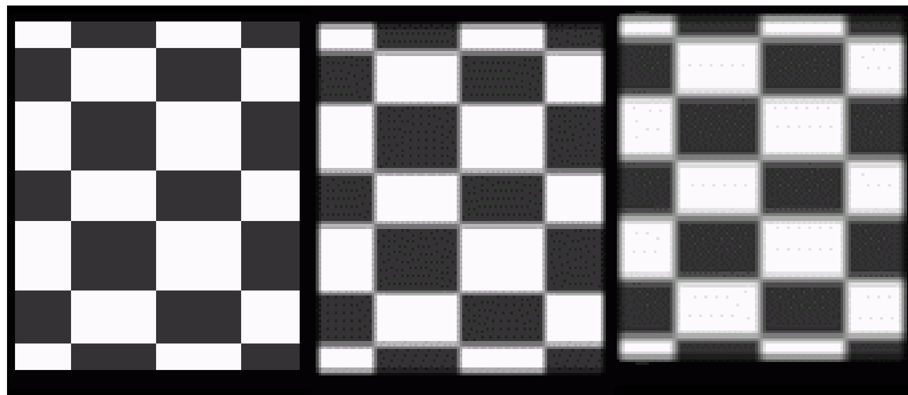

Figure 5.20: Slices in the Y-Z Planes, different X-values, from the Cube Reconstruction, Using Smoothing Kernel of 1 (No Smoothing), 3, and 5

Figure 5.21: Subsections of Figure 5.20, Magnified 8 Times. Note that Some of the Minor Artifacts Were Caused by the Magnification in **xv**

Chapter 6

Conclusions and Future work

As a result of this thesis work, the algorithmic bottlenecks in the reconstruction stage of the volume animation pipeline have been removed. The reconstruction code itself has been optimized for speed on the Sun E10k machines, and has been tested out with various datasets.

The reconstruction process is still the bottleneck for the volume animation pipeline. To improve upon the reconstruction time even further, it will be necessary to make use of hardware acceleration methods. By the use of hardware, it may be possible to transform an entire sphere from the original dataset into the transformed volume, without having to scan through every voxel of the final volume. Specialized hardware routines may be able to take care of problems of forward mapping, thereby making the order of reconstruction proportional to the size of the volume alone. If this can be achieved, the current major speed deterrent - presence of overlapping spheres - will no longer hinder the process.

Appendix A

User Manual

A.1 AddHeader.sh

Source: /teal/caip10/ksen/work/volume/bin/AddHeader.sh

Binary: /teal/caip10/ksen/work/volume/bin/AddHeader.sh

Author: Kundan Sen

Usage: AddHeader.sh [original volume] [xSize] [ySize] [zSize] [new volume]

Adds a header of exactly 100 bytes to the volume. The header is ASCII text, and contains the x, y and z dimensions of the volume that follows. Volumes with the 100-byte headers are conventionally named as .VOL - for instance, the volume newcubes.vol would become newcubes.VOL with the header. Since the header is ASCII data, to check the size of the volume, simply do a 'head -1' on the volume with the header.

A.2 AddSkel2Vol

Source: /teal/caip10/ksen/work/volume/src/AddSkel2Vol.cpp

Binary: /teal/caip10/ksen/work/volume/bin/AddSkel2Vol

Author: Kundan Sen

Usage: AddSkel2Vol [old volume] [xSize] [ySize] [zSize] [mskel file] [new volume] [skelValue (255)]

Takes a gray volume, and a skeleton file, and inserts the points in the skeleton into the volume. By default, these points get a value of 255, but the user can override this by specifying the alternate value on the command line.

To view the output properly, use a colormap that gives a high alpha value to the data value representing the skeleton points (255, by default), and a lower-than-usual alpha value to the rest of the colormap. This will make the skeleton points stand out, and the rest of the volume more transparent.

A.3 AlphaFactor

Source: /teal/caip10/ksen/work/volume/src/AlphaFactor.cpp

Binary: /teal/caip10/ksen/work/volume/bin/AlphaFactor

Author: Kundan Sen

Usage: AlphaFactor [inputFile] [factor] [outputFile]

Take in an alpha file and reduce all the alpha values by the given fraction. The files are expected as ASCII, non-indexed format suitable for loading into **icol**.

A.4 Bob

Source: not available

Binary: http://www.arc.umn.edu/gvl-software/gvl.tar.gz

OR: /teal/caip11/gagvani/quota/packages/bob/bin/bob

Author: University of Minnesota

Usage: bob -s [xSize]x[ySize]x[zSize] [volume file]

Bob was developed by Graphics and Visualization Lab of the Army High Performance Computing Research Center (AHPCRC) of University of Minnesota. Documentation on bob can be found at the developer's website at http://www.arc.umn.edu/gvl-software/bob.html. This is a very easy to use 3-D visualization toolkit that runs on Silicon Graphics machines.

A.5 combineToBinaryVol.sh

Source: /caip/u60/ksen/scripts/combineToBinaryVol.sh

Binary: /caip/u60/ksen/scripts/combineToBinaryVol.sh

Author: Nikhil Gagvani, Kundan Sen

Usage: combineToBinaryVol [source directory] [destination volume file]

A.6 connect2identitytskel

Source: /caip/u60/ksen/male/src/connect2identitytskel.cpp

Binary: /caip/u60/ksen/bin/connect2identitytskel

Author: Kundan Sen

Usage: connect2tskel [connectFile] [transFile] [new tskelFile]

Transforms the connect file information into a tskel file containing identity transformations. Since the transformations are not taken from the motion capture, but default to identity matrices, the reconstruction should give the original object. This is particularly useful to see if the extraction of information from the motion capture sequence is working properly, since a mismatch will yield a non-recognizable volume after reconstruction.

Of course, reconstruction of an incorrectly transformed sequence will produce a non-recognizable volume even with the transformations in place, but then we can not localize the error in the pipeline- it can be with the transformations, with the connectivity information, or with the reconstruction program itself.

A.7 connect2tskel

Source: /caip/u60/ksen/male/src/connect2tskel.cpp

Binary: /caip/u60/ksen/scripts/connect2tskel

Author: Kundan Sen

Usage: connect2tskel [connect file] [trans file] [new tskel file]

Takes the connect file and the transformations file, and merges the data to form the transformed skel, or tskel, file. The transformations file (*.trans) and the connect file (*.connect) must be based on the same articulate skeleton, and the root nodes must appear

in the same sequence in either file.

Root nodes that are not connected to any other skeleton points are handled correctly by this program.

A.8 Convert2Binary

Source: /caip/u60/ksen/male/src/Convert2Binary.cpp

Binary: /caip/u60/ksen/bin/Convert2Binary

Author: Kundan Sen

Usage: Convert2Binary [old volume] [xSize] [ySize] [zSize] [new volume] [[replace by (1)]]

This program is essentially a saturating high-pass filter - it scans the volume for all voxel values that are non-zero, and replaces the non-zero values by the replace parameter passed. In the absence of the last parameter, the value defaults to 1.

To convert a volume to a binary (0-1) volume for the purpose of skeletonization, run the program with the last parameter as '1', or leave out the last parameter entirely. To view a binary (0-1) volume in conventional tools like **view.tcl** or **bob** without having to create a new colormap, run this program to replace the 1's with 255, or some other high number, so that the color defaults to a light shade in the automatic colormap.

A.9 convertall.sh

Source: /caip/u60/ksen/scripts/convertall.sh

Binary: /caip/u60/ksen/scripts/convertall.sh

Author: Nikhil Gagvani, Kundan Sen

Usage: convertall.sh

This script handles automatic conversion of *.dxf frames to *.tskel files for a given motion capture sequence. The dxf files are placed in the working directory, and the script copied to the same. The script may need some modifications, such as:

set connectFile='/caip/u60/ksen/euclid/half/connects/euclid.40.culled.connect'

set connectionsFile='/caip/u60/ksen/euclid/half/connections/half.connections'

set bonesFile='/caip/u60/ksen/euclid/half/bones/save.euclid.half.iv'

The three lines above set the most common parameters that vary from one sequence to another. The "connectFile" variable points to the file saved from **skelselect** that contains the connections guidelines to be followed for mapping the current sequence. This file defines the skeleton cloud associated to each bone in the articulated skeleton, and thus indicate how the entire skeleton should map to the motion capture skeleton extracted from the dxf files.

The "connectionsFile" variable is not expected to change, unless a new articulate skeleton has been designed. This file contains information on how the named joints are connected to each other. A few lines from the file are shown here:

NEW_L_THIGH_BOX NEW_L_KNEE_BOX

NEW_R_KNEE_BOX NEW_R_ANKLE_BOX

NEW_L_KNEE_BOX NEW_L_ANKLE_BOX

NEW_R_ANKLE_BOX NEW_R_FOOT_BOX

NEW_L_ANKLE_BOX NEW_L_FOOT_BOX

The order of the points is important, so this file should not be changed unless the user is sure of what changes are to be carried out.

The "bonesFile" contains the articulate skeleton points in the format acceptable by **Inventor**. If the points in the articulate skeleton are adjusted to compensate for minor breakage, or shifted for some other reason, the coordinates of the bones file have to be updated to reflect the current coordinates.

A.10 Count

Source:/caip/u60/ksen/male/src/Count.cpp

Binary:/caip/u60/ksen/bin/Count

Author: Kundan Sen

Usage: Count ¡volumeFile¿ ¡xSize¿ ¡ySize¿ ¡zSize¿

Counts the zero and non-zero voxels in the volume, and prints the result to the standard output. Also prints a histogram showing the general distribution of data in the volume, in steps of 10, within the unsigned character range of 0-255. The program reads in the volume in layers, minimizing memory requirements while processing the data.

A.11 CropVolume

Source: /teal/caip10/ksen/work/volume/src/CropVolume.cpp

Binary: /teal/caip10/ksen/work/volume/bin/CropVolume

Author: Kundan Sen

Usage: CropVolume [old volume] [xSize] [ySize] [zSize] [xStart] [yStart] [zStart] [xStop] [yStop] [zStop] [new volume]

This program creates a new bounding box for the volume. If the bounds passed, i.e. the cuboid defined by the two diagonal points [xStart, yStart, zStart] and [xStop, yStop, zStop], are smaller than the volume, a subset of the volume is dumped into the output file. If, on the other hand, the bounds are larger than the current volume, such as negative coordinates for the starting point coordinates, or coordinates larger than the size of the volume for the ending point coordinates, then a suitable padding is applied around the volume. The padding voxels are filled with zeros.

This program can also be used to extract single slices from a volume, by giving full extends on two of the axes, and having a difference of one between the start and stop coordinates of the remaining axis.

A.12 euclidskel

Source: /teal/caip11/gagvani/quota/modeling/euclid/euclidskel.c++

Binary: /teal/caip11/gagvani/quota/bin/euclidskel

Author: Nikhil Gagvani

Usage: euclidskel [volfile] [xs] [ys] [zs] [outfile] [thresh] [measureTimeFlag]

This program performs skeletonization on the volume passed in the parameter *[volfile]* and creates an mskel file containing the multi level skeleton. All voxels having values at least as large as the threshold *[thresh]* are considered for the skeletonization process.

The skeleton produced is in Eucledian metric, and is of multi-resolution format, as is the convention for the mskel file format. To sort the skeleton in the order supported by tools like skelselect, perform the following command after the skeletonization is complete: *sort +4n -r [outfile] [sortedoutfile]*

This will sort the mskel file in decreasing order of thinness, as expected by skelselect. To extract a thicker skeleton from this mskel file, simply extract lines from the head of the file until the desired thinness level has been reached.

A.13 Fill

Source:/caip/u60/ksen/male/src/Fill.cpp

Binary:/caip/u60/ksen/bin/Fill

Author: Kundan Sen

Usage: Fill [old volume] [xSize] [ySize] [zSize] [xStart] [yStart] [zStart] [xStop] [yStop] [zStop] [value]

Fills in the entire cube from [xStart, yStart, zStart] up to, but not including, [xStop, yStop, zStop] with the value passed in the last parameter. Care must be taken before the bounds are entered, since the data will be destroyed within this bound, and replaced by the single data value passed as the parameter [value].

A.14 FillSection

Source: /teal/caip10/ksen/work/volume/src/FillSection.cpp

Binary: /teal/caip10/ksen/work/volume/bin/FillSection

Author: Kundan Sen

Usage: FillSection [volfile] [OutVolFile] [xSize] [ySize] [zSize] [[xStart] [yStart] [zStart] [[xStop] [yStop] [zStop]]]

Fills in the section of the volume within the bounds specified in the parameters with 1's.

Deprecated. Use Fill instead.

A.15 FitVolume

Source: /teal/caip10/ksen/work/volume/src/FitVolume.cpp

Binary: /teal/caip10/ksen/work/volume/bin/FitVolume

Author: Kundan Sen

Usage: FitVolume [vol file] [outputVol] [xSize] [ySize] [zSize] [header]

This program fits each dimension of the volume to the nearest integral power of two that is at least as large as the current dimension. The file is read after an offset equaling the parameter [header] Bytes. The header is not written back to the final volume. The new dimensions are printed to the standard output. It is important to note this information down, preferably by renaming the new volume to reflect the new size (by convention, the name of the volume would be [filename].[xSize]x[ySize]x[zSize].vol), since it would be very difficult to figure out the size once this information is lost.

A.16 flip

Source: /teal/caip10/ksen/work/volume/src/flip.cpp

Binary: /teal/caip10/ksen/work/bin/flip

Author: Kundan Sen

Usage: flip [volume file] [xSize] [ySize] [zSize] [out file] [flip: [X/Y/Z]]

Flips the volume in-place along one of the major axes. Useful when the volume needs to be 'turned', or coordinate differences between the animation phase and the back-end support need to be resolved.

Apart from the usual parameters, the last parameter - [flip] - can take in one of the

values [x, y, z, X, Y, Z], and indicates the axis of flipping of the volume. As with several other programs, since the input and output files are opened for reading and writing at the same time, this program can not be used to do an 'in-place' flip of the volume, i.e. flip the volume into the same filename. Passing the same filename for both the *[volume file]* and *[out file]* parameters will result in loss of the volume data.

A.17 float2uchar

Source: /teal/caip10/ksen/work/volume/src/float2uchar.cpp

Binary: /teal/caip10/ksen/work/bin/float2uchar

Author: Kundan Sen

Usage: float2uchar [old volume] [xSize] [ySize] [zSize] [new volume]

Transforms a volume in floating points into unsigned characters. Floating point volumes are common in tools like AVS (Advanced Visualization Systems). Since all our work in the domain of volume animation are based on unsigned character volumes, the easiest way to import a volume from AVS is by transforming it into unsigned characters.

It may be noted that since the data is now normalized into the interval [0-255], minute differences in data values are lost. In particular, if the data has a very wide range (where 'range' is defined as the difference between the maximum and the minimum data value), but most of the data is concentrated over a very small interval, normalization will lead to loss of important characteristics of the data. It is advisable to pass the volume through a histogram utility, and a band-pass filter, to isolate the data value region of interest, before converting the volume into unsigned characters. For example, if the data range is from -65536 to +65536, but 90% of the data and/or the region of interest is between 1.0 and 2.0, pass the volume through a band pass utility, normalizing the values to within the range [1.0 -2.0], and then subject the new volume to conversion by float2uchar.

A.18 GenerateHeader

Source: /teal/caip10/ksen/work/volume/src/GenerateHeader.cpp

Binary: /teal/caip10/ksen/work/volume/bin/GenerateHeader

Author: Kundan Sen

Usage: GenerateHeader [xSize] [ySize] [zSize] [outVol]

Generates a 100-byte header containing the volume dimensions as ASCII text, and prints it to the output volume [outVol]. This program is called internally from AddHeader.sh, and need not be called directly unless it is desired to generate the header only, and not append the volume data to it.

A.19 getskel

Source: CVS Repository

Binary: /teal/caip11/gagvani/quota/bin/getskel

Author: Nikhil Gagvani

Usage: getskel [filename] [thinness] [flag] [measureTimeFlag]

For binary files : getskel -b filename thinness xsize ysize zsize [flag] [measureTimeFlag]

This program performs skeletonization on ascii or binary files. The output is a skeleton file of fixed thinness parameter value, and uses the 3-4-5 metric for computation of distance transforms. **Deprecated**. Use euclidskel instead.

A.20 getVoxelCount

Source: /caip/u60/ksen/male/src/getVoxelCount.cpp

Binary: /caip/u60/ksen/bin/getVoxelCount

Author: Kundan Sen

Usage: getVoxelCount [volumeFile.vol] [[xSize] [ySize] [zSize]]

OR: getVoxelCount [volumeFile.VOL]

Counts the number of zero and non-zero voxels in a volume, and prints the result to standard output. **Deprecated** - use **Count** instead, after removing the header for the .VOL file.

A.21 Icol

Source: not available.

Binary: /teal/caip11/gagvani/quota/bin/icol

Author: University of Minnesota

Usage: icol

Icol is an interactive colormap editor program, developed by Graphics and Visualization Lab of the Army High Performance Computing Research Center (AHPCRC) of University of Minnesota. Documentation on icol can be found at the developer's website at http://www.arc.umn.edu/gvl-software/icol.html.

A.22 InterpolateAlpha

Source: /teal/caip10/ksen/work/volume/src/InterpolateAlpha.cpp

Binary: /teal/caip10/ksen/work/volume/bin/InterpolateAlpha

Author: Kundan Sen

Usage: InterpolateAlpha [firstAlpha] [lastAlpha] [no. of steps] [[startIndex]]

Generates a number of alpha-map files, which form a linear gradient from the starting alphamap, [firstAlpha], to the final alphamap, [lastAlpha]. The parameter [no. of steps] indicates the number of alpha files to be generated to bridge the gap between the end-values. The parameter [startIndex] sets the user preference in naming the intermediate alpha files - if this is omitted, the numbering starts from the default, frame0.alpha. The files are compatible to the colormap editing utility, 'icol', when fired up in alphamap-editing mode.

If it is desired to keep the length of the numeric section of the filename constant, for example, name files from frame000.alpha to frame123.alpha, uncomment the first line in

the processing loop, comment out the second line, and recompile the program.

A.23 InvCMap

Source: /caip/u60/ksen/color/InvCMap.cpp, http://www.cs.utah.edu/gdc/projects/urt/

Binary: /caip/u60/ksen/color/InvCMap

Author: Utah Raster Toolkit

Usage: see page http://www.cs.utah.edu/gdc/projects/urt/

This program takes in the colormap in the file "*reducedmap.txt*", where the file mentioned has 255 lines, each having red, green, and blue values as integer constants. It now produces the Voronoi diagram of the colormap provided, in a 24-bit colorspace, i.e. 0-255 in each axes, as a volume with unsigned characters, "*colormap.vol*". To map a 24-bit color to it's 8-bit equivalent, look up the [R,G,B] voxel in *colormap.vol* - the data value at the voxel gives the 8-bit equivalent that maps the 24-bit volume to the colormap in *reducedmap.txt.*

A.24 ivbbox

Source: contact Nikhil Gagvani

Binary: /teal/caip11/gagvani/quota/bin/ivbbox

Author: Nikhil Gagvani

Usage: ivbbox [Inventor file]

This program takes in an inventor file, and prints the bounds of the object contained within the file to standard output. To align two inventor representations of the same volume, for example, to align an isosurface of the visible male with the skeleton, use ivbbox to compute the bounds of each, and shift the origins of both to their respective centers.

A.25 ivskeladjust

Source: /caip/u60/ksen/male/src/ivskeladjust.cpp

Binary: /caip/u60/ksen/bin/ivskeladjust

Author: Nikhil Gagvani, Kundan Sen

Usage: ivskeladjust ivfile1 ivfile2 outfile

Compares corresponding segments from the two Inventor files passed, and makes the segments of the first file match in length to those of the second file. Segments are stretched or compressed, while maintaining their connectivity with other segments within the file. The output to standard output indicates how well the segments were matched up. It is imperative to have the two files contain the segments in exactly the same order, since the nth segment of the first file is matched up to the nth segment of the second file and adjusted accordingly. If there are remarkable differences in the two columns printed to the standard output, then the segments in the two files may not be in the same order.

Note: This file may require modifications that are specific to a particular motion sequence. In particular, the dependency graph of the nodes on each other are stored in a static array in this program, and may require modifications if the number of nodes in the skeleton is changed, or the dependency is changed in any other manner. In such cases, it is advisable to make a local copy of the source code in the working directory for the particular animation, and preserve the code in the archives of the animation.

A.26 meltMan

Source: /caip/u60/ksen/male/src/meltMan.cpp

Binary: /caip/u60/ksen/bin/meltMan

Author: Kundan Sen

Usage: meltMan [tskelFile] [numFrames (20)] [sequenceHeader (meltFrame)]

Creates a 'melting' effect by linearly reducing the distance transforms of the tskel file. Running the program with just the first parameter produces 20 frames, named from *meltframe0.tskel* to *meltframe19.tskel.* For better results, couple with multiVolumeMelt.

A.27 MergeMaps

Source: /teal/caip10/ksen/work/volume/src/MergeMaps.cpp

Binary: /teal/caip10/ksen/work/volume/bin/MergeMaps

Author: Kundan Sen

Usage: MergeMaps [colormap] [alphamap] [lutFile]

Simple program to take an ASCII non-indexed colormap and similar alphamap, both outputs of **icol**, merges them to the R-G-B-A format, and prints to the lutFile parameter passed. The alpha values are normalized to the 0-1 domain, since the lutFile format demands this. The lutFile may now be passed as a rendering parameter to **Volumizer**-based programs.

A.28 Merger

Source: /teal/caip10/ksen/work/volume/src/merger.cpp

Binary: /teal/caip10/ksen/work/volume/bin/merger

Author: Kundan Sen

Usage: Interactive mode: merger

OR: Non-interactive mode: merger [inputVolumeFileName] [inputSkelFileName] [outputVolumeFileName] [xSize] [ySize] [zSize] [headerLength]

Once the program is started, it prompts for the input volume filename, the input skeleton filename, and the output volume filename. The program then decreases the intensity of the original volume, by decreasing the value at each voxel to half of the current value, and then adds the skeleton points into the volume, marking them with full intensity (250). The resulting volume is then dumped to the new volume file.

In the non-interactive mode, if all the parameters are passed to the program, the command line entry is disabled, and the program executes to competion.

This program is useful to see how well the skeleton registered with the volume. When the final volume is viewed in grayscale mode, we can clearly make out the skeleton points

within the volume, specially if a low transparency value (alpha) is used in the colormap.

Note: this program expects a skeleton file, not an mskel file. To convert a mskel file to a skel file, drop the last column of the mskel file.

A.29 multiVolumeMelt

Source: /caip/u60/ksen/male/src/multiVolumeMelt.cpp

Binary: /caip/u60/ksen/bin/multiVolumeMelt

Author: Kundan Sen

Usage: multiVolumeMelt [volumeHeader] [volStartIndex] [volEndIndex] [numberOfFrames] [frameHeader] [acceleration] [[desiredFrame]]

This program is a toolkit for insertion of a parametric crumble effect into an existing volume animation sequence. The existing volumes are taken by appending integers between parameters *[volStartIndex] and [volEndIndex]* to the string parameter *[volumeHeader]*, and adding a ".vol" to the filename. The parameter *[numberOfFrames]* indicates how many frames the crumbling effect produced from the existing frames should span. *[acceleration]* indicates how fast the voxels should fall ('crumble') to the base, typically the feet of the man. If the last parameter, *[desiredFrame]*, is given, instead of generating the entire sequence, only this frame is generated, and named as the first frame of the sequence.

This program shifts the voxels to the base ('floor') of the volume, with a speed controlled by the acceleration parameter. When the voxels reach the base, they form a heap, the height of which is proportional to the number of voxels that reached that point. The effect produced is that of the volume crumbling and leaving a conical heap on the floor. The color of each of the heap voxels is the cumulative of the data values of all the voxels that crumbled into it, with a cut-off set at 255. If this program is coupled with the volumes generated by meltMan, the effect is one of simultaneous melting and crumbling - a few frames of the melting sequence can be fed to multiVolumeMelt to produce a large number of crumbling volumes. When rendering this sequence, the best effects are produced when the notion of a floor is added at the correct position of the volume, so that the heap formed at the end

rests on the floor.

A.30 NewSort

Source: /caip/u60/ksen/color/newSort.cpp

Binary: /caip/u60/ksen/bin/newSort

Author: Kundan Sen

Usage: newSort [inputColormap.txt]

The input colormap is an ASCII text file, with red, green, and blue values stored as integers. The program sorts the colors, using one or more of the several modules built into it, and writes the sorted colormap, with the old mapping index, to standard output. To re-map a volume in the old colormap to the new colormap, split the output into two text files, the first containing the first column only, forming a lookup table for the remap, and the second containing all the other columns, forming the new sorted colormap. Follow up with **remapVolume**, with the first file passed as the parameter 'remapIndexFile', to convert the volume to the new colormap.

A.31 OneVolume

Source: /teal/caip10/ksen/work/volume/src/OneVolume.cpp

Binary: /teal/caip10/ksen/work/volume/bin/OneVolume

Author: Kundan Sen

Usage: OneVolume [baseName] [startIndex] [stopIndex] [step] [outVol] [threshold]

Combines all the volumes that start with the [baseName], and are followed by numbers that are in the range of [startindex] and [stopindex], into a single volume. Voxel data values that fall below the threshold are set to zero.

A.32 Peek

Source: /caip/u60/ksen/male/src/Peek.cpp

Binary: /caip/u60/ksen/bin/Peek

Author: Kundan Sen

Usage: Peek [volume file] [xSize] [ySize] [zSize]

Once started up, keeps a file pointer to the volume file passed, and asks the user for a query point in the volume. If this point is within the bounds of the volume, then the data value at this point is printed. The volume is expected as a binary file with unsigned characters, one value per voxel. The file may be modified by Poke or Fill while Peek is running, and the changes made by the other programs can be seen by all future queries by Peek without having to re-start the program. Concurrency issues while reading/ writing to the disk are not dealt with, but are left to the operating system.

A.33 polyr

Source: not available

Binary: /teal/caip11/gagvani/quota/bin/polyr

Author: Jesper James Jenson

Usage:

This program produces isosurfaces of volumes efficiently. The parameters help us choose the resolution of the isosurface, by limiting the number of triangles. Keeping the number of triangles too high produces a huge isosurface file, which takes a lot of time to load up into the viewer programs, like **ivview**. Keeping the number too small yields a very coarse isosurface with most of the details of the structure lost.

A.34 Poke

Source:/caip/u60/ksen/male/src/Poke.cpp

Binary:/caip/u60/ksen/bin/Poke

Author: Kundan Sen

Usage: Poke [volume file] [xSize] [ySize] [zSize]

Once started up, keeps a file pointer to the volume passed, and asks the user for a point within the volume, and the new data value to be saved at that point. If the entry is within the bounds of the volume, the voxel value is updated. The value is expected to be within the range of unsigned characters, i.e. [0-255]. The program may be used with Peek, to update data values and verify the update. Concurrency issues while reading/ writing to the disk are not dealt with, but are left to the operating system.

A.35 Reconstruct1

Source: /caip/u60/ksen/male/src/euclid/Reconstruct1.cpp

Binary: /caip/u60/ksen/bin/Reconstruct1

Author: Kundan Sen

Usage: /teal/caip10/ksen/tec/recon/summer00/Reconstruct1 [tskelfile]
[origObject] [outObj] [xSize] [ySize] [zSize]
[xMin] [xMax] [yMin] [yMax] [zMin] [zMax]

Fastest reconstruction program in the 3-4-5 metric. The program has most of the optimizations in ReconstructEuclid, but works for the 3-4-5 metric instead of Euclidean. The parameters are self-explanatory, since they are the basic properties of the volume to be reconstructed. *[origObject]* refers to the original volume, the basis of the reconstruction, *[outObj]* is the new volume to be generated, *[xSize] [ySize] [zSize]* form the size of the original volume, and *[xMin] [xMax] [yMin] [yMax] [zMin] [zMax]* indicate the bounds of the volume to be reconstructed. It is important to compute the bounds correctly by using a program like tskelbounds, and to keep in mind that the entire process has to be in the 3-4-5 domain as far as computation of distance transforms are concerned.

The program can be modified in several ways to have customized reconstructions. The defined values HOTSPOTS, BINRECON, SAMPLED, MOTTLED in the first few lines

of the program indicate the chosen mode of reconstruction. Only one of these parameters should be enabled at any time, and the program compiled to reflect the change. The value PADDING, following the lines above, indicates if the volume bounds need to be stretched outwards (padded) to produce an envelope of boundary voxels around the volume.

The program will print errors to standard output. The most common error is voxels falling off the bounds of the volume. Is this occurs for a small number of voxels, and the points are off by single-digit voxels (i.e. 1-9 voxels), the easiest way to solve the problem is to increase the padding. However, if the errors are larger, they are probably due to a more important reason - check that the tskel file is in 3-4-5 metric, and that tskelbounds is run in the correct computation mode.

The only difference from Reconstruct (deprecated) is the addition of a stencil buffer to incorporate the 'relaxation rule' - when a voxel falls within the boundaries of several spheres, the sphere having the minimum distance from it's center to the voxel in question is designated to be the owner of the voxel. See details in the thesis.

A.36 ReconstructEuclid

Source: /caip/u60/ksen/male/src/euclid/ReconstructEuclid.cpp

Binary: /caip/u60/ksen/bin/ReconstructEuclid

Author: Kundan Sen

Usage: /teal/caip10/ksen/tec/recon/summer00/ReconstructEuclid [tskel file]
[origObject] [outObj] [xSize] [ySize] [zSize]
[xMin] [xMax] [Min] [yMax] [zMin] [zMax]

This is the fastest reconstruction program to-date for handling sampled reconstruction in the Euclidean metric of distance transform computations. The basic scan-fill algorithm is the same as in Reconstruct1, modified to accommodate Euclidean transformations instead of the 3-4-5 notation. This lead to a few more optimizations, which are discussed in the thesis.

All parameter values have the exact same implications as in Reconstruct1, except that

the tskel file has to be in the Euclidean metric.

A.37 remapVolume

Source: /caip/u60/ksen/color/remapVolume.cpp

Binary: /caip/u60/ksen/bin/remapVolume

Author: Kundan Sen

Usage: remapVolume [old volume] [xSize] [ySize] [zSize] [remap index File] [new volume]

This program is useful when the colormap of a volume needs to be changed. Since volumes map to colormaps with mapping indices, the mapping indices need to be converted to the new values. The parameter *[remap index file]* is an ASCII file containing 255 values, the nth value replacing all occurrences of 'n' in the old volume. The resulting volume is written to the file *[new volume]*.

A.38 ReverseData

Source: /teal/caip10/ksen/work/volume/src/ReverseData.cpp

Binary: /teal/caip10/ksen/work/volume/bin/ReverseData

Author: Kundan Sen

Usage: ReverseData [old volume] [xSize] [ySize] [zSize] [new volume]

Reverses the data in a volume by subtracting the current data value from 255, the maximum value for unsigned characters. The new volume is then written to the file *[new volume]*. Since both the input and the output files are opened at the same time, the destination filename has to be different from the source filename, otherwise the data in the source filename will be erased before it can be read in.

A.39 rmhdr

Source: Contact Nikhil Gagvani

Binary: /teal/caip11/gagvani/quota/bin/rmhdr

Author: Nikhil Gagvani

Usage: rmhdr [infile] [outfile] [hdrsize]

Removes the mentioned number of Bytes from the start of the file, and prints out the rest of the file to the new file.

A.40 Skeleton.tcl

Source: /teal/caip10/ksen/apps/vtcl/src/skeleton.tcl

Binary: /teal/caip10/ksen/apps/vtcl/src/skeleton.tcl

Author: Kundan Sen

Usage: skeleton.tcl

This program was to put an easy to use interface to the old volume animation pipeline, making it simple to manage repeatative tasks like entering the original volume, it's size, and the like, at every stage in the pipeline.

The program started with an interface like this:

Once the base volume filename is entered, the corresponding names for the other files to be created in the pipeline are displayed in the appropriate textfields. The first time a volume is loaded, the user has to enter the size of the volume in the size textfields. Once this is done, the interface stores this information in a .size file, so that in the future, the size of the volume is read in directy from the file.

The program lists several buttons below the textfields, which show the next program to be run in the pipeline once all the stages above have been completed. The files that already exist have the file indicator bar to the left of the file description colored green, while those that are missing have the bar in red. The filename textfields at each stage are followed by a file browse button, and by a button that points to the viewer for that particular stage, if any. The large "X" at the bottom of the dialog closes the user interface.

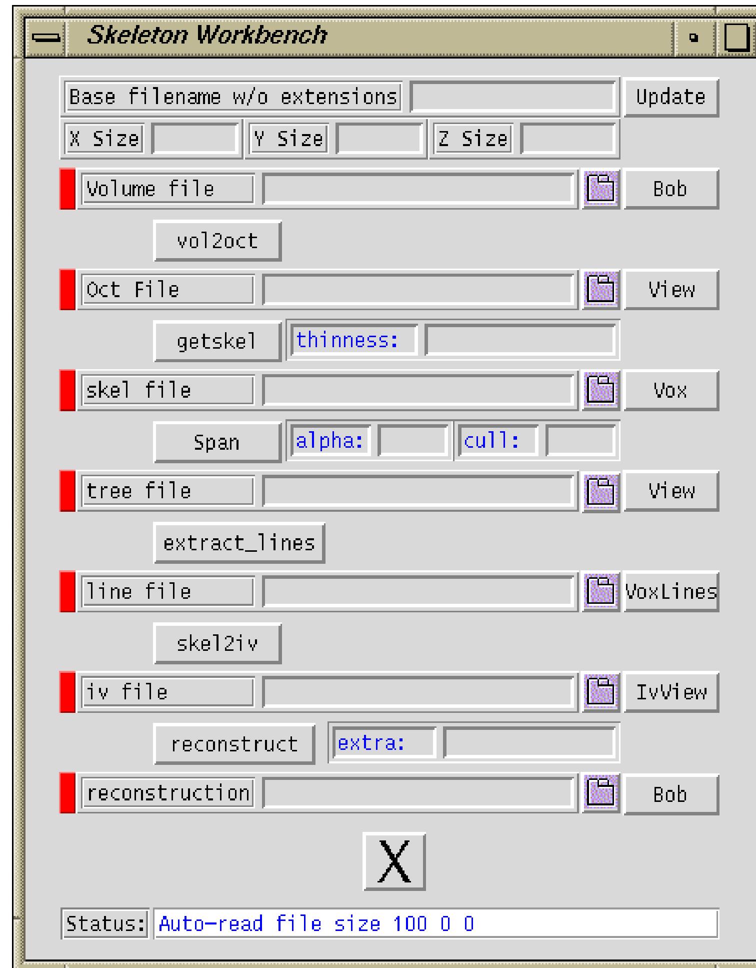

Figure A.1: Skeleton.tcl - User Interface for the Old Volume Animation Pipeline

A.41 Skelselect

Source: CVS Repository

Binary: /teal/caip11/gagvani/quota/modeling/mskeltools/skelselect

Author: Nikhil Gagvani

Usage: skelselect [sorted mskelfile] [Bones File] [-s skelfile] [-c connectfile]

This is, beyond doubt, the most complicated program in use in the volume animation pipeline, and deserves detailed use guidelines. The program works in 3 modes, as follows:

A.41.1 Selecting the Articulate Skeleton

Selection of the points that go on to create the articulate skeleton is easiest when the skeleton is loaded up along with the isosurface of the binary segmented volume. This is because the skeleton itself does not allow us to understand the exact locations of the joints in the body as well as the isosurface.

For this mode, skelselect is executed in this fashion:

skelselect [mskel file] -s [isosurface file]

It is important to make the isosurface non-pickable by inventor. If the surface is pickable, then we shall not be able to penetrate it and pick the skeleton points that are contained within this shell. This is accomplished by adding the following lines to the beginning of the isosurface Inventor file:

```
PickStyle {
        style UNPICKABLE
}


Material {
        transparency 0.5
}
```

The lines may be inserted anywhere before the start of the list of points.

Once the surface and the skeleton points have been loaded up, select the 'Select Skel' mode of operation. In this mode, every pair of points selected by clicking the mouse on them will be joined with a white line, and will be saved as a part of the bones file. By checking the box marked 'Remove Lines', an incorrect line may be erased by clicking on it. To form connection A - B - C, connect A - B, select B again, and connect to C. To save the articulate skeleton, click on the 'save skel' button. This saves the skeleton as a save.iv file, in **Inventor** format. Here are some hints and tips to make the process of selecting the articulate skeleton easier:

• Before selecting a point, make sure there is no parallax error in visualizing that point - since all points look roughly the same, it is very difficult to make out points in the hands

from points well within the abdomen area. Rotate the volume in every axis until it is certain that the point is located in the correct region.

• The skeleton in it's full thinness is too cluttered to work upon. It is easier to search for favorable points using a skeleton with greater thinness, and gradually decrease the thinness to find a better point that is close to the first one. To change the thinness of the skeleton, drag the thinness slider - the greater the thinness, the lesser the number of visible points.

• When the mouse is clicked on a valid point, the coordinates of the point are printed to the console. A valid point is defined as any point that can be selected - clicking on a line, or the isosurface shell, may not select a valid point. It is important to observe this screen when selecting points, as the coordinates are a sure way of detecting the spatial position of the point and avoid any parallax errors.

• If it is difficult to select the skeleton in proper order, an easier way is to note down the coordinates of each point selected to be a part of the skeleton, and create the bones file by hand, inserting the points in the correct order.

A.41.2 Selecting the Root Node

Once the articulate skeleton has been selected in the previous step, the skeleton is saved, and skelselect is closed. If required, the skeleton file (typically, save.iv) is edited by hand to make the points fall into the same order as mentioned in the connections file.

Once this is done, we have to connect the cloud of skeleton points surrounding each segment / bone of the articulate skeleton, so that all these points are given the same transformation as the bone, when the motion capture information is applied to the bone. For this, we load up skelselect thus:

Skelselect [mskel file] -s [bones file]

Once loaded up, we have to assign indices to the bones, so that the connect file that we shall save contains the segments in the order that is expected by the pipeline. For this, increase thinness to maximum by dragging the thinness slider to extreme right. Only the bones should be visible now. Click on the radio-button marked 'Select Root' to set the mode. Now click on the root nodes in the sequence that has been decided for the animation sequence. In our case, the visible human's right thigh (to the left of the viewer) was selected

as node 0, followed by left thigh, and so on to the palms of the feet, followed by the abdomen and chest segments to the neck and head, followed by the arms traversed exactly like the legs, in right-left alternating manner. It is important to decide on a sequence and stick to it, as every new sequence would require changes to several program source codes.

The segments are colored as they are clicked on. The colors are taken from a palette of 12 colors, so the colors cycle after a while. The program prints out the index of each segment to the console as it is clicked on. These indices should be verified with the expected values.

A.41.3 Selecting the Skeleton Points to Join the Selected Root Node.

Once all the segments have been clicked on (and have become colored), we start connecting the skeleton points to these bones. While still in the previous mode, click on a segment to select that segment as the working bone. Verify the selection by the index printed on the console. Click on the radio-button marked 'Select Cube' to enter the cloud connection mode. Clicking on any skeleton point will now draw a cube centered at that point. The size of the cube can be varied with the slider to the bottom right of the screen. When a satisfactory cube has been placed, click on 'connect' to attach all points within the cube to the working bone segment. Since the boundaries of the different regions of the dataset can not be well defined by a cube selection cursor, it is important to keep the size of the cube small, and make the selection of the region in multiple passes, in order to get proper connectivity. Connectivity is the most crucial part of the volume animation pipeline, and unless it is done as best as possible, the reconstruction process will yield unsatisfactory results.

As the points are connected to the bones, they will get colored with the same color as the bone. If a cube has been wrongly connected, click on 'remove' to remove all connectivity information from the enclosed points. To make the process of placement of the cube easier, start with a higher thinness value, and decrease as the points get connected, to trap the points close to the surface.

Once a region has been defined, move on to the next segment by repeating from the root selection step.

To save the connectivity information, click 'save', and enter the filename in the console window. By convention, these files are named as "[sequence].[thinness].connect", thinness being the position of the slider when the save was clicked. Only those points having thinness greater than the value of the slider are saved when the button is clicked.

Note: As of now, this program can not load up a connect file, so the only way to re-connect a skeleton to it's articulate counterpart is to re-start from scratch and connect all over again. For instance, if a volume is all connected, the connect file saved and skelselect exited, and then the reconstruction phase indicates slight breakage in the shoulder, then skelselect has to be loaded up again with the original bones file and the mskel file, and the process of connecting the skeleton points has to be started all over again.

To avoid this, it's advisable to have skelselect running continuously until reconstruction is carried out with satisfactory results.

Quick tip: To toggle between the selection and the move cursors of **skelselect**, hit the escape key. This is standard with most applications that are built on the **Inventor** interface.

A.42 SortTskel

Source: /caip/u60/ksen/male/src/euclid/SortTskel.cpp

Binary: /caip/u60/ksen/bin/SortTskel

Author: Kundan Sen

Usage: /teal/caip10/ksen/tec/recon/summer00/SortTskel [tskel file] [sorted output]

This program sorts the skeleton points associated to each root node of the tskel file in decreasing order of their distance transforms, and prints the new tskel file to *[sorted output]*.

It has been observed that it is easier to measure progress of reconstruction when the skeleton points associated to a single root node are thus sorted. This way, when a root node is selected, the speed of reconstruction of the spheres associated to it follow an exponential curve, since reconstruction time per sphere decreases exponentially with the radius of the sphere. The progress is indicated by the number of points reconstructed, which is printed

at regular intervals to the standard output by the reconstruction programs, Reconstruct1 and ReconstuctEuclid.

A.43 teleportMan

Source: /caip/u60/ksen/male/src/teleportMan.cpp

Binary: /caip/u60/ksen/bin/teleportMan

Author: Kundan Sen

Usage: teleportMan [tskel file] [number of frames (20)] [sequence header (meltFrame)] [desiredframe])

This program works similar to meltMan in creating the melting sequence by decreasing the distance transforms associated to each skeleton point.

A.44 TiffToIfl

Source: /teal/caip10/ksen/work/volume/src/TiffToIfl.cpp

Binary: /teal/caip10/ksen/work/volume/bin/TiffToIfl

Author: Kundan Sen

Usage:: TiffToIfl [tiffFile] [iflFile]

Converts a tiff image file into IFL (Image Format Library) format. See **man ifl** for more information on supported IFL formats.

A.45 TightBounds

Source: /teal/caip10/ksen/work/volume/src/TightBounds.cpp

Binary: /teal/caip10/ksen/work/volume/bin/TightBounds

Author: Kundan Sen

Usage: TightBounds [old volume] [new volume] [-silent]

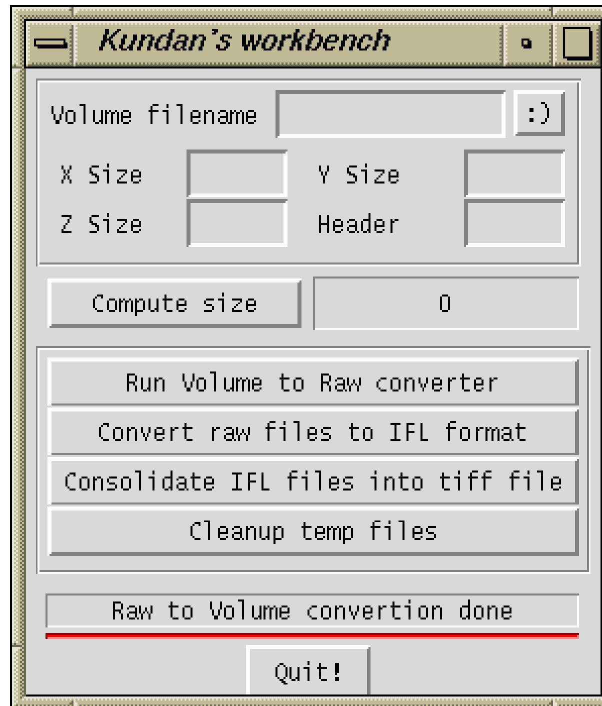

Figure A.2: Toolbar.tcl - User Interface for Volume to tiff Converter

Compacts a volume by removing the blank space around it. The volumes are expected to have the standard 100-Bytes headers - which are generated by AddHeader.sh.

A.46 Toolbar.tcl

Source: /teal/caip10/ksen/apps/vtcl/src/toolbar.tcl

Binary: /teal/caip10/ksen/apps/vtcl/src/toolbar.tcl

Author: Kundan Sen

Usage: toolbar.tcl

This program brings up a basic user interface for scripts to convert a volume in .vol format to a 3D tiff format. The interface looks like this:

The program loads up with blank values in the textfields. The volume filename can either be typed in at the textfield, or the file selected from a browse popup window which can be brought up by clicking on the browse button to the right of the filename textfield. The size of the volume is then entered in the appropriate textfields, and the 'Compute Size'

button is clicked. This is to confirm that the size of the file has been entered correctly - the number in the size textfield should be the same as the actual size of the file, seen with a command like **ls -l [filename]** in the shell.

Once the volume file has been entered, the buttons pointing to the scripts are clicked in sequence. After each button is clicked, the status indicator text shows what is being done, and the status indicator bar goes red. If the program being executed ends succesfully without errors, the bar will go green, and the interface is ready for the next stage. Errors reported from the program are brought up in pop-up message boxes, if any.

A.47 View.tcl

Source: /caip/u60/ksen/bin/gagvani/view.tcl

Also: /teal/caip11/gagvani/quota/bin/view.tcl

Binary: /caip/u60/ksen/bin/gagvani/view.tcl

Author: Nikhil Gagvani

Usage: view.tcl [vtk volume file]

This is a very simple Vtk/ Tcl program to view a volume a slice at a time. To convert a volume to the vtk format, use the utility Vol2Vtk. The program fires up a graphical screen with a slider to inspect the volume. The volume is applied a linear gray colormap, with the range [0-255] depicted as [black - white]. The slider below the volume panel can be dragged to view a particular slice in the volume.

To view binary segmented [0-1] volumes, convert the voxels having value '1' to value '255', using an utility like Convert2Binary, and having 255 as the replace parameter.

A.48 Vol2Vtk

Source: Contact Nikhil Gagvani

Binary: /teal/caip11/gagvani/quota/bin/vol2vtk

Author: Nikhil Gagvani

Usage: Vol2Vtk [volume file] [vtk file] [xSize] [ySize] [zSize]

Converts a volume from .vol format to the .vtk format, the latter being compatible to the **Visualization Toolkit**. See vtk user guides for more information on the vtk format. The first few lines of the vtk file contain information such as the size of the volume, spacing, size of data at each voxel, and the like. This information can be extracted by a command like **head -5** [**vtkfile**].

A.49 VolumeSuperDiff

Source: /teal/caip10/ksen/work/volume/src/euclid/VolumeSuperDiff.cpp

Binary: /caip/u60/ksen/bin/VolumeSuperDiff

Author: Kundan Sen

Usage: VolumeSuperDiff [vol1] [vol2] [xSize] [ySize] [zSize] [out Volume]

Computes the difference between two volumes of the same size, on a voxel by voxel basis. The results are printed to a third volume, also of the same size. The voxels in the output volume can have one of 4 values: **0**, indicating the voxel has the same value in both the volumes.

100, for voxels that have a non-zero value in the first volume only.

150, for voxels that have a non-zero value in the second volume only.

200, for voxels that have different non-zero values in the two volumes.

A.50 Vtk2Vol

Source: Contact Nikhil Gagvani

Binary: /teal/caip11/gagvani/quota/bin/vtk2vol

Author: Nikhil Gagvani

Usage: vtk2vol [vtk file] [volume file]

Converts a vtk file to a .vol format. The .vol format is a binary file with raw data,

ordered by x, then y, then z, without any header, and consisting of one byte of unsigned char data per voxel.

A.51 Zap

Source: /caip/u60/ksen/male/src/Zap.cpp

Binary: /caip/u60/ksen/bin/Zap

Author: Kundan Sen

Usage: Zap [volume file] [xSize] [ySize] [zSize]

Replaces in-place all data in the file with zeros. Use with caution, as the original volume will be lost. Particularly useful when working with parametric computational solid geometry programs, and it is desired to re-start from a clean slate.

Appendix B

Troubleshooting

- The Skeleton Does Not Appear Centered - It's More Like A Surface Mesh.

  The segmented volume used for skeletonization has holes in it. Regions within the volume fell below the threshold while performing segmentation, leaving these holes. If the threshold can be reduced, try segmenting at a lower threshold.

  Verify absence of holes in the resultant binary volume by loading up the slices in a viewer like **view.tcl** after converting the 0-1 binary volume to 0-255 binary volume, using a program like **Convert2Binary** to convert the 1's to 255's.

  If the resultant volume still has holes in it, use **Fill** to fill in the holes manually. Verify proper filling with **view.tcl**, following steps above.

  If the threshold required for producing a completely filled volume fails to exclude some speckled points in the boundary, use **Fill**, with '0', to erase the noise from the boundary.

  When a neatly filled in volume is passed to any of the skeletonization programs, a neatly centered skeleton, with points radiating out all the way to the surface, will be produced.

- The Size of the Color Volume is Much Larger Than the Product of it's Dimensions

  Follow steps described under the volume animation pipeline to reduce the volume to 8-bit color. If the volume is already in 8-bit color, either the size information is incorrect, or there is a header preceding the data values. Guessing the correction in either case is near to impossible unless the size of the correct volume is known, or can be computed otherwise. If the size of the volume is known, and it can be concluded that the size difference is due to a header, use **rmhdr** to remove the header.

Note: if the volume has been given a header by 'AddHeader.sh' (in which case, by convention, it should be named as "[VOLUME NAME IN CAPITALS].vol") the size of the header is exactly 100 Bytes.

- Skelselect Gives A Core Dump and Exits When Loaded Up With the Skeleton and the Bones File

  All the points in the articulated skeleton must be contained in the cloud of skeleton points loaded from the mskel file. In case the articulated skeleton was produced by selecting points not present in the skeleton cloud, insert these points at the very end of the mskel file. Assign a Distance transform (DT) less than 3, and a thinness less than the smallest thinness, to these points. A token thinness of 0.01 and token DT of 1 will ensure non-participation of these points in the reconstruction process.

  Note: If these additional points are given high DTs, they will take active role in the reconstruction process, and can significantly alter the appearance of the final volume.

- The final volume appears broken at the joints

  This may result from several different factors, and is the most common trouble to run into while running the pipeline. The most common causes of this are:

  a) The connectivity is incorrect. Re-connect the skeleton points to the articulate skeleton using **skelselect**. While connecting near the breakage, adjust to compensate for the breakage - if, for example, the shoulder breaks to leave a part of the upper arm fixed to the shoulder while the rest of the arm moves away, move the points in the upper arm to the bone segment in the upper arm instead of the bone of the shoulder. If, on the other hand, a part of the shoulder moves away with the arm, do the reverse - lessen the attachment to the upper arm. Every such case is different, and has to be resolved on a case-by-case basis.

  Use the tools **tskel2iv** and **iskel2sph** to verify proper connectivity in the surface domain before going through the time-intensive sampled reconstruction process

  b) The choice of points for the articulate skeleton can be improved upon. If, for instance the point in the knee is selected too much to the front, the knee is bound

to break when the foot is bent forward - this is because the reconstruction process does not compensate for stretching of the joints, but merely rotates them about the articulate skeleton points. Move the points in the articulate skeleton as much to the center as possible. On special cases, such as when it's known that the foot will only bend backwards, move the knee point to the best position for that particular sequence, that minimizes breakage at the point.

- The final volume, when rendered, does not make any sense - it looks like a speckled object without any sensible coloring

  The colormap is incorrect in some way. In most cases, this is because the colormap is not sorted by visual gradient of the colors - the anti-aliasing effect of the rendering engine produces colors that are blends of the neighboring voxel color indices, and lead to random values. See section under "Sorting the colormap" in the volume animation pipeline on how to sort the colormap and resolve this issue.

- The final volume has flipped palms for hands and feet

  This may result from several factors, but the most common case is misaligned axes - the coordinate system of the motion capture sequence is different from that of the back-end processing and reconstruction. Usually, this difference can be compensated by one or more rotations - in steps of 90 or 180 degrees - in each of the axes. Modify the code in **ivskelgetrot** to compensate for this rotation before comparing the segments to figure out the transformation matrices. Look at the code for **ivskelgetrot** for comments on how to do this.

- The final volume has some of the segments rotated out of the volume

  This is also caused by the difference of coordinates discussed above.

- The final volume has the torso reversed / rotated in crazy angles

  This can happen when the abdomen and chest area is selected as a single bone in the articulated skeleton. This is because, when two straight lines are compared for rotations, the rotations about themselves (i.e. about an axis that coincides with the

length of the line) can not be figured out. This would not happen if the animation package returned the exact rotations each segment undergoes. However, in our pipeline to date, the animation pipeline only returns the point coordinates, and the exact transformations are figured out by reverse-computation, and so rotations about the axis of any segment can not be figured out correctly.

To avoid this, select the articulate skeleton as a zigzag line through the chest and abdomen region.

- Error in executing extractpoints.tcl - “no such point”.

  The output of the animation package has to follow the same naming convention for points as defined in the connections file. Check for inconsistencies in naming the points - common errors are like naming ‘LEFT_FOOT’ in the connections file and ‘LEFT_LEG’ in the animation sequence, or ‘TORSO1’ and ‘CHEST1’.

- The animation has been rendered, but the camera keeps swinging for every frame.

  The current rendering engine uses the VTK toolkit to render the volume. The camera in this package is always centered at the center of the volume. If each frame in the sequence has been reconstructed to a final volume of a different size, then the camera will adjust itself to center on each of these volumes, and create a swinging effect. To avoid this effect, use **tskelbounds** or **euclidtskelallbounds** to compute the volume bounds for the entire sequence of frames, and not just a single frame.

- The reconstruction program reports voxel out-of-bounds errors

  The bounds of the volume are not correct. Check the bounds with **tskelbounds**. Make sure the bounds for sequences using the Euclidean distance metric are computed with the *-e* flag in *tskelbounds*, otherwise the program will (incorrectly) return the 3-4-5 bounds. Similarly, passing the *-e* flag to a 3-4-5 volume will return an incorrect result.

References

[1] The National Library of Medicine's Visible Human Project. *http://www.nlm.nih.gov/research/visible/visible_human.html.*

[2] D. Meagher Geometric Modeling Using Octree Encoding. *Graphical Models and Image Procesing*, 19:129–147, 1982.

[3] C. Arcelli and G. Sanniti di Baja. A Width-Independent Fast Thinning Algorithm. *IEEE Transactions on Pattern Recognition and Machine Intelligence*, 7(4):463–474, 1985.

[4] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In Maureen C. Stone, editor, *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169, July 1987.

[5] J. Lasseter Principles Of Traditional Animation Applied To 3D Computer Animation *ACM Computer Graphics*Volume 21, Number 4, July 1987

[6] R. Drebin, L. Carpenter, and P. Hanaran. Volume Rendering. *Computer Graphics (SIGGRAPH 88 Conference Proceedings)* ,pages 65–74. ACM SIGGRAPH, 1988.

[7] T.H.H. Cormen, R.L. Rivest and C.E. Leiserson. Introduction to Algorithms MIT Press November 1990.

[8] F. Aurenhammer. Voronoi Diagrams: A Survey of a Fundamental Geometric Data Structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[9] T. Beier and S. Neely. Feature-based Image Metamorphosis. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):35–42, July 1992.

[10] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware . In Arie Kaufman and Wolfgang Krueger, editors, *Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, October 1994.

[11] S. Marschner, R. Lobb. An Evaluation of Reconstruction Filters for Volume Rendering. In *Proceedings of Visualization*, pages 100-107, October 1994

[12] E. Horowitz, S. Sahni and D. Mehta. Fundamentals of Data Structures in C++ *W. H. Freeman Company*, February 1995.

[13] J.L. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. *Morgan Kaufmann Publishers*, August 1995.

[14] G. Borgefors. On Digital Distance Transforms in Three Dimensions. *Computer Vision and Image Understanding*, 64(3):368–376, November 1996.

[15] N. Gagvani. Skeletons and volume thinning in visulaization. *Thesis (MS)–Rutgers University*, 1997.

[16] N. Gagvani, D. Kenchammana-Hosekote, and D. Silver. Volume Animation Using The Skeleton Tree . In *IEEE Volume Visualization Symposium*, pages 47–54, October 1998.

[17] A. Brentzen et. al. The Carpeaux Gallery. *http://lin1.gk.dtu.dk/home/jab/carpeaux/gallery.html*, August 1995.

[18] M. Heller. Developing Optimized Code with Microsoft Visual C++ 6.0. In *Microsoft Developer Network online library (MSDN)*, March 1999.

[19] N. Gagvani and D. Silver. Parameter Controlled Volume Thinning. *Graphical Models and Image Procesing*, 61(3):149–164, May 1999.

[20] B. Stroustrup. The C++ Programming Language *Addison Wesley Longman, Inc.*, December 1999.

[21] V. Chandru, N. Mahesh, M.Manivannan, and S. Manohar. Voxel-based Sculpting and Keyframe Animation System *Computer Animation Conference*, May 2000.

[22] N. Gagvani and D. Silver. Shape-based Volumetric Collision Detection. In *Proc. IEEE Volume Visualization Symposium*, pages 57–61, October 2000.

[23] N. Gagvani. Parameter-Controlled Skeletonization – A Framework for Volume Graphics *Thesis (PhD)–Rutgers University*, 2001

[24] A. Bhattacharya. An Interactive Volume Animation Toolkit. *Thesis (MS)–Rutgers University*, 2001